

嵌入式系统系列

Patterns for Time-Triggered Embedded Systems

Building reliable applications with
the 8051 family of microcontrollers

时间触发嵌入式系统设计模式

〔英〕 Michael J. Pont 著
Kent Beck 序
周敏 译

使用 8051 系列微控制器开发可靠应用



附赠 CD 包含全部源代码及
Kent Beck 教程和零件模型



中国电力出版社
www.infopower.com.cn

Patterns for Time-Triggered Embedded Systems

Building reliable applications with the 8051 family of microcontrollers

时间触发嵌入式系统设计模式 使用 8051 系列微控制器开发可靠应用

“这些模式可以作为一个例子来说明使用模式能够比一般的尝试完成更多的事情。模式在问题和解决方案之间架起了一座桥梁，将人们的需求和情感与技术连接在一起，并为那些有问题要解决的人提供新的思路。”

——引自 Kent Beck 所作的序

本书前所未有的提出了一整套软件模式，为开发嵌入式软件系统提供帮助。书中讨论了基于广为使用的 8051 系列微控制器进行设计并实现嵌入式应用软件的方法，此外还重点关注了其可靠性。

书中一共有超过 70 个软件模式，并介绍了如何将这些技巧应用到你自己的项目中。作者为迅速创建各种各样的嵌入式应用提供了很多实用的资料和建议。本书从简单系统到复杂系统，列举了大量详尽的实例，主要内容包括：

- 针对使用一个或多个微控制器的嵌入式应用，设计实现完整的调度操作系统。
- 采用开关、键盘、LED 显示、LCD 等元件创建用户界面。
- 有效地使用网络和通信协议。
- 在监控系统设计中应用 PID 算法脉冲宽度调制。

本书特点：

- 通过列举大量的例子来说明如何将特定模式应用到实际项目中。
- 在相关的网站 (www.engg.le.ac.uk/books/Pont) 上包括了众多的详细案例研究。

随书附送的 CD 上包括：

- 所有模式和例子的 C 语言完整源代码，包含一系列完整的调度器。
- 工业标准的 Keil C 编译器和硬件模拟器的一个评估版本，因此不需要额外购买硬件就可以测试书中的例子。

Michael J. Pont 是 Leicester 大学电子与软件工程专业的高级讲师。在过去的 10 年中其研究涉及多个领域的软件设计和编程，并为一批跨国公司提供了咨询与培训服务。由 Michael 编著及合著的技术文献的数量已超过 70 种，其中包括《C 语言嵌入式系统开发》（已由中国电力出版社出版）和《C++ 与 CASE 工具软件工程》。



ISBN 7-5083-2206-1



9 787508 322063 >

责任编辑 / 朱恩从
封面设计 / 王红柳

ISBN 7-5083-2206-1

定价：85.00 元（含 1CD）

嵌入式系统系列

Patterns for Time-Triggered Embedded Systems

Building reliable applications with
the 8051 family of microcontrollers

时间触发嵌入式系统设计模式 使用 8051 系列微控制器开发可靠应用

[英] Michael J. Pont 著
Kent Beck 序
周敏 译

Patterns for Time-Triggered Embedded System (ISBN 0-201-33138-1)

Michael J. Pont

Authorized translation from the English language edition, Patterns for Time-Triggered Embedded System, published by Addison Wesley. Copyright©2001

All rights reserved.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

CHINESE SIMPLIFIED language edition published by China Electric Power Press Copyright©2004

本书由美国培生集团授权出版。

北京市版权局著作权合同登记号 图字：01-2003-3634 号

图书在版编目 (CIP) 数据

时间触发嵌入式系统设计模式：使用 8051 系列微控制器开发可靠应用 / (英) 庞特 (Pont, M. J.) 著；周敏译. 北京：中国电力出版社，2004

(嵌入式系统丛书)

ISBN 7-5083-2206-1

I.时... II.①庞... ②周... III.微控制器—系统设计 IV.TP332.3

中国版本图书馆 CIP 数据核字 (2004) 第 029860

丛书名：嵌入式系统丛书

书 名：时间触发嵌入式系统设计模式：使用8051系列微控制器开发可靠应用

编 著：(英) Michael J. Pont

翻 译：周敏

责任编辑：朱恩从

出版发行：中国电力出版社

地址：北京市三里河路6号 邮政编码：100044

电话：(010) 88515918 传 真：(010) 88518169

印 刷：北京丰源印刷厂

开 本：87×1092 1/16 印 张：49 字 数：1110 千字

书 号：ISBN 7-5083-2206-1

版 次：2004 年 6 月北京第 1 版 2004 年 6 月第 1 次印刷

定 价：85.00 元 (含 ICD)

版权所有 翻印必究

前言

现在你面对着的是一种模式语言。尽管它和我自己的模式主题有相当大的距离，然而从本质上讲，Michael 的目标是对的。

Ward Cunningham 和我以前在早期商业化 Smalltalk 时曾一起工作。Smalltalk 从一开始就被设计为一种无缝环境。你可以使用由 Smalltalk 编写的文字处理软件启动调试程序，修改该程序并继续输入。

Tektronix 的一些最初使用 Smalltalk 的用户相当古怪。我们经常谈到 Ray，这个从一家大化学公司来的老家伙理解 Smalltalk 并确实使它跳转和运行、处理并显示实验数据。看他的演示是一种乐趣，因为他为自己所完成的工作而骄傲。

读 Ray 的程序代码却是另一回事。为了使程序工作，他会做任何事，而无论多么骇人听闻。结果是程序一片混乱，完全不可维护而且只使用了 Smalltalk 的一小部分功能。

我们经常把 Ray 作为某些软件开发人员的象征——有问题需要解决而不得不设计软件的人。这种功利的态度与我们对软件的“不妥协精神”有着显著区别。在我们看来，力求解决方案的简明而优雅比解决该问题更重要。然而如果我们想影响众人，就不能只追求自己的美感，必须同时设法帮助 Ray。

最后所得到的模式语言是深思熟虑的建议（决不使用你不能亲自关掉的计算机）和老套的日常文档书写注意事项（将源代码中的括号对齐为矩形格式）的古怪的混合物。目的是帮助 Ray 从 Smalltalk 当中得到更多功能。但在这方面我们基本上失败了，从那时起我的职业生涯就转为四处漂泊，为写程序的人们提供建议和辅导。

这就是为什么我爱读 Michael 的文稿的原因。它为那些正好有问题要解决，然而又不想成为该解决方案的专家的人答疑解惑。现在我就是 Ray。我希望把微控制器聚集到一起用于解决各种各样的问题（好的，所以我是个书呆子），读过这些模式语言后使我有信心能把它做好。

这些模式决不只是使我鼻子中有松香的气味，或是手中有绕线枪的感觉，而是举例说明使用模式将比一般的尝试能够完成更多的事情。模式在问题和解决方案之间架起了一座桥梁。将人们的需要和情感与技术连接在一起，并为那些有问题要解决的人们提供新的思路。

现在就点热烙铁并享受开发的乐趣吧！

Kent Beck
三河研究所
Merlin, 俄勒冈州

序言

时间触发嵌入式系统模式——用 8051 系列微控制器创建可靠的应用系统

Michael J. Pont
由 Kent Beck 致前言

嵌入式软件无所不在。它是无数系统的核心部分，这些系统包括飞机、汽车、医疗设备、孩子们的玩具、录像机和微波炉等。本书提供了一整套互有联系的软件模式，用于支持这类应用的开发。

序言的其余部分将为读者对目录可能存在的具体疑问提供解答。

1 本书的主要特点

- 本书主要关注使用软件模式迅速地开发时间触发嵌入式系统的软件。第 1 章解释了时间触发的含义，第 2 章介绍了软件模式。
- 所有的系统都基于广为应用的 8051 系列微控制器。许多公司生产这种 8 位芯片，包括 Philips、Infineon、Atmel、Dallas、Texas Instruments 和 Intel 等公司。第 3 章回顾了现有的各种 8051 微控制器。
- 时间触发技术通常用于有安全要求的应用场合，即可靠性是一个关键性的设计要求。然而，对可靠性的要求并不局限于电传操纵汽车、航天系统或工业机器人用监测系统。即使在最低端，那些不能准时闹铃的闹钟或是运行过程中时断时续的录像机虽然不存在安全问题，但是同样也不会有好的销路。这里给出的模式将简单而高效地把时间触发技术应用到几乎所有嵌入式设计项目中。
- 详细讨论必须在 ms 级的时间间隔里执行任务或响应事件的应用系统。使用本书讨论的方法，即使使用 8 位微控制器，也能够经济而可靠地实现这种级别的响应。
- 软件全部用 C 语言编写，附送的 CD 上包括本书所有的例子。
- 本书由互联网站提供支持，站点上包括：各种详细的例子研究，其他的技术信息和资料来源的链接 (<http://www.engg.le.ac.uk/books/Pont>)。

2 如何创建时间触发嵌入式系统？

- 本书中的时间触发系统使用调度器创建。简要地说，调度器是一种非常简单的“操作

系统”，适合于嵌入式应用（关于这一主题的详细介绍请参见第 13 章）。

- 用于单个微控制器应用系统的系列完整的调度器结构在第 14 章～第 17 章中描述并讨论，CD 上包含了一些不同调度器的完整源代码。
- 越来越多的应用系统使用多个微控制器，这里介绍的许多系统也涉及使用多个微控制器：第 25 章～第 29 章描述了能够支持此类应用的一系列共享时钟调度器结构。许多这种系统都使用通用串行标准，包括 CAN 总线和 RS-485。
- 在第 8 篇中，挑选了一些特殊的调度器结构加以介绍。包括一种能够长时间提供非常精确定时的“稳定”调度器，一种为运行单个任务优化的调度器和为低功耗及低存储器应用而设计的通用调度器（参见第 36 章和第 37 章）。

3 本书讨论的其他问题

- 所有嵌入式系统都涉及硬件设计，本书适当给出了一些硬件基础知识。包括设计振荡器和复位电路，以及连接外部 ROM 和 RAM 存储器的技术（参见第 4 章、第 5 章和第 6 章）。此外，本书还包括了分别适用于低压和高压交、直流负载的接口电路（参见第 7 章和第 8 章）。
- 同时也适当给出了软件基础知识，包括用于嵌入式应用的一种简单结构（第 9 章），用于控制端口引脚的技术（第 10 章），用于产生延迟的技术（第 11 章）和使用看门狗定时器的技术（第 12 章）。
- 一些嵌入式应用的主要用户接口是连接到桌面或者笔记本 PC 的 RS-232 接口，而其他一些嵌入式系统使用 LCD 或者 LED 显示和一组开关或键盘创建用户接口。第 18 章～第 22 章中给出了用于处理这些不同接口模块的技术。
- 各种外围器件（LCD、LED 显示、EEPROM、模数转换器和数模转换器等等）现在都具有串行接口，因此这些器件可以不需要大量端口引脚便能连接到微控制器上。用于两种主要串行通信协议（I²C 和 SPI）的完整的软件库在第 23 章和第 24 章中给出。
- 第 7 篇给出了用于状态监视和控制应用的技术，包括对“PID 控制”的讨论。同样，也提供了详细的代码库（第 30 章～第 35 章）。

4 本书的读者对象

在写这本书时，我设想了三类读者：

- 具有桌面系统经验的软件工程师，现在开始设计嵌入式系统。
- 希望更多地理解有关嵌入式系统开发中的软件问题的硬件工程师。
- “电子和软件工程”、“软件工程”、“计算机科学”、“电子工程”专业或者涉及嵌入式系统高级模块的类似课程的大专院校学生。

必须注意，本书不能用作编程入门，其读者应已经具有使用 C、C++ 或者类似高级语言开

发桌面软件应用的经验。建议缺少这一领域经验的读者在阅读本书时准备一本有关 C 语言的介绍性书籍，诸如 Herbert Schildt 的《Teach Yourself C》(Schildt, 1997)^①。

类似地，假定读者熟悉软件设计的原则。这里，如果有一些“面向对象”设计和“面向过程”设计（“结构化分析”）的经验将很有用。建议缺少这一领域经验的读者准备一本以前我写的关于软件设计的介绍性书籍 (Pont, 1996)。

最后，有一些非常基本的电子学知识也是十分有用的。建议缺少硬件设计经验的读者阅读《The Art of Electronics》(Horowitz and Hill, 1989)。

在大多数情况下，具有桌面编程经验、熟悉“数据流程图”或“UML”，以及具有基本硬件知识的读者将比较容易掌握本书的内容。请注意，并不要求读者具有软件模式的知识。

5 采用哪种类型的微控制器硬件？

微控制器市场是巨大的。据估计，每卖出一个用于台式 PC 的处理器的同时，将卖出 100 个微控制器用于嵌入式系统。

正如副标题所提示的，本书主要关注 8051 系列微控制器。该微控制器最初由 Intel 公司开发，然而现在有超过 300 种不同类型的 8051 系列微控制器，并且由多家不同的公司生产，包括 Philips、Infineon、Atmel 和 Dallas。在设计中使用 8051 系列微控制器没有风险。巨大的 8051 家族的销售加在一起估计占有 8 位微控制器市场的 50% 以上，在整个微控制器市场拥有最大的份额（大约 30%）。

注意，本书不仅讨论了最新版本的“标准”8051（4 个端口、40/44 个引脚，如 Atmel 的 89C52；Dallas 的 89C420；Infineon 的 C501；Philips 的 89CRD2），还讨论了现代微控制器完整的系列，包括“精简”8051（2 个端口、20/24 个引脚，如 Atmel 的 89C4051；Philips 的 87LPC764），以及“扩展”8051（多达 10 个端口，~100 个引脚，片内集成了 CAN、模数转换器等等，如 Infineon 的 C509；Infineon 的 C515c；Dallas 的 80c390）。

请注意：本书的相关代码全部用 C 语言编写。将代码转换到其他硬件平台上使用十分简单。

6 CD上的内容

CD 上包括了所有软件模式的完整源代码文件。如上所述，所有代码均用“C 语言”设计。

这些模式的源代码和作为工业标准的 Keil C 编译程序完全兼容，CD 上包括该编译程序的一个评估版本和一个完整的硬件模拟器，这使得根本不需要购买或者设计任何硬件，就可以在台式 PC 上研究大多数模式。

^① 文中所参考文献的详细来源在第 39 章中给出。

目录

前 言
序 言

绪 言

第 1 章 什么是时间触发的嵌入式系统	3
1.1 引言	3
1.2 信息系统	3
1.3 桌面系统	5
1.4 实时系统	5
1.5 嵌入式系统	7
1.6 事件触发系统	8
1.7 时间触发系统	10
1.8 小结	11
第 2 章 使用模式来设计嵌入式系统	12
2.1 引言	12
2.2 现有软件设计技术的局限	14
2.3 模式	17
2.4 时间触发嵌入式系统模式	20
2.5 小结	20

第 1 篇 硬 件 基 础

第 3 章 8051 系列微控制器	25
引言	25
标准 8051	25
精简 8051	34
扩展 8051	38
第 4 章 振荡器硬件	44
引言	44
晶体振荡器	44
陶瓷谐振器	52
第 5 章 硬件复位	55

引言	55
阻容复位	55
可靠的复位	62
第 6 章 存储器问题	66
引言	66
片内存储器	66
片外数据存储器	75
片外程序存储器	81
第 7 章 直流负载驱动	88
引言	88
直接 LED 驱动	88
直接负载驱动	93
IC 缓冲放大器	95
BJT (双极结型三极管) 驱动器	100
IC 驱动器	109
MOSFET 驱动器	113
固态继电器驱动 (直流)	117
第 8 章 交流负载驱动	121
引言	121
电磁继电器驱动	121
固态继电器驱动 (交流)	127

第 2 篇 软件基础

第 9 章 基本的软件体系结构	133
引言	133
超级循环	133
项目头文件	138
第 10 章 使用端口	142
引言	142
端口输入/输出	142
端口头文件	150
第 11 章 延迟	158
引言	158
硬件延迟	158
软件延迟	167
第 12 章 看门狗	175
引言	175
硬件看门狗	176

第3篇 单处理器系统的时间触发结构

第13章 调度器的介绍	187
13.1 引言	187
13.2 桌面操作系统	187
13.3 对超级循环结构的评价	188
13.4 更好的解决方案	190
13.5 例子：闪烁LED	194
13.6 在不同的时间间隔执行多个任务	196
13.7 什么是调度器	198
13.8 合作式调度和抢占式调度	199
13.9 抢占式调度器详解	202
13.10 小结	204
13.11 进阶阅读	204
第14章 合作式调度器	206
引言	206
合作式调度器	206
第15章 学会以合作的方式思考	240
引言	240
循环超时	240
硬件超时	245
第16章 面向任务的设计	255
引言	255
多级任务	255
多状态任务	259
第17章 混合式调度器	267
引言	267
混合式调度器	267

第4篇 用户界面

第18章 通过RS-232与PC通信	289
引言	289
PC连接(RS-232)	289
第19章 开关接口	317
引言	317
开关接口(软件)	318
开关接口(硬件)	326
通断开关	329
多状态开关	335

第 20 章 键盘接口	343
引言	343
键盘接口	343
第 21 章 多路复用 LED 显示	355
引言	355
多路复用 LED 显示	355
第 22 章 控制 LCD 显示面板	367
引言	367
字符型 LCD 面板	368

第 5 篇 使用串行外围模块

第 23 章 使用 I²C 外围模块	389
引言	389
I ² C 外围模块	389
第 24 章 使用 SPI 外围模块	410
引言	410
SPI 外围模块	410

第 6 篇 多处理器系统的时间触发体系结构

第 25 章 共享时钟调度器的介绍	425
25.1 引言	425
25.2 额外的 CPU 性能和外围硬件	425
25.3 模块化设计的优点	426
25.4 怎样连接多个处理器	428
25.5 为什么增加处理器并不一定能改善可靠性	434
25.6 小结	436
第 26 章 使用外部中断的共享时钟调度器	437
引言	437
共享时钟中断调度器（时标）	437
共享时钟中断调度器（数据）	467
第 27 章 使用 UART（通用异步收发器）的共享时钟调度器	479
引言	479
使用 UART 的共享时钟调度器（本地）	479
使用 UART 的共享时钟调度器（RS-232）	504
使用 UART 的共享时钟调度器（RS-485）	507
第 28 章 使用 CAN 的共享时钟调度器	530
引言	530

共享时钟 CAN 调度器.....	531
第 29 章 多处理器系统的设计.....	557
引言.....	557
数据联合.....	557
长任务.....	560
多米诺骨牌任务.....	563
第 7 篇 监视与控制组件	
第 30 章 脉冲频率检测.....	569
引言.....	569
硬件脉冲计数.....	569
软件脉冲计数.....	575
第 31 章 脉冲频率调制.....	580
引言.....	580
硬件脉冲频率调制.....	580
软件脉冲频率调制.....	585
第 32 章 模拟-数字转换器 (ADC) 的应用.....	591
引言.....	591
单次模数转换.....	591
模数转换前置放大器.....	606
序列模数转换.....	610
A-A 滤波器.....	618
电流传感器.....	625
第 33 章 脉冲宽度调制.....	629
引言.....	629
硬件脉宽调制.....	629
脉宽调制信号平滑滤波.....	637
3 级脉宽调制.....	640
软件脉宽调制.....	646
第 34 章 数模转换器的应用 (DAC)	653
引言.....	653
数模转换输出.....	653
数模转换平滑滤波.....	663
数模转换驱动.....	666
第 35 章 进行控制.....	669
引言.....	669
PID 控制器.....	669

第 8 篇 特殊的时间触发结构

第 36 章 减少系统开销	695
引言	695
255-时标调度器	695
单任务调度器	707
一年调度器	712
第 37 章 提高调度的稳定性	722
引言	722
稳定调度器	722

结 论

第 38 章 本书试图实现的目标	731
38.1 引言	731
38.2 本书试图实现的目标	731
38.3 小结	732
第 39 章 收集的参考文献和书目	733
39.1 出版书刊一览表	733
39.2 其他模式	739
39.3 实时嵌入式系统设计技术	739
39.4 高可靠性系统设计技术	740
39.5 8051 微控制器	741
39.6 作者的相关著作	741

附 录

A 设计表示法以及 CASE 工具	745
概述	745
CASE 工具	745
表示法	745
B CD 指南	763
概述	763
CD 的主要内容	763
本书的源代码	763
C 互联网站点指南	764
概述	764
URL	764
互联网站点上的内容	764
错误报告和程序代码更新	764

绪 言

本篇中的两个介绍性章节回答了下列问题：

- 什么是嵌入式系统？
- 什么是时间触发系统？还有哪些其他的系统结构？
- 为什么通常认为时间触发系统比基于其他结构的系统更可靠？
- 什么是软件模式？
- 怎样借助模式来创建可靠的嵌入式系统？



1

Chapter

什么是时间触发的嵌入式系统

在这个介绍性的章节中，将讨论“嵌入式系统”和“时间触发系统”的含义，并研究这两个重要的领域有哪些重叠之处。

1.1 引言

当前的一些软件系统的分类常常令人迷惑：

- 信息系统
- 桌面应用程序
- 实时系统
- 嵌入式系统
- 事件触发系统
- 时间触发系统

在这些各种各样的领域之间有相当多的重叠。本章将简要地讨论上述六种应用系统，而将对时间触发嵌入式系统（time-triggered embedded system）的讨论放到本书的其余章节。

1.2 信息系统

信息系统（information system, IS），特别是“商业信息系统”代表了大量的应用系统。尽管信息系统开发中所遇到的挑战与在本书中将要关注的问题有很大的不同，但对这类系统有一个基本的了解是很有用的，这是因为目前大多数嵌入式实时系统的开发方法都是由最初的信息系统领域的开发方法改编而来。

作为基本信息系统的一个例子，考虑图 1.1 所示的用于计算工资报表的应用程序。

假定，这个应用程序使用用户提供的和系统存储的雇员资料来打印公司的工资报表。票据的打印可能要花费好几个小时。如果在财务年度的末尾要求做一套特别复杂的计算，打印将因此被延迟几分钟，这很可能会造成不方便。在后面的例子中将把这些“不方便”和实时系统中的延迟可能造成的破坏性影响做一个对比。

信息系统往往和针对保存在磁盘文件中的大量数据所进行的存储与处理联系在一起。在 20

世纪 60 年代和 20 世纪 70 年代，一般使用诸如 COBOL 这样的面向文档的语言来实现。今天，这样的系统仍在广泛应用，不过大多数是处于维护阶段，而很少使用类似语言做新的应用设计。

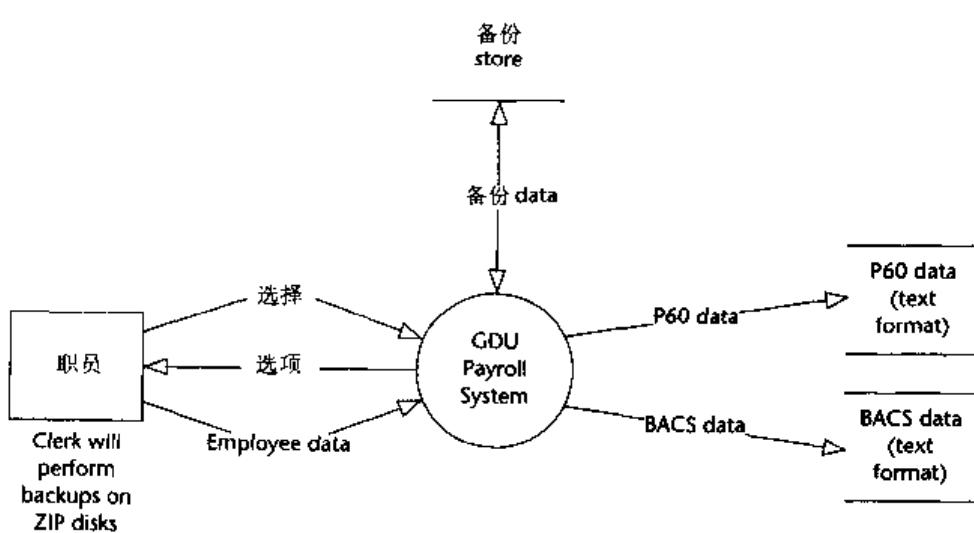


图 1.1 一个简单的工资报表系统的高级示意图（数据流图）详细说明参见附录 A

实现现代信息系统的更常用方法是使用关系数据库（relational database），通过 SQL 语言进行数据的存取和操作。关系数据库技术建立在严谨的数学基础之上，成熟而且安全。设计实现可靠的大型关系数据库系统决非易事，但开发用于家庭和小型企业的应用系统所需的技巧却不复杂。

因此，像这样的小型关系数据库系统的实现不再成为一种专门处理。如今，关系数据库的设计工具作为标准 office 软件包的一部分，为许多台式计算机用户广泛使用。

然而，信息系统的设计人员又将面对新的要求。例如，除了传统的文字档案，许多医院希望能够存储信号波形[例如：心电图信号（ECG）或听觉反应信号]或者图像（例如，X 光或磁共振图像），以及其他来自医学检查的复杂数据。图 1.2 列出的是一个心电图信号的扫描轨迹。



图 1.2 心电图信号的一个例子

关系数据库系统可以用于存储图像、语音或波形，这种系统针对处理有限种类的数据类型

(诸如字符串、字符、整数和实数)而做的优化效果并不理想。这使得人们对面向对象的数据
库系统(对象数据库)的兴趣逐渐增加。通常认为面向对象的数据库系统使用起来更加灵活。

1.3 桌面系统

在许多信息系统中，桌面/工作站环境和通用桌面应用程序(诸如文字处理软件)占主导地位。现代桌面环境的普遍特征是用户通过高分辨率的图形屏幕、键盘和鼠标与应用程序交互(图1.3)。

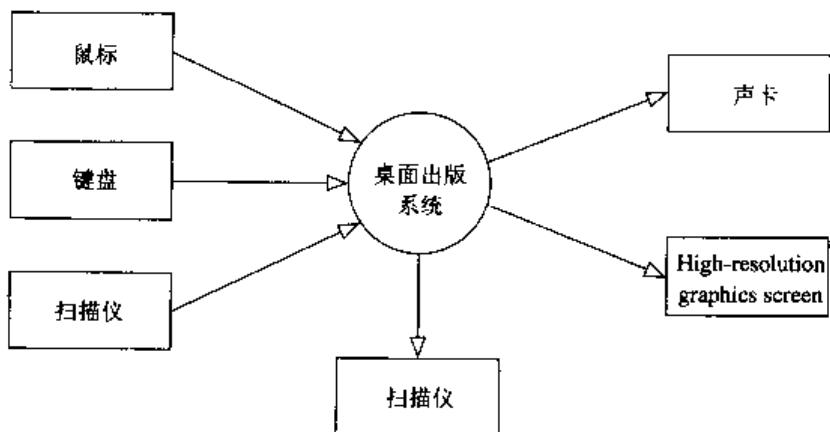


图1.3 图形用户界面驱动的桌面应用程序的部分设计的例子

除了复杂的用户交互之外，桌面系统独特的特征是相关的操作系统，其范围从DOS到某一版本的Windows或UNIX操作系统。

正如所看到的，嵌入式系统的开发人员所面对的系统很少用到操作系统、屏幕、键盘或鼠标。

提示

$$1 \text{ 秒 (s)} = 1.0 \text{ 秒} (10^0 \text{ 秒}) = 1000 \text{ 毫秒}$$

$$1 \text{ 毫秒 (ms)} = 0.001 \text{ 秒} (10^{-3} \text{ 秒}) = 1000 \text{ 微秒}$$

$$1 \text{ 微秒 (\mu s)} = 0.000001 \text{ 秒} (10^{-6} \text{ 秒}) = 1000 \text{ 纳秒}$$

$$1 \text{ 纳秒 (ns)} = 0.000000001 \text{ 秒} (10^{-9} \text{ 秒})$$

1.4 实时系统

大多数软件系统的用户都希望他们的应用程序能够快速响应，不同的是：对于大多数信息
系统和一般的桌面应用程序而言，快速反应仅仅是个有用的性能。但对于许多实时系统，它却
是必要的性能。

例如，考虑图1.4中所示的经过极大简化的飞机自动驾驶系统。

这里假定飞机已经进入所需航向而系统为了使飞机保持航线必须做出频繁且有规律的变

化来操作方向舵、升降舵、副翼和发动机的工作状态。

这个系统的重要特点是需要以 ms 为时间标度，非常迅速地处理输入和产生输出。在这种情况下，即使方向舵偏转角度的调整有微小延迟也可能导致难以接受的飞机振动，极端情况下甚至会导致坠毁。由于需要进行快速处理，所以几乎没有哪个软件工程师不认为自动驾驶系统可以代表大多数的实时系统。

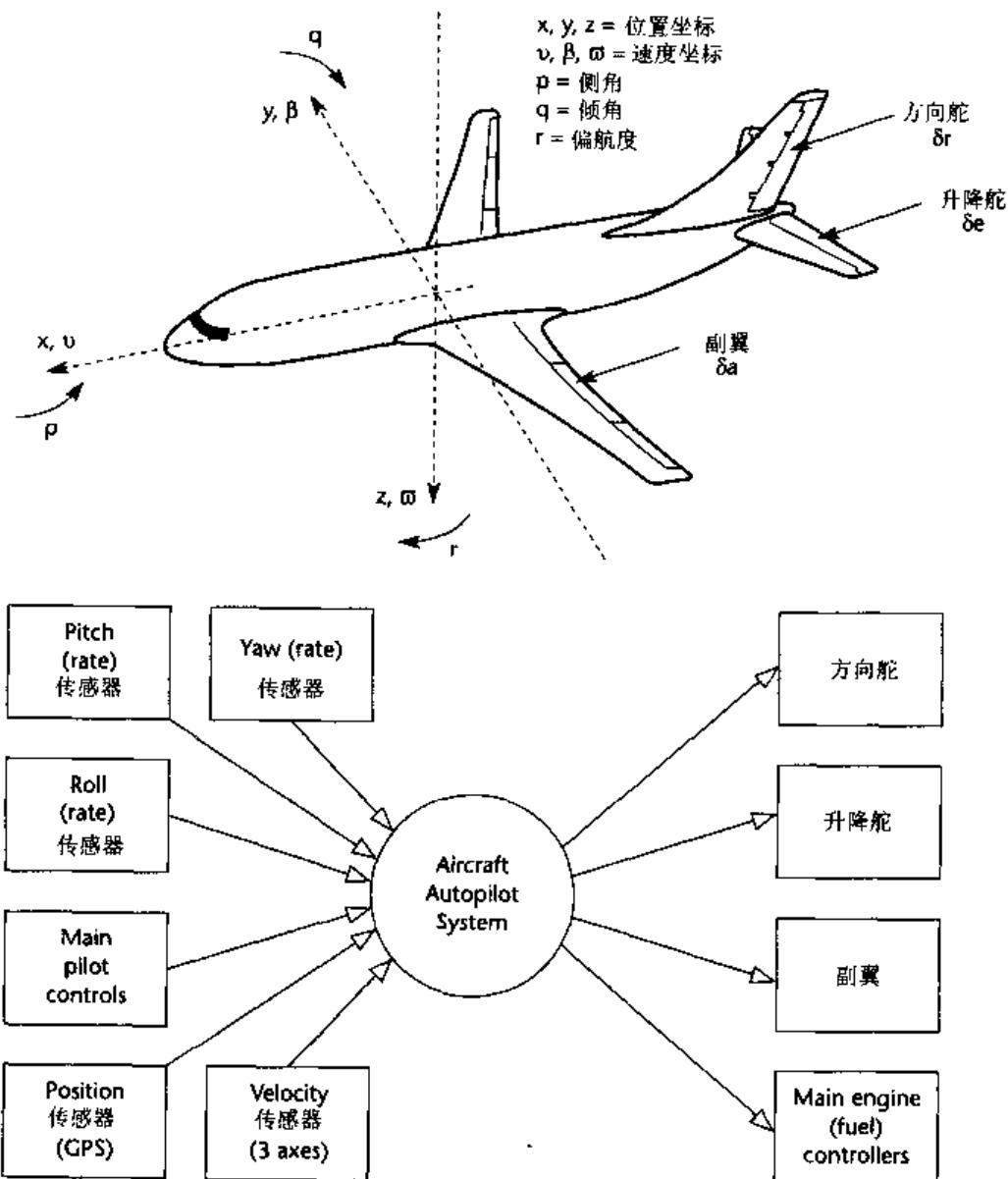


图 1.4 一个简单的自动驾驶系统的高级示意图

在实践中为了验证飞行系统能够正确工作，仅仅保证尽可能快的处理速度是不够的。正如其他许多实时系统一样，在实时情况下，主要性能是处理的确定性。这意味着在许多实时系统中需要能够保证某个处理总是在（比方说）2ms 内完成或者精确的每隔 6ms 完成。如果处理

过程不能满足这个要求，那么该系统就不单单是比我们想要的缓慢，而是毫无价值。

在实践中，Tom De Marco 提供了对这种硬实时要求的图形化描述，并引用了一个软件项目经理的话：

我们建立了驻留在小型遥测计算机上的系统，该系统配备有各种各样的传感器用来测量电磁场和温度、声音变化，以及机械损伤。我们分析这些信号并且通过宽带连接将结果回传到远程计算机上。计算机安装在一米长的杆的一端，而杆的另一端是一个核装置。将它们一起放在地下的大洞中，当装置被引爆时计算机在爆炸波的前沿采集数据。爆炸后的前 2.25ms 是最值得关注的。当然，对于计算机来说，在远未到达第三个 ms 之前，信号已经迅速平稳下来。我们认为这就是实时约束。

(引用自 De Marco, 为 Hatley 和 Pirbhai 写的前言, 1987)

在这种情况下，很明显该实时系统必须及时完成数据记录：它没有第二次尝试的机会。这是个极端的例子，有时被称作是一种“硬”实时系统。

注意，许多应用系统（如前面提到的飞机系统）和这个军事应用的例子不同，涉及对现实世界中的数据源的重复采样（通过传感器和模数转换器），经过数字处理之后产生适当的模拟输出信号（通过数模转换器和制动器）。假定以 1000Hz 对输入信号进行采样，那么作为一个实时系统，必须在下一个采样（0.001s 之后）到来之前处理输入并产生相应的输出。

作为总结，参考下列对实时系统进行定义的“词典”中的内容：

一种对外界正在发生的事件做出反应的程序。例如，飞机中的自动驾驶程序为了修正航向偏差必须迅速响应。过程控制、机器人、游戏和许多军事应用系统都是实时系统的例子。

(《Hutchinson 新世纪百科全书》，光盘版 1996)

需要强调的是：无论是对设计人员还是对最终客户，仅仅希望快速处理本身并不能正确地描述“实时性”。这常常被误解，甚至软件行业的开发人员也不例外。例如，Waites 和 Knott 写到：

“一些商业信息系统也要求实时控制……典型实例包括航空机票预订和那些要求快速流动的库存管理系统。”

(Waites 和 Knott, 1996, 第 194 页)

实际上，这些系统都不能称为确切的实时系统。

1.5 嵌入式系统

虽然普遍与实时系统有关，但嵌入式系统的分类如同桌面系统一样，例如，它也包括实时的和特殊的信息系统。嵌入式系统的独特特性概括如下：

嵌入式系统是一种包含至少一个可编程计算机（一般为微控制器、微处理器或数字信号处理器芯片）的系统，使用者基本上不关心系统是基于计算机的。

这种嵌入式系统的典型例子包括普通家用电器，诸如录像机、微波炉和冰箱。其他的例子

包括汽车、联合收割机、飞机和各种防御系统。请注意这个定义排除了诸如“掌上电脑”这类的系统。从开发人员的角度来看，它们只是台式计算机系统的精简版本。

为了支持复杂的操作系统和应用程序，台式机市场是由对更高性能的需求来驱动的，而嵌入式系统市场却有着不同的需要。例如，现代汽车工业中的相关法规和技术改造意味着越来越多的汽车将包含嵌入式系统。有时候，引入这种系统主要是用来降低生产成本。例如，在现代化的汽车中，昂贵的 (~£600.00) 多芯电缆现在被一种两线式的控制器局域网 (CAN) 计算机总线替代，其成本是原来的几分之一 (Perier 和 Coen, 1998)。在其他场合，比如引入主动悬挂系统，通过嵌入式系统来改善乘坐的舒适性和操纵性能 (Sharp, 1998)。

为了有助于说明嵌入式市场的一些要求，现在讨论一个非常简单的例子：一种用于汽车的信号灯电路。在这个应用系统中，需要通过方向盘后面的开关控制六个或更多的信号灯，使司机打算转弯、改变车道或停车时可以通知路上的其他人。对于美国（和其他）市场，要求方向信号灯电路与尾灯共同起作用（这样单个灯闪烁用来指示转向的方向）；在欧洲，则要求方向指示灯和尾灯分别工作。此外，在一些国家，希望使用信号灯作为“停车灯”，以避免在晚上停下时被别人撞上。

沃尔沃 131 (亚马逊) 展示了解决这个问题的传统方案。这种 20 世纪 60 年代经典的欧洲车采用了大量的导线和机械开关来控制信号灯和制动器动作。如果想要把这种汽车调整为以美式风格运行，则需要改动大量导线套管的布置。为了避免这种代价昂贵、劳动密集的工作，现代化的汽车通过使用微控制器来提供所需的控制动作。微控制器解决方案不但简单、便宜、减少了导线，而且还可以只通过按下一个开关或改变存储器芯片就能够在美式信号灯标准和欧式信号灯标准之间进行转换。

这种简单的应用系统突出了嵌入式系统的四个重要性能。

- 许多系统和这种信号灯系统一样，使用微控制器的原因不是因为处理复杂，而是因为微控制器比较灵活。更重要的是，它是一种性价比很高的解决方案。因此，许多产品中的嵌入式微控制器在大部分使用寿命中几乎都是空闲着的。事实上，大多数通常使用的微控制器与现代的桌面微处理器相比确实要缓慢得多，然而这并没有什么可担心的。
- 与大多数微处理器不同，微控制器与外界相互作用时，并不通过键盘和图形用户界面，而是通过开关、小型按键、LED 等等。提供各种各样的 I/O 功能是许多微控制器厂商推陈出新的主要动力。
- 和信号灯系统一样，大多数嵌入式系统都要求在精确的时间间隔或特定瞬间执行某种任务。例如，信号灯以精确的频率和占空比闪烁来满足法律上的要求。这类应用系统将在第 2 章中更详细地讨论。
- 与大多数桌面系统不同，许多嵌入式系统有安全上的要求。例如，如果在开车的时候信号灯失灵，则将导致事故。因此，可靠性是很多嵌入式系统的关键要求。

1.6 事件触发系统

如今，有很多应用系统都被描述为“事件触发的”或“事件驱动的”。例如，就现代桌面

系统而言，各种各样的应用程序在运行中必须对诸如单击鼠标或移动鼠标这样的事件做出反应。用户希望这样的事件将引起“即时的”响应。

在嵌入式系统中，事件触发行为往往通过使用中断（参见以下框内内容）实现。事件触发系统在系统总体结构上往往通过提供多级中断服务程序来支持该功能。

什么是中断？

从底层来看，中断是一种硬件机制，用来通知处理器发生了一个“事件”。这样的事件可能是“内部的”事件（诸如定时器的溢出），也可能是“外部的”事件（诸如串行接口接收到一个字符）。

从上层的角度来看，中断提供了一种创建多重任务应用的机制。也就是说，从表面上看，系统可以在单个处理器上同时执行多个任务。图 1.5 给出了一个嵌入式系统中断处理的原理示意图来说明这个问题。

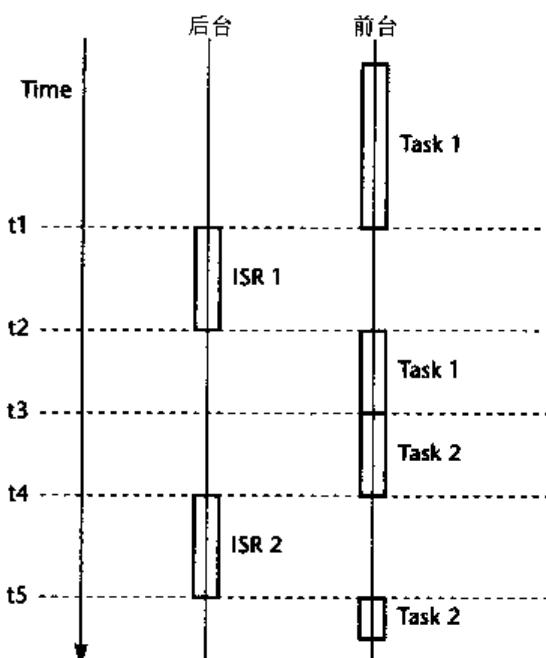


图 1.5 嵌入式系统中断处理的原理示意图

在图 1.5 中，系统执行两个（后台）任务：任务 1 和任务 2。在执行任务 1 期间，有一个中断产生，“中断服务程序”（ISR1）处理这个事件。在执行任务 2 期间，另一个中断产生，这次由 ISR2 处理。

注意，从程序员的角度来看，中断服务程序只不过是一个由某个硬件事件引发的、“由微控制器调用”的函数。

1.7 时间触发系统

事件触发系统结构的主要替代方案是时间触发结构（参见 Kopetz, 1997）。和事件触发结构一样，时间触发结构既可以用于桌面系统也可以用于嵌入式系统。

以下这个例子有助于理解这两种结构之间的区别：医院里的一个医生，必须在一些护理人员的帮助下，通宵照管十个危重病人。这个医生完成这个任务时可以考虑两种方案：

- 安排护理人员在某个病人出现严重问题时唤醒她。这就是“事件触发”解决方案。
- 把闹钟设置为每小时闹铃。当闹钟闹铃时，起床去依次探访每一个病人，检查他们是否舒服，如果需要的话，则给予治疗。这就是“时间触发”解决方案。

对于大多数的医生来说，事件触发方法似乎更有吸引力，因为这样他们在整个夜晚可以睡上几个小时。相比而言，如果使用时间触发方法，医生将不可避免地失眠。

然而，对于许多嵌入式系统来说它们并不需要睡眠，采用时间触发方法有很多好处。实际上，在那些特别关心安全性的工业部门，诸如航空航天工业和汽车工业，时间触发方法得到了广泛的使用，因为这个方法有助于改善可靠性和安全性（对于这个问题的讨论参见 Allworth, 1981; MISRA, 1994; Storey, 1996; Nissanke, 1997; Bates, 2000），所以被系统开发人员和权威认证机构所接受。

在安全相关的应用系统中，时间触发方法成为首选的主要原因是系统的行为可以预计。再看看那家医院的例子，你就明白为什么会是这样。

例如，假设由“事件触发”的医生正在安睡。病人有一个表面上看起来较小的问题，护理人员决定不叫醒医生而自己处理该问题。两个小时之后，当有四个病人存在“小”问题时，护士最终决定不得不叫醒医生。医生一看到病人就意识到其中两个有严重的并发症，因而不得不开始手术。在完成第一个病人的手术之前，第二个病人已经非常濒临死亡了。

看看同样例子里的“时间触发”医生。因为以每小时的间隔探访病人，所以医生能在严重的并发症出现之前探访每个病人并安排合理的治疗。从另一个角度来看，工作量在整个晚上平摊。于是，所有病人都能平安地度过这个夜晚。

在嵌入式系统中，这种医院（更确切地说是以死亡为主题的）场合反映到事件驱动的应用系统中，就是同时发生几个事件（也就是说同时产生几个中断）。例如，在一架飞机上同时检测到两个不同的故障或者一种简单的情况（如，同时按下小键盘上的两个按键）。

为了理解为什么当两个中断同时产生时会出现问题，先看一下这种情况下 8051 体系结构的工作过程。与许多微控制器相似，最初的 8051 体系结构支持两种不同的中断优先级：低优先级和高优先级。如果两个中断（称做中断 1 和中断 2）迅速地连续发生，系统将做如下处理：

- 如果中断 1 是一个低优先级中断，而中断 2 是一个高优先级中断：

低优先级中断激活的中断服务程序（ISR）可以被高优先级中断打断。在这种情况下，为了让高优先级中断服务程序执行，低优先级中断服务程序将暂停，低优先级中断服务程序将在高优先级中断服务程序之后完成。在大多数情况下，系统将正确地运行（只要这两个 ISR 不互相干扰）。

- 如果中断1是一个低优先级中断，而中断2也是一个低优先级中断：
由低优先级中断激活的ISR不能被另一个低优先级中断打断。于是，对第二个中断的响应将被延迟；在一些情况下将被完全忽略。
- 如果中断1是一个高优先级中断，而中断2是一个低优先级中断：
由高优先级中断激活的ISR不能被低优先级的中断打断。于是，对第二个中断的响应将被延迟；在一些情况下将被完全忽略。
- 如果中断1是一个高优先级中断，而中断2也是一个高优先级中断：
由高优先级中断激活的ISR不能被另一个高优先级的中断打断。于是，对第二个中断的响应将被延迟；在一些情况下将被完全忽略。

注意：当心，这意味着什么！嵌入式系统的开发人员普遍有一种错误观念，即中断事件决不会被丢失。这完全不正确。如果多个中断源可能在“随机的”时间间隔产生中断，则中断响应可能被遗漏。实际上，在同时有几个有效的中断源的情况下，几乎不可能创建程序代码来正确地处理所有可能的中断组合。

需要处理同时发生的多个事件不但增加了系统复杂性，而且降低了对事件触发系统在所有情况下的行为做出预计的能力。相比而言，在时间触发嵌入式系统中，设计人员能够通过仔细安排可控的顺序，保证一次只处理一个事件。

正如已经提到的，时间触发系统的可预测的特性使这种方法成为安全相关的系统的通常选择，在这些系统中可靠性是关键的设计要求。然而，对可靠性的要求不仅仅局限于诸如飞机的电传操纵系统和汽车的自动驾驶系统，甚至在低档的应用系统中也有要求。例如，没能准时响铃的闹钟、运行时断时续的录像机或者（每年一次）丢失几个字节数据的监控系统，这些产品虽然没有安全上的隐患，但同样不可能畅销。

除了能够提高可靠性之外，使用时间触发方法将有助于降低CPU的负荷并减少存储器的使用量。因此，正如贯穿于本书中所展示的，即使在小型嵌入式系统中采用这种系统结构，也能从中获益。

1.8 小结

本章介绍的时间触发的嵌入式系统的各种特性将在本书的其他部分详细讨论。

在下一章里，将讨论为什么“传统的”软件设计方法仅仅能够为这类应用系统的开发人员提供有限的支持，并说明使用软件模式能够为现有方法提供有益的帮助。

Chapter 2

使用模式来设计嵌入式系统

在第二个介绍性章节里，将讨论为什么“传统的”软件设计技术仅仅能为嵌入式系统开发人员提供有限的支持，并说明使用软件模式能够为现有方法提供有益的帮助。

2.1 引言

大多数工程分支都具有悠久的历史。例如，控制系统领域也许可以说是早在 1760 年左右詹姆士·瓦特对飞轮调速器的早期研究就开始了，而电机工程可以回溯到迈克尔·法拉第的工作，通常认为是他于 1821 年发明了电动机。在所有起源当中，土木工程的实际应用是否具有最悠久的历史仍有争议，或许它起源于埃及人建造金字塔，或者希腊、罗马时代。当然，于 1818 年在英国创立的土木工程师学会是世界上最古老的专业工程学会。

对于软件工程师来说，情况完全不同。第一种大量生产的小型计算机 PDP-8 于 1965 年投放市场，而第一种微处理器于 1971 年投放市场。由于小型的可编程计算机系统起步相对较迟，而其后的发展非常迅速，因此软件工程领域难以发展成熟。在这短短的几十年时间里，软件工程的大量工作集中在设计过程，特别是开发和使用各种图形表示法。包括面向过程的表示法，诸如数据流程图 (Yourdon, 1989; 参见图 2.1)，以及面向对象的表示法，诸如“统一建模语言”(Fowler 和 Scott, 2000)。这些表示法建立在“方法学”的基础之上，即软件设计方法的汇集，详述在什么情况下以及如何在设计项目中运用特定的表示法 (参见 Pont, 1996)。使用这种方法而产生的设计由一组相连结的图表组成，这些图表遵循标准的表示法，并附有相应的配套文档 (Yourdon 1989; Booch, 1994; Pont, 1996; Fowler 和 Scott, 2000)。

正如本章标题所示，本书涉及嵌入式系统软件的开发。在过去，尽管嵌入式系统无所不在，但针对这种系统的设计并没有成为软件领域的主要焦点。甚至于几乎在所有的情况下，软件设计技术已经发展为首先满足桌面系统开发人员的需要 (DeMarco, 1978; Rumbaugh 等, 1991; Coleman 等, 1994)，然后再“被修改”来满足嵌入式实时应用开发人员的需要 (Hatley 和 Pirbhai, 1987; Selic 等, 1994; Awad 等, 1996; Douglass, 1998)。第 2.2 节将证明这样最后所得到的软件设计技术虽然并不乏优点，但是不能直接满足嵌入式系统设计人员的需要。然后将在第

2.2节和第2.4节中建议通过利用软件模式作为现有技术的补充，提出了解决当前一些问题的比较有发展前途的方法。

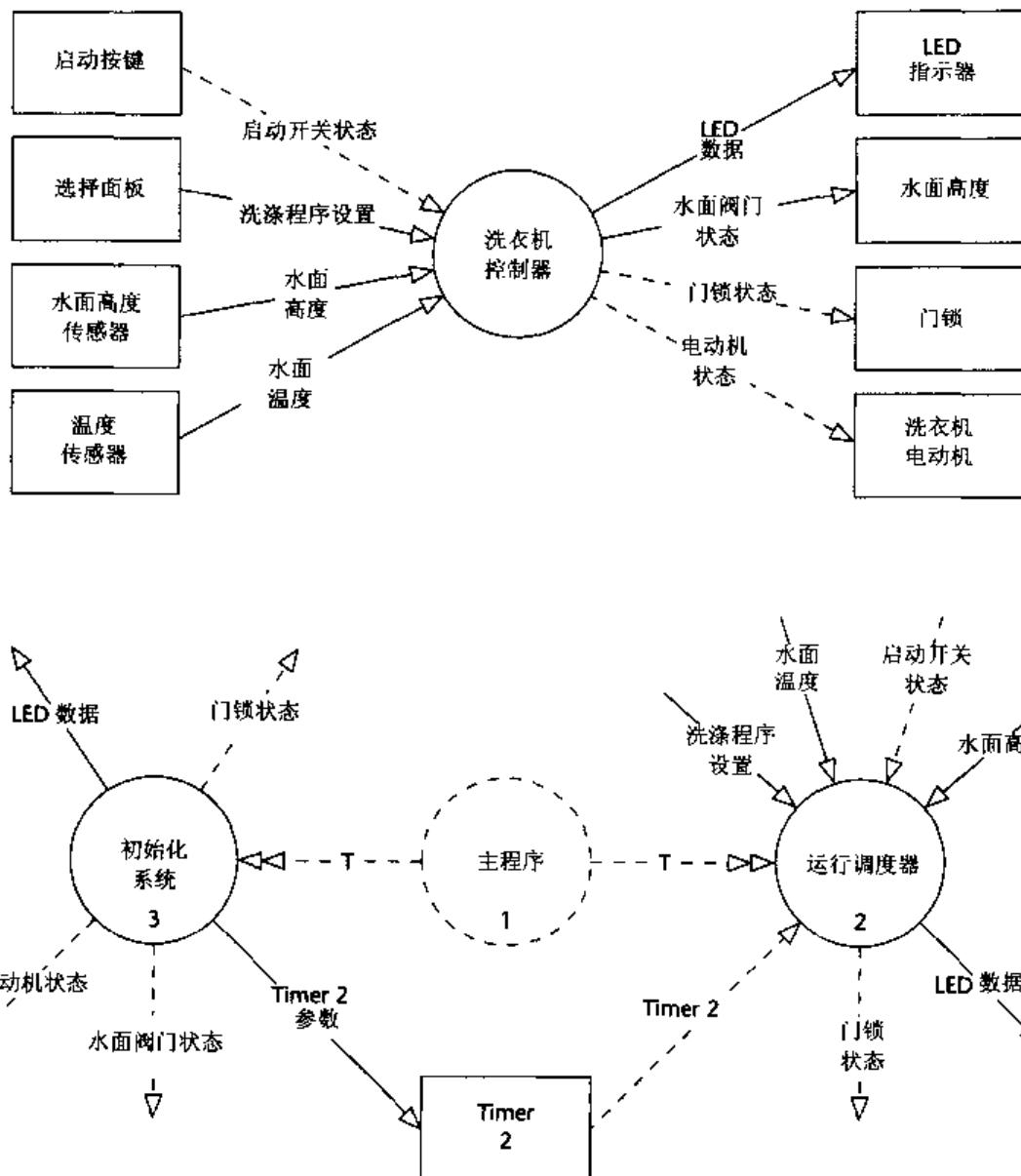


图 2.1 洗衣机系统的部分设计的例子，上半部分显示了接口关系的数据流图，下半部分显示了第一层的数据流图

注意，在这个例子里，使用面向过程的（数据流）表示法^①来描述设计方案，这种表示方法将贯穿于本书。它仍然是嵌入式系统的最常用方法，在某种程度上是因为面向过程的语言（特别是 C 语言）是这个领域流行的设计语言。虽然面向对象的语言（如 C++）目前在基于微控制器的嵌入式项目中相对少见，但面向对象的设计表示法同样可以用来描述这里介绍的设计。

^① 这种表示法的细节请参阅附录 A。

2.2 现有软件设计技术的局限

下面通过讨论两个例子来引出本章的主要内容,这些例子说明了使用标准设计技术来开发嵌入式系统时的局限性。

巡航控制系统

作为第一个例子,考虑一种用于机动车的巡航控制系统(cruise-control system, CCS)。巡航控制系统经常被用来说明实时软件设计方法学的有效性(参见 Hatley 和 Pirbhai, 1987; Awad 等, 1996)。通常要求这种系统接受维持机动车速度恒定的任务,以适应地形的变化(例如,道路上的斜坡或转弯)。巡航控制根据某种情况(典型地,当机动车在高速档并超过了预置的最低速度时)动作,此外,还要求能够由驾驶员通过仪表板上的按钮开关启动巡航控制,并通过踩制动踏板关闭它。

作为一个例子,在图 2.2~图 2.5 中说明了一个针对这种系统的面向过程的(结构化的)简要设计。这个设计是由 Hatley 和 Pirbhai (1987) 编写的一本关于实时软件设计的标准教科书所介绍的内容改编而来。图 2.2 从最抽象的层次出发,展示了该系统的“接口关系数据流图”。图 2.3 给出了相应的第一层数据流图,描述了图 2.2 中的“简单巡航控制”流程的更详细的处理过程。同样,图 2.4 给出了与图 2.3 中的主要控制过程相对应的状态转移图。

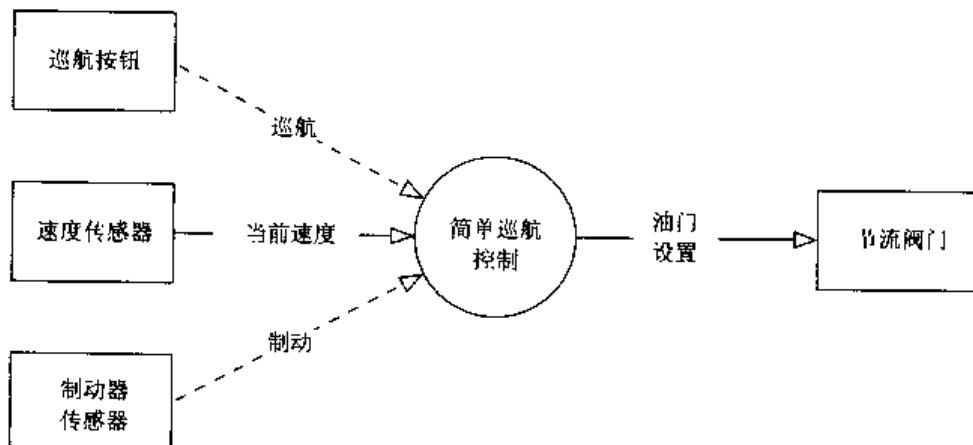


图 2.2 简单巡航控制系统的接口关系数据流图

必须更精确地定义“设置速度”和“保持速度”两个流程的操作运行,来完成这个简单的设计。这两个流程比较起来,后一个流程更加复杂,将在下面进行讨论。Hatley 和 Pirbhai (1987) 为他们的“保持速度”流程给出了如图 2.5 所示的“流程说明”。

总体说来,这个设计方案有一些问题。例如,很少考虑所使用系统的总体结构,以及由此造成的对设计其他部分的影响。此外, Hatley 和 Pirbhai (1987) 所介绍的控制算法版本的核心部分基本上没有注解(参见图 2.5)。他们没有采用“标准”控制技术,诸如“比例-积分-微分”(PID) 控制,作为这个问题的解决方案^②,而且没有指出该如何评估所选定方法的有效性。

^② PID 控制算法将在第 35 章中讨论。

(甚至是否应该做评估)。这在软件教科书中并不罕见。例如，大约十年以后，Awad等(1996)更加详细地描述了一个用于同样的巡航控制系统的面向对象的设计，再一次基本上忽视了这些问题。

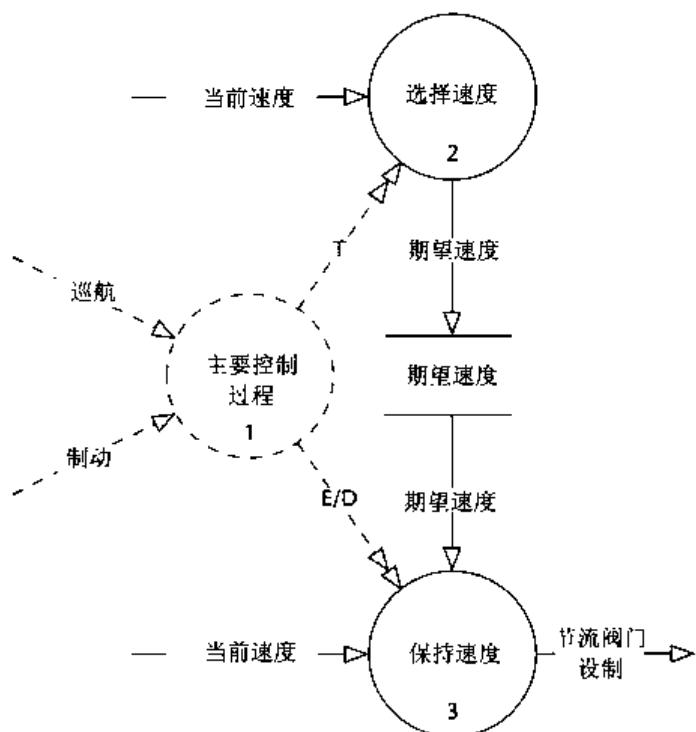


图 2.3 巡航控制系统的第一层数据流图

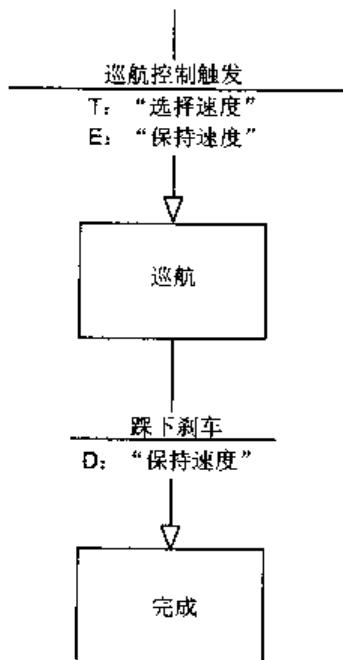


图 2.4 与图 2.2 中的控制过程相对应的状态转移图

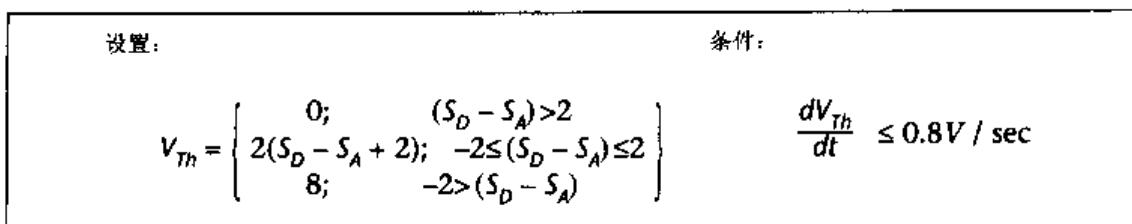


图 2.5 “保持速度”流程的流程说明（改编自 Hatley 和 Pirbhai, 1987, 第 291 页）

注意, V_{Th} = 油门设置; S_D = 期望速度; S_A = 实际速度。这里假定油门设置与输出电压成比例, 即 0V 输出为关闭油门, 而 8V 输出把它完全打开。流程的目的是当实际速度超过或低于期望速度 2mph 时改变油门位置。油门的变化速度被限制为小于 0.8V/s。

为了成功地设计嵌入式系统（比如这个巡航控制系统），不仅要求具备软件工程和微控制器硬件知识，而且还需要借助相关的工程技术领域，通常包括仪表设备、数字信号处理、人工智能和控制理论。根据我们的经验，这种借助对于实时软件项目的成功往往是必要的。然而它们并不在大多数软件工程师的正规训练之列，也常常被实时软件设计方面的书籍和论文所忽视。

闹钟

毫无疑问，巡航控制系统是一种特殊的产品。大多数开发人员显然没有在汽车领域的从业经验，因而不会涉及这方面的应用。然而，类似的问题同样出现在最简单的嵌入式系统中。

例如，假定要求研制一种闹钟系统，运行方式如下：

- 在 LED 上显示时间。
- 用户可以调节时间。
- 闹铃将在用户设定的时刻闹响（可选）。

图 2.6 给出了要求的用户界面的草图。

按传统的方法来创建这样一个应用的设计，将从画接口关系数据流图开始（图 2.7）。

和巡航控制系统一样，该表示法只帮助我们描述设计方案，不能为制定方案提供帮助。例如，在这个应用中，基本的要求是在 LED 上显示信息。在大多数情况下，为了减少成本将使用多路复用显示。正如将在第 21 章论述的，对于一个多路复用、4 位数字的 LED 显示，必须每隔约 5ms 刷新一次显示。在设计过程的开始阶段，就必须考虑要求以这种频率更新显示对整个系统的软件体系结构将产生的影响。如果系统的设计人员没有意识到这个基本要求，那么设计中许多以此为基础的设想将可能是错误的，这样当项目到达实施阶段时，将需要大量昂贵且费时的重新设计。

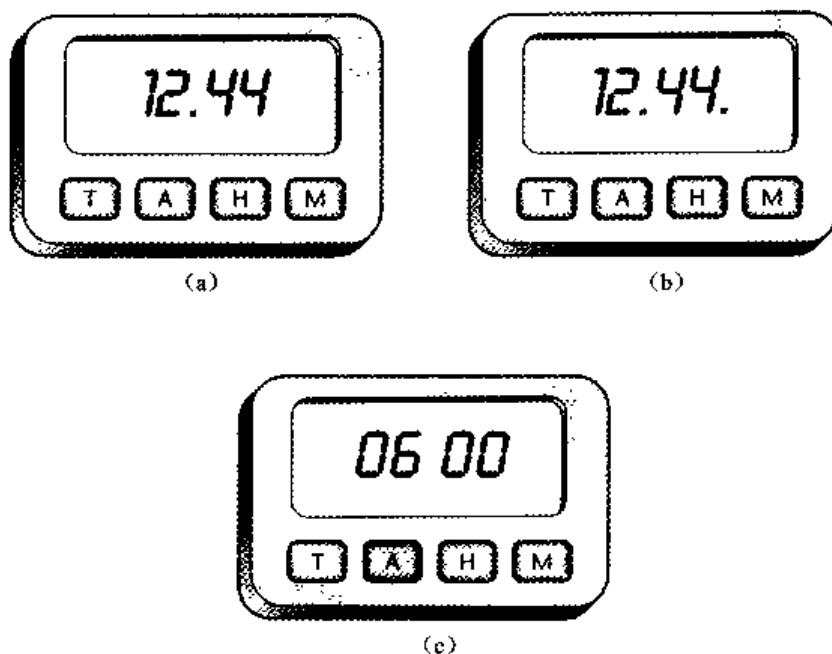


图 2.6 要求的用户界面

(a) 当闹钟没有被设置时的正常显示（当前时间）；(b) 当闹钟被设置时的正常显示（当前时间）；

(c) 当 A 按钮被按下时显示闹铃时间

注意，按下 A+H 按钮将允许用户改变闹铃时间（小时）；A+M 按钮同样将改变闹铃时间（分钟）。同样的操作，使用 T 按钮将用来改变显示的时间。当闹钟闹铃时按下 A 按钮将停止闹铃。

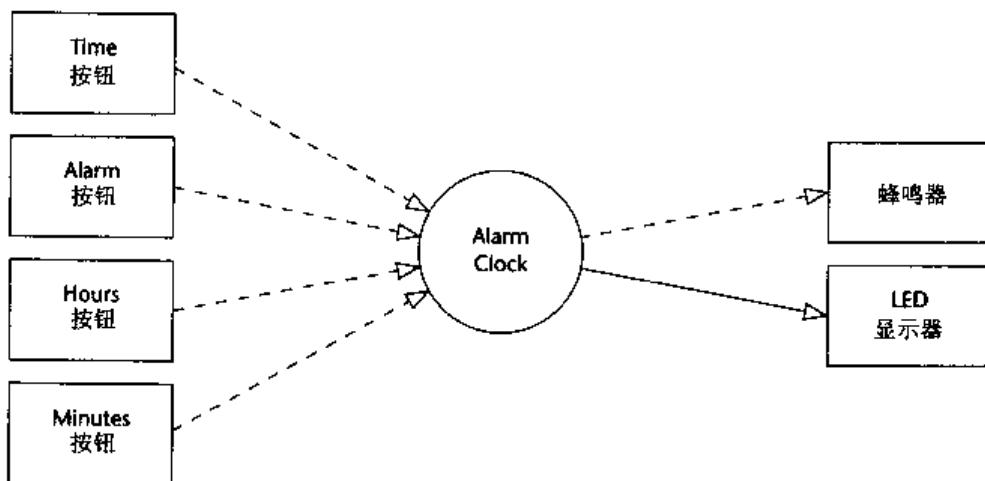


图 2.7 闹钟系统的接口关系数据流图

2.3 模式

可以通过这两个例子得出结论：对于那些具有控制系统设计经验或 LED 显示的使用经验的开发人员来说，这个任务很简单。然而，对于那些没有此类经验的人来说，即使非常小

的决策都可能有出乎意料的影响。令人遗憾的是，现有的标准设计表示法没有提供用来弥补某些设计人员经验缺乏的方法。后果不难预计，引用一个有经验的嵌入式系统开发人员的一段话即可简明地概括：“软件设计人员在每个项目中总是重新发明轮子是荒谬的”(Ganssle, 1992)。

使用“对象”、“代理”或其他新的软件标准组件或设计表示法对于解决这个问题不会有太大的帮助。相反，这里需要的是一种可以称作是“循环设计经验”的方法。具体说就是要找一个在设计过程中能够重复利用以往成功设计方案的方法。

最近，许多开发人员发现软件模式是满足这种要求的一种方法。当前有关软件模式的工作受到了 Christopher Alexander 和他的同事的工作的启发(Alexander 等, 1977; Alexander, 1979)。作为一个建筑师，Alexander 首先描述了所谓的“模式语言”，将各种各样的建筑学问题和好的设计方案联系起来。他将模式定义为“三要素规则，表示了特定的适用场合、问题和解决方案之间的关系”(Alexander, 1979, 第 247 页)。

例如，考虑图 2.8 中简要描述的 Alexander 的 WINDOW PLACE 模式。它采取了将可认识的问题与相应的解决方案联系起来的形式。更具体地说，和所有好模式一样，WINDOW PLACE 能够：

- 清楚而简明地描述针对某个定义明确的重要问题的成功解决方案。
- 描述适合于应用这种解决方案的环境。
- 提供这种解决方案的基本原理。
- 描述应用这种解决方案的后果。
- 为解决方案起一个名字。

这种描述问题-解决方案图的基本概念由 Ward Cunningham 和 Kent Beck 采用，他们使用 Alexander 的一些方法作为小型“模式语言”的基础，用来为初学 Smalltalk 的程序员提供指导(Cunningham 和 Beck, 1987)。这些工作由 Erich Gamma 和他的同事们继续进行，他们于 1995 年在一本影响广泛的书中发表了通用的面向对象的软件模式(Gamma 等, 1995)。

例如，考虑“观察者”模式(Gamma 等, 1995)，如图 2.9 所示。它描述了在有多个组件的应用中如何连接各个组件，以便当系统的一部分的状态改变时，其他相关组件将被通知到，如果需要的话则修改相关组件。这种模式成功地解决了该问题，使得系统不同的组成部分之间仍然是松散耦合，以便在以后的项目中重复使用或更加容易地修改。

WINDOW PLACE

适用场合

WINDOW PLACE 是一种建筑学的模式，常用来设计旅馆、办公室或坚固的房子。

问题

要求设计一个“起居室”，人们将坐在里面交谈、喝茶、读报纸等等。

解决方案

为了针对这个问题研制解决方案，Alexander 等人做了以下观察：

- 白天人们一般不喜欢待在没有窗户的房间。
- 当房间有窗户时，人们才愿意待在里面。因此，如果座位靠近窗口（这样人们可以从他们的座位上看见外面），那么大多数人将感觉舒适。
- 相反，如果窗口在房间的一边而座位在另一边，大多数人将感觉不舒适。

基于这些观察（原文中比这里描述的更加详细），Alexander 等人建议：为了解决这个问题，设计者应该致力于创建一个光线充足的“WINDOW PLACE”，使人们能够舒适地坐在窗口旁边。

图 2.8 WINDOW PLACE 建筑学模式的摘要（改编自 Alexander 等，1977）

观察者

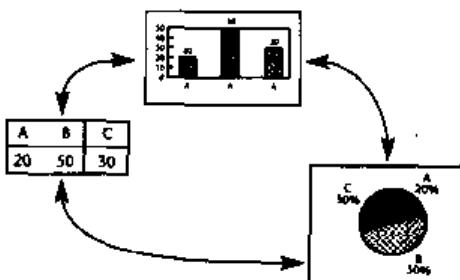
适用场合

观察者是一种软件模式，设计用在有两个或更多互连组件的应用中。Gamma 等人的描述集中在桌面系统的开发，然而该模式也可以应用在某些（相对复杂的）嵌入式系统中。

问题

要求在应用中实现组件之间一对多的相关性，这样当某个组件被修改时，其他相关组件也得到通知，如果需要的话将修改这些相关组件。

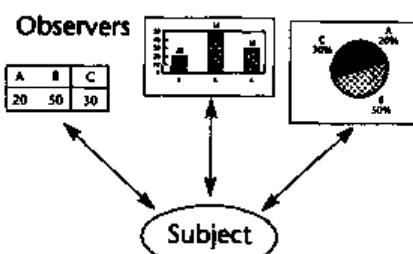
例如，假设正在研制一种电子表格应用。在这样的程序中，同样的信息可能在表格中或以柱状图形式和饼式图形式同时显示。这几种显示中的任何一个改变都必须反映在其他的显示中，如图所示：



解决方案

观察者描述如何将这样的应用分解为“对象”和“观察者”两个组件。

正如 Gamma 等人所描述的，一个对象可以有许多观察者，当对象的状态改变时所有观察者将得到通知。作为响应，观察者通常将把他们的状态和对象的状态同步，如图所示：



可应用性和结果

一种观察者可能的应用场合是当一个组件改变时需要改变其他组件，而不知道有多少个组件需要

修改。

使用这个模式的重要结果是，各种互连的组件松散地耦合在一起。这意味着，增加新的组件或删除现有的组件将几乎不影响程序的其余部分。

相关的模式和替代解决方案

观察者所描述的这种相互作用也被称为发布-预订关系。

Gamma 等人描述了两个相关的模式：MEDIATOR 和 SINGLETON。在此不对它们做详细讨论。

例子

Gamma 等人描述了使用观察者的一些应用。

图 2.9 观察者模式的概述（改编自 Gamma 等，1995）

2.4 时间触发嵌入式系统模式

Gamma 等人（1995）描述的软件模式比较实用。然而，对于时间触发的嵌入式系统的针对性还不够。因此，我们根据对（8051 及其他系列的）微控制器的应用开发经验开始整理各种模式。

这些模式的第一个版本主要用作“课堂”教学和培训，然后再更广泛地讨论并发布模式的后续版本，不只是出席关于模式的研讨会（参见 Pont 等，1999a；Pont，付印中），而是参加更广泛的技术讨论会（参见 Pont，1998；Pont 等，1999b）。通过这些讨论，我们获得了关于这个项目的许多有用的反馈，并再次完善了所收集的模式。最终的结果是贯穿于本书所详细描述的一整套模式。

最终版本的模式的结构在图 2.10 中进行了说明。模式通过章来分组，本书更进一步将其分成了许多小节。这样的安排便于查找所需要的信息。

2.5 小结

在这第二个简要的介绍性章节中，说明了嵌入式系统的开发人员能够从模式驱动的设计技术中所得到的好处，因为许多嵌入式项目要求开发人员既具备软件设计的知识，又了解一系列相关技术和工程领域。

我们用四点来总结本章：

- 不应把软件模式看作是解决嵌入式软件设计及实现问题的灵丹妙药或是如 Brooks (1986) 所谓的“银色子弹”之类的东西。软件开发是一种多层面的行为，要求智能和创造力，问题的解决方案来自聪明而有创造力的人，不要期待什么“奇迹疗法”或“银色子弹”之类的东西。
- 基于同样的理由，使用本书中的模式并不保证系统将一定可靠。例如，当你第一次坐上飞机时，飞行员放心地宣布，飞行控制软件是由使用最新的“时间触发软件模式”的工程师设计的，这可能意味着该软件将比那些没有接触过本书所提出模式的设计和编程人员设计出的软件更加可靠。然而，这绝不意味着该飞行软件没有任何缺陷。

<模式名字>

适用场合

“适用场合”概述了该模式可能的适用场合。总的说来，本书中的大多数模式的适用场合是相似的：

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。

然而，在大多数情况下，模式将有针对性。例如：

- 该应用将由电池供电。
- 正在创建应用系统的用户界面。

问题

“问题”提供该模式用来解决哪些问题的概要。例如：

- 什么时候以及如何使用石英晶体为 8051 系列微控制器建立一个振荡器？
- 怎样创建并使用合作式调度器？

背景知识

“背景知识”提供信息来帮助那些经验不足的开发人员充分利用该模式。

解决方案

解决方案这一节描述通过该模式解决问题的一个或多个解决方案。解决方案包括软件设计、源程序清单及硬件原理图。

硬件资源

每个模式都将提供或占用硬件资源。例如，微控制器模式（第 3 章）提供了 CPU 和有限的存储器资源，而外部存储器模式（第 6 章）提供了丰富的扩展存储器资源。与此对比，大多数其余模式既需要 CPU 资源又需要存储器资源。部分设计过程将涉及对硬件资源的供给和占用的平衡。“硬件资源”有助于完成这一部分的设计。

可靠性和安全性

许多模式都有潜在的可靠性和安全性问题，本节将讨论这类问题。

可移植性

本节讨论有关将模式移植到不同微控制器上的问题。

优缺点小结

本节总结了模式的优点和缺点。

相关的模式和替代解决方案

模式未必正好是你所需要的。“相关的模式和替代解决方案”讨论了替代方案，并提供了其他相关的可能使人感兴趣的模式的参考文献。

例子

每个模式的应用至少给出一个例子。

进阶阅读

“进阶阅读”给出了与使用这些模式有关的辅助信息的来源。

图 2.10 本书中模式的结构

- “模式解决方案”充其量不过是部分的解决问题。不管软件工程师或其管理者的教育背景如何，期望通过基于幻灯片模式的观察者为他们提供足够的相关领域的知识，来保证他们能够创建诸如飞行控制系统或故障诊断系统的合理设计，是不可行的。然而，我们也许能够实现的是，使软件项目经理和他们管理的团队更好地认识到是否应该在项目中任命（比方说）一个人工智能、信号处理或控制专家，完成这样的职责。
- 收集有用的模式永远不会结束，我们将逐步增加更多的模式而不断改进现有的模式。这里所收集的模式当然也不例外。正如在序言中提到的，非常感谢您对本书的评论和反馈。

1

Part

硬件基础

第 1 篇中的模式与创建基本硬件基础有关，所有基于微控制器的嵌入式系统都需要创建基本硬件基础。

首先，就微控制器自身来讨论。往往在项目初期就必须做出处理器的最初选择。这种选择将对后期的软件和硬件设计决策有相当大的影响，如果可以避免以后的改变则能够大幅降低开发费用。在第 3 章中将通过对一组模式的介绍来帮助你判断 8051 系列芯片是否适合你的系统，如果是的话，又应该使用哪一种芯片。

然后，把注意力转向振荡器电路。所有的数字计算机系统都是由某种形式的振荡器驱动的。该电路是系统的“脉搏”，是正确操作的关键。例如：如果振荡器失灵，系统将根本无法运行；如果振荡器不规律地运行，系统执行的所有有关时间的计算都将有误差。在第 4 章中讨论并比较了两种主要的振荡器电路形式。

接着讨论当加上电源时微控制器启动的特殊流程。由于系统的复杂性，必须运行一段由厂家定义的短小的“复位程序”来使硬件处于一种正确的状态，然后再开始执行用户程序。复位程序的运行需要时间，并且要求微控制器的振荡器已经工作。在第 5 章中讨论创建正确的复位电路的不同方式。

存储器是接下来讨论的重要议题。具体而言，在第 6 章中探讨了有效使用 8051 系列微控制器的内部存储器的方法，以及必要时如何为系统添加外部（程序及数据）存储器。

最后，讨论怎样设计直流负载（第 7 章）和交流负载（第 8 章）的驱动硬件。



8051 系列微控制器

引言

在大多数嵌入式项目的初期，必须做出微控制器的最初选择。使用的硬件平台将对后期的软件和硬件设计决策有相当大的影响，随着项目的推进，可能必须更换微控制器，而如果可以避免这种改变将能够大幅降低开发费用。

在本章中，将介绍三个模式来为这个选择过程提供帮助：

- 标准 8051
- 精简 8051
- 扩展 8051

注意：使用任何系列的处理器都需要对硬件、软件和人员培训做相当大的投资，几乎没有哪个公司有能力为每个项目都选择不同的器件。相反，一般都倾向于专门研究单一或很少的系列处理器。正如将在下面看到的，8051 微控制器有极多的器件可供选择，并且其数量还在日益增多。因此，对于各种各样的项目而言，8051 微控制器已成为一种极好的选择，对它的投资常常被证明是物有所值的。

标准 8051

适用场合

正在开发一个基于微控制器的嵌入式系统，要求选择使用一种硬件平台。

问题

是否将系统建立在标准 8051 系列微控制器基础之上？

背景知识

作为一个整体来看，在半导体世界中，8051 系列有一段非常漫长的历史。其体系结构的

基础起源于英特尔公司的 8048 微控制器(发布于 1976 年),第一个 8051 发布于 1980 年(Intel, 1985)。

最初的 8051 体系结构具有以下特性:

- 最高 12MHz 的工作频率。
- 32 个数字输入/输出引脚(作为四个 8 位端口)。
- 内部数据(RAM)存储器, 128 字节。
- 程序存储器有三种不同版本可供选择:
 - 无程序存储器。所有程序都需要保存在外部存储器中(8031)
 - 4K×8 位内部掩模只读存储器(8051)
 - 4K×8 位紫外线擦除的可编程只读存储器(8751)
- 两个 16 位的定时/计数器(定时器 0 和定时器 1)。
- 五个中断源(两个外部中断), 提供两个优先等级。
- 一个可编程、全双工串行端口。

图 3.1 说明了 8051 的外部接口。

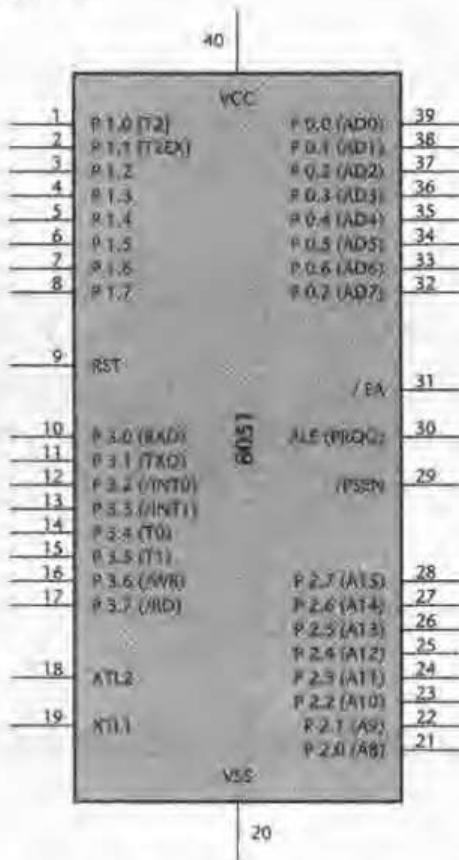


图 3.1 理论 8051 微控制器的外部接口(40 引脚封装)

注意,许多数字输入/输出引脚有复用功能。例如:在用到 UART 串行接口的系统中,将使用引脚 3.0 和 3.1(参见第 18 章)。还要注意引脚 1.0 和 1.1 的复用功能只有 8052 系列才提供。

在 8051 投放市场之后不久，8052 也随之面世（也由英特尔公司推出）。8052 在几个重要方面和之前的 8051 不同，具有以下特性：

- 内部数据（RAM）存储器增加到 256 字节。
- 根据程序存储器的不同选择，8052 有两种版本：
 - 无程序存储器：所有程序需要保存在外部存储器中（8032）
 - 8K × 8 位内部掩模 ROM（8052）
- 三个 16 位的定时/计数器（定时器 0、定时器 1 和定时器 2）。
- 六个中断源（两个外部中断），提供两个优先等级。

8052 在基本体系结构的基础上增加了一些有用的特性，尤其是具有更多的 RAM 和定时器 2。它与 8051 向上兼容。也就是说，它与 8051 的引脚和代码兼容。因此，在几乎所有的情况下，当前的标准 8051 微控制器都以 8052 系列为基准。本节将讨论“标准 8051”微控制器，它的引脚和代码与 8051 或者与 8052 微控制器兼容。

流行的 Atmel 的 89S53 是一个现代的标准 8051 的典型例子。这里简要列出 AT89S53 的主要特性：

- 完全的静态运行：0~24MHz 工作频率。
- 32 个输入/输出引脚（作为四个 8 位端口）。
- 内部数据（RAM）存储器，256 字节。
- 12KB 的“可在线编程”ROM。
- 三个 16 位的定时/计数器（定时器 2 可以递增/递减计数）。
- 九个中断源（两个外部中断），提供 2 个优先等级。
- 可编程看门狗定时器（参见第 12 章）。
- SPI 接口（参见第 24 章）。
- 低功耗空闲模式和省电模式。
- 工作电压范围为 4V~6V。

现代的标准 8051 器件（例如 AT89S53）通常为 40 脚 DIP 封装、44 脚 PLCC 封装或 44 脚 MQFP 封装。各种封装的例子如图 3.2 所示。

8051 微控制器的命名

请注意，8051 系列各种各样的器件命名给新的开发人员带来了相当大的混乱。例如：8031、8751、8052、8032、C505c、C515、C509、80C517、83C452、AD μ C812 和 80C390 全部是 8051 系列的成员。这些微控制器的名字和整个系列几乎没有联系。

最初，有一些基本约定用来识别 8051 器件的不同变体。例如：标准 8051 有掩模 ROM、无只读存储器的 8031 和有紫外线擦除只读存储器的 8751 之分。这种命名的基本约定现在已经很少遵循了。例如：就 Infineon 的 C501 来说，不同的后缀用来区分不同的 C501 版本：C501-I_R、C501-I_E 等等。

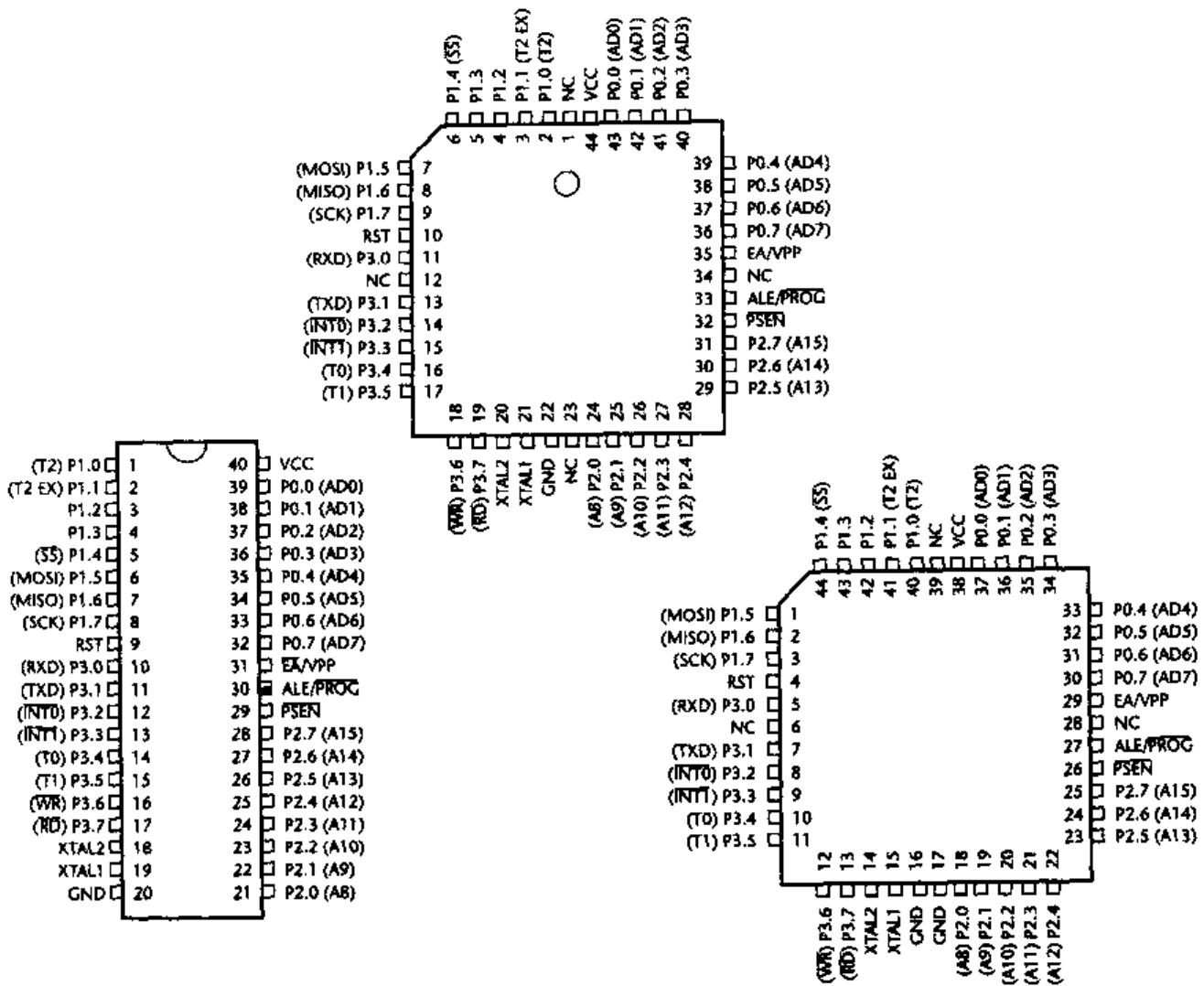


图 3.2 标准 8051 通常采用的封装的例子（这里以 ATMEL 的 AT89S53 为例）。

(左) DIP 封装。 (中) PLCC 封装。 (右) MQFP 封装 (承蒙 ATMEL 公司允许转载)

解决方案

本模式的目标是帮助决定是否在系统中使用标准 8051。要做出这样的决策，需要解决以下的问题：

1. 该微控制器的性能是否足以满足所需完成的任务的需要？
2. 该微控制器是否有足够的片内存储器来存储需要的代码和数据？如果不够，那么该微控制器是否允许使用适当的外部储存器？
3. 该微控制器是否有适当的片内模块（例如，CAN 接口、PWM 接口）来支持所需的任务？
4. 该微控制器是否有足够的端口引脚（或合适的串行接口）来满足连接外部元件（诸如

按键、小键盘、LCD显示)的所有要求?

5. 所选择的微控制器的功耗是否合适(特别是与电池供电的系统有关)?

下面将讨论每一个问题。

性能问题

当针对项目考虑选择微控制器时,要问的第一个问题是,它是否至少能满足性能上的要求。描述性能可以有各种各样的方式:一种方法是看每秒能够执行多少条机器指令,通常用MIPS(million instructions per second, 百万次指令运算每秒)来表示。

例如,英特尔公司最初的8051微控制器(和大多数现在的8051系列微控制器一样),要求最少12个振荡时钟周期来执行一条机器指令。因此,一个12MHz的8051的性能充其量不过为大约1MIPS。

提高性能的一种简单方法是提高时钟频率。现代的标准8051微控制器允许使用超过原型8051的12MHz时钟频率。例如,Infineon的C501允许时钟频率高达40MHz,这样将把该微控制器的性能提高到大约3MIPS。

提高运算性能的另一种方法是改变微控制器的内部实现,使执行每个机器指令要求的振荡时钟周期数减少。Dallas的“高速微控制器”芯片(87C520和相似产品)使用了这种方法。这样,执行一条机器指令只要求4个振荡时钟周期。这些Dallas的微控制器同样允许更高的时钟频率:一般高达33MHz。这些变化结合在一起,能够提供大约6MIPS的总体性能。Winbond的标准8051微控制器系列也做了类似的改变(参见Winbond W77E58),结果性能指标最多达到10MIPS。

显而易见,为了获得最高的性能,希望以每个振荡时钟周期执行一条机器指令的速度运行。Dallas的“超高速”89C420是实现这个目标的第一个8051器件。因此,它的运行速度是最初的8051的12倍。此外,89C420可以运行在高达50MHz的情况下,使整体性能达到40~50MIPS。

作为对比,流行的Infineon的C167系列(16位)微控制器虽然有着更现代的体系结构,但其性能指标也不过大约10MIPS。显而易见,就微控制器而言,许多8051微控制器的性能相当出色。

存储器问题

需要问的第二个问题是所考虑的微控制器是否支持系统所需要的存储器。

标准8051的存储器体系结构如图3.3所示。

首先需要注意的是,对于标准8051,直接支持最大64KB的程序长度。可以通过存储器“分段切换”来使用大型程序(这种方法将在“片外程序存储器”中讨论),然而这种方法比线性地址空间技术复杂、效率低,而且易于出错。同样,可用的数据存储器也被限制在64KB。在那些需要超过64KB的代码或数据存储器的场合,使用扩展8051往往是一种更好的选择。

需要注意的第二点是,图3.3所示的(内部和外部)存储器总数适用于所有的标准8051

器件。最好所有需要使用的存储器都尽可能在片内的器件中，因为这能够提高可靠性、降低成本、减少应用程序的长度并降低功耗。正如在“背景知识”一节中讨论的，最初的 8051 系列数据与程序代码分别有最多 128 字节的 RAM 和 4KB 的 ROM。更新的标准 8051 很少提供多于 1KB 的 RAM（通常为 256 字节），然而可能提供高达 64KB 的 flash、OTP 或掩模 ROM（详细说明参见第 6 章）。



图 3.3 8051 系列的主要存储区的原理示意图

注意，并不是所有系统都包括虚线范围内的区域。随着 8051 系列器件的发展，某些器件将提供另外的存储区。同样注意，对各种各样的片内存储器的有效使用将在“片内存储器”一节讨论。

片内可用的硬件模块

选择使用微控制器的一个主要原因是它在一个芯片里集成了系统所需的大多数（或所有）硬件功能。在这个领域，8051 系列给人的印象格外深刻。8051 系列的许多变体有效地满足了大量项目的需要，而不用借助于使用很多外部器件。

标准 8051 的一些片内可用的硬件模块如下所示：

- 所有标准 8051 都有至少一个串行端口，支持 RS-232 串行协议。这使得实现如下载数据到台式 PC 这样的功能变得很容易。我们将在第 18 章讨论嵌入式系统和桌面系统之间的连接。
- 所有的标准 8051 都有两个或三个定时器。
- 许多标准 8051 都有片内“看门狗定时器”。关于这种模块的使用将在第 12 章讨论。
- 一些标准 8051 芯片内支持 SPI 总线。我们将在第 23 章讨论这种重要的串行总线协议。
- 一些标准 8051 芯片内支持 I²C 总线。我们将在第 24 章讨论这种重要的串行总线协议。

注意，扩展 8051 器件包括很多新的特性。

虽然有这些变化，但其核心体系结构仍然不变，某个变体的软件一般能够不用做较大的改变即可在其他器件上使用。

引脚数量

所有的标准 8051 都有四个 8 位端口可用，允许扩展一些外部器件。

请注意：

- 端口0、端口2和端口3的部分引脚用来支持外部存储器（如果使用的话）。所以使用外部存储器将显著减少空闲的端口引脚的数目（详细说明参见第6章）。
- 利用新的串行总线标准，诸如I²C和SPI（参见第23章和第24章），这意味着只要共享少量端口引脚的总线，器件可以连接许多外设。

功耗

所有现代的标准8051微控制器都至少有三种运行模式：

- 正常模式
- 空闲模式
- 省电模式

“空闲”和“省电”模式设计用来在不需要进行处理时节省电源。表3.1中列出了各种模式下的典型电流要求。

表3.1 一组标准8051器件的典型的电流消耗数据

Device	Normal	Idle	Power down
Atmel 89S53	11mA	2mA	60μA
Dallas 87C520	15mA	8mA	50μA
Infineon C501	21mA	5mA	50μA
Intel 8051	160mA	—	—
Intel 80C51	16mA	4mA	50μA

注意：该数值随振荡器频率而（近似线性地）变化；这里假定所有器件的时钟频率为12MHz。

Infineon的C501是标准8051器件的一个例子，它提供和8052以及许多其他现代器件一样的省电模式。以下关于C501空闲模式的描述是根据其用户手册改编而来，详细地描述了这些模式。请注意，这些描述同样适用于大多数的标准8051。

空闲模式

在空闲模式下，C501的振荡器继续运行，而CPU将与时钟信号断开。但是，中断系统、串行端口以及所有定时器仍然连接到时钟。CPU的状态被全面保存。

这种特性所能减少的功耗取决于运行的外设数量。如果停止所有定时器和串行接口，就能最大程度地减小功耗。开发人员必须决定哪些外设必须继续运行，而哪些外设必须停止。

通过设置空闲标志位（PCon.0）可以进入空闲模式。因为PCon不是一个可位寻址寄存器，所以设置空闲位的最容易的方法是通过以下的“C”语句：

```
PCon |= 0x01; // 进入空闲模式
```

设置空闲位的指令是进入空闲模式之前执行的最后一条指令。

有两种方式可以结束空闲模式：

- 激活任意已使能的中断。这个中断将得到服务，然后程序将继续执行设置空闲位指令

之后的指令。

- 进行硬件复位。

省电模式

在省电模式下将停止片内振荡器。因此所有功能都被停止，只有片内 RAM 中的内容被保持。

省电模式通过设置 PDE (PCON.1) 标志位进入。通过如下的“C”代码可以非常容易地实现：

```
PICON |= 0x02; // 进入省电模式
```

设置 PDE 位的指令是进入省电模式之前执行的最后一条指令。退出省电模式的惟一方式是硬件复位。

硬件资源

总而言之，标准 8051 提供了以下硬件资源：

- (大约) 1MIPS~50MIPS 的 CPU 性能。
- 可用的片内存储器最多为 64KB 程序存储器和(典型地)至少 256 字节的数据存储器。
- 一个“RS-232”串行端口。
- 两个或三个硬件定时器。
- 正常运行方式下电流消耗约 20mA，空闲模式下 5mA，而省电模式下 50μA。

可靠性和安全性

没有数据可以证明标准 8051 比其他微控制器系列的可靠性更高(或更低)。尽管如此，各种各样的 8051 系列成员能够提供的功能是有区别的，而选择不合适的微控制器将对系统的安全性及可靠性造成不良影响。

区别仅仅来自于可利用的片内资源的不同。正如前面已经提到的，例如，这包括不同数量的存储器 (RAM 和 ROM)、串行接口 (SPI 和 I²C) 和模数转换器。尽可能地使用片内模块，通常将提高系统的可靠性，原因如下：

- 使用外部元件时，每一个焊点都有失效的风险，特别是在振动及湿度较大的情况下：减少焊点的数目能降低这种风险。
- 外部引线起着小型天线的作用，增加了电磁干扰 (EMI) 的易感性。因此减少外部引线将使系统更不易受到电磁干扰。
- 没有外部元件，就降低了硬件设计的复杂性，这意味着硬件设计及布线的错误机率将更少。

(在几乎所有的例子中)“片内”解决方案不但生产起来便宜而且系统硬件更简单。很显然，应该尽可能地使用所需要的所有微控制器片内资源。然而需要注意的是，使用不太通用的片内资源将使设计的可移植性不强(参见下一节)。

可移植性

由于有大量不同的 8051 系列器件可用，因此基于标准 8051 的设计本质上是可移植的。然而，如果假定使用非标准模块（额外的 RAM、额外的串行接口等等），设计将不可移植。

优缺点小结

总之，标准 8051 具有以下优点和缺点：

- ⊕ 它是一种灵活、通用的微控制器，适用于许多项目。
- ⊕ 它价格便宜。
- ⊕ 它的体系结构为许多开发人员所熟悉。
- ⊕ 许多开发工具都支持它。
- ⊕ 整个系列有超过 300 种不同的器件。
- ⊕ 由非常多的厂家提供，如果一个公司破产，将有另一个替代。
- ⊕ CPU 性能（在一些版本中）相当于或大于许多 16 位器件。
- ⊖ 它可用的存储器有限制（和 16 位或 32 位微控制器相比）。
- ⊖ 它的存储器体系结构相对复杂。

相关的模式和替代解决方案

在本节中我们将讨论标准 8051 微控制器的一些替代方案。

精简 8051 可选器件

如果系统不需要外部储存器，而且输入输出需要不超过（大约）15 个端口引脚，则精简 8051 将是一种好选择，参见精简 8051 的相关内容。

扩展 8051 可选器件

扩展 8051 通常能够做标准 8051 能做的每一件事。此外，它们通常有很多可用的端口引脚和各种各样的片内硬件模块，诸如数模转换器、CAN 接口、SPI 接口、I²C 接口等等。有时候，扩展 8051 还提供对于大容量外部存储器的支持，在一些情况下，高达 16MB。

我们将在“扩展 8051”一节中讨论这些器件。

英特尔公司的 80251 系列

英特尔公司的 MCS-251 与标准 8051 系列在软件和硬件都兼容。这意味着，在大多数情况下，可以加载现有的程序到 251 中并且将 251 系列的微控制器（40 引脚或 44 引脚）直接焊到现有的基于 8051 的电路板上。

据称通过这种方法（和最初的 1MIPS 的 8051 相比）性能可以提高 5~15 倍，并可以通过重新编译及重写代码来发挥新的体系结构的优势，从而进一步提高性能。

总之，251 的主要特性是：

- 与标准 8051 软件和硬件兼容。
- 性能是原型 8051 的 5~15 倍。
- 巨大的（高达 16MB）线性地址空间。
- 和标准 8051 相比增加了片内 RAM（高达 1KB）。
- 其他增加的组件，诸如看门狗定时器和 PWM 模块。
- 一些版本具有两个串行端口。

总体说来，251 系列的主要优势在于可以提供更加强大的功能，而且能够直接替换最初的 8051，而不用购买其他的工具（诸如编译器）。然而，251 系列没有 8051 那样流行，它所提供的功能几乎没有目前的标准和扩展 8051 器件所不能提供的。

如果希望了解更多有关 251 系列的资料，那么你可能会对 Ayala (2000) 感兴趣。

例子：使用标准 8051

贯穿于本书将给出使用标准 8051 器件的许多例子。

进阶阅读

光盘中包含了经过整理的一系列标准 8051 器件的数据手册。

精简 8051

适用场合

正在开发一个基于微控制器的嵌入式系统，要求选择使用一种硬件平台。

问题

是否将系统建立在精简 8051 系列微控制器基础之上？

背景知识

桌面微处理器市场的特点在于对更高性能的持续需要。因此，处理器在批量生产两三年后就会过时。相比而言，8051 体系结构有超过 20 年的历史，然而该系列的市场仍然不断扩展与普及。

正如在第 1 章了解到的，嵌入式市场背后的驱动力确实不同于桌面系统，只有这样才能解释上面这种现象。在嵌入式市场中，趋势是在日益广泛的系统中利用价格便宜的微控制器的灵活性。实际上，随着价格的下降，这些微控制器在那些几年前由少量分立元件（晶体管、二极管、电阻、电容）实现的系统中找到了用武之地，现在，这些系统可以利用微控制器来实现。

针对嵌入式市场的这些截然不同的特点，一些更新的 8051 器件与最初的版本相比并不更加强大或者有更多功能，反而通常具有的功能更少。

最直接的事实是，这些精简 8051 器件一般具有 20 或 24 个引脚，只有大约 15 个输入/输出引脚。除了它们的外形尺寸比较小之外，精简 8051 的另一个通用特性是它们不支持外部存储器。例如，就流行的 AT89C1051、AT89C2051 和 AT89C4051 来说，最初版本的器件上的端口 0 和端口 2^①（以及 ALE、PSEN 引脚和端口 3 上的一些引脚）被完全省略，使外部引脚的数量可以减少到 20 个引脚。Philips 的 87LPC764 和 80c751 器件也做了类似的改变（参见图 3.4）。

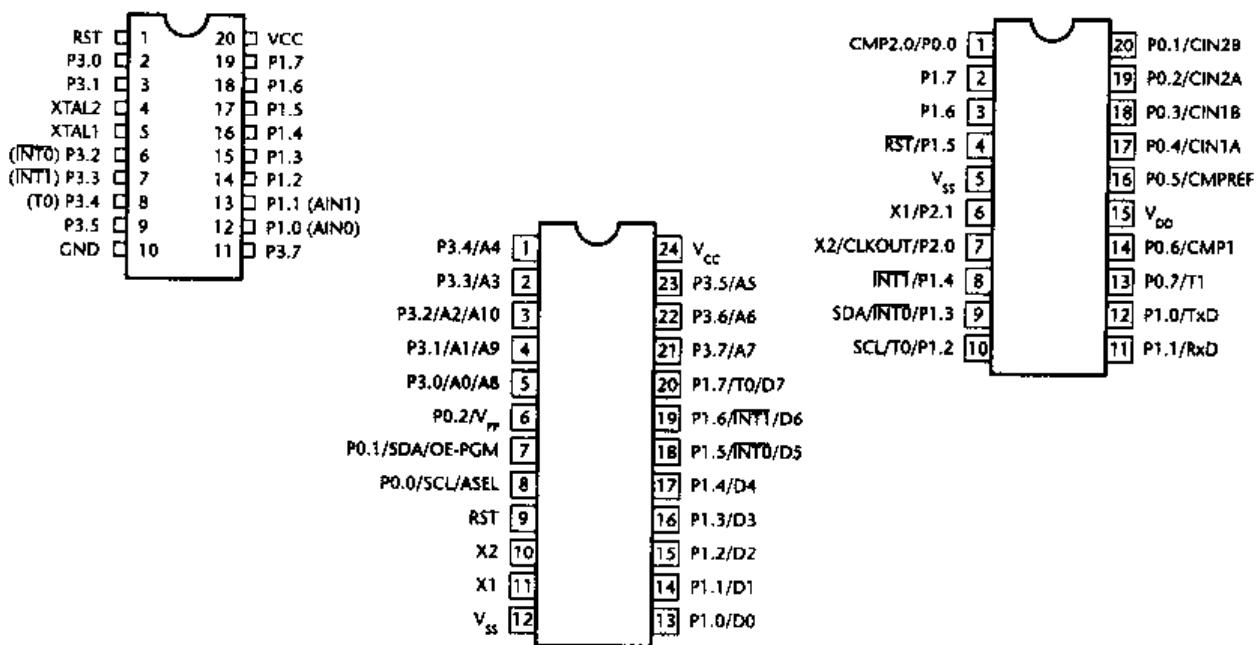


图 3.4 精简 8051 的三个例子，适用于不需要外部存储器的应用场合

[左边]Atmel 的 AT89C1051；[中间]Philips 的 80c751；[右边]Philips 的 87LPC764（以上图示由 Atmel 和 Philips 公司提供）

解决方案

是否在系统中使用精简 8051？

性能问题

大多数精简 8051 提供 1~2MIPS 的 CPU 性能（参见标准 8051 的相关内容）。

存储器问题

精简 8051 的一个主要特性是它们不支持标准的外部数据和地址总线。因此，典型的精简 8051 的存储器映像如图 3.5 所示。

片内可用的硬件模块

各种精简 8051 器件的片内硬件模块有很大的区别。流行的 Atmel 系列几乎没有片内硬件

^① 正如在“标准 8051”一节中介绍的，端口 0 和端口 2 用于扩展外部存储器时作为地址和数据总线。详细说明请参见第 6 章。

模块，而 Philips 公司的器件总是具有非常多的功能，包括模数转换器和脉宽调制器（PWM）^②模块。例如，Philips 公司的 87LPC768 是一种价格便宜、具有 20 个引脚、基于 8051 的微控制器，它包括 4KB 的 OTP 只读存储器、一个 8 位模数转换器和一个 PWM 模块。

引脚数量

显然，精简 8051 具有极少的引脚数量。



图 3.5 一种流行的精简 8051——Atmel 的 89C1051 的主要存储区的原理示意图

注意，89C2051、89C4051 和 89C1051 类似，然而具有更多的 RAM 且闪存分别有 2KB 和 4KB。

当必须扩展外部元件时，可以考虑使用标准 8051 微控制器来替换精简 8051，也可以考虑使用串行总线（例如 I²C，参见第 23 章）来连接外部设备。

功耗

标准 8051 一般具有比较小的工作电压范围：4.5~5.5V。精简 8051 器件的一个重要的特性是它们具有非常宽的工作电压范围，一般在 3~7V 左右。这种宽工作电压范围使得它非常适合于创建低成本的、由电池供电的系统。

此外，和大多数标准 8051 一样，精简 8051 具有三种运行方式：正常、空闲和省电。不同运行方式下所需要的电流典型值在表 3.2 中给出。同时需要注意，假定电源为 5V。

表 3.2 两种精简 8051 器件需要的典型电流

Device	Normal	Idle	Power down
Atmel 89C1051	9mA	1.5mA	12μA
Philips 87LPC764	8mA	4mA	1μA

注意，数据是近似的，并随振荡器频率的变化而变化。这里假定在所有情况下振荡器频率均为 12MHz。

② 我们将在第 33 章中讨论 PWM 模块的使用。

硬件资源

精简 8051 提供了以下硬件资源：

- CPU 性能在大约 1~3MIPS 之间。
- 可用的内部存储器一般最多为 4KB 程序存储器和 256B 数据存储器。不支持外部存储器。
- 通常具有一个全双工 RS-232 串行端口。
- 两个或三个硬件定时器。
- 正常运行方式下电流消耗约 10mA，空闲模式下为 5mA，而省电模式下为 10μA。
一些器件还具有其他特性。

可靠性和安全性

没有数据可以证明精简 8051 比其他微控制器系列的可靠性更高（或更低）。

可移植性

请注意，精简 8051 具有一个基于 8051 系列的核心体系结构。然而，从图 3.4 可以看出，不同的精简 8051 的引脚不兼容，在功能特性上也有极大的不同。因此，为某个精简 8051 编写的程序的可移植性比为标准 8051 写的程序要差。

优缺点小结

总之，精简 8051 具有以下优点和缺点：

- ◎ 基于 8051 核心体系结构，这样就具有许多标准 8051 的优点。
- ◎ 具有较小的外形尺寸。
- ◎ 价格便宜。
- ◎ 具有有限的片内 RAM 和 ROM 存储器，并且不支持外部存储器。
- ◎ 由于这种微控制器的系列分支的特性变化很大，所以和基于标准 8051 的设计相比，基于精简 8051 的设计不便于移植。

相关的模式和替代解决方案

精简 8051 的主要替代方案是标准 8051。

或者，也可以考虑 microchip 系列的 PIC 器件，诸如 8 引脚的 PIC12CE673。虽然这些器件基于完全不同的体系结构，但是它们的性能和精简 8051 器件相似。

例子：使用精简 8051

贯穿于本书将给出使用精简 8051 器件的许多例子。

进阶阅读

光盘中包含了经过整理的一系列精简 8051 器件的数据手册。

扩展 8051

适用场合

正在开发一个基于微控制器的嵌入式系统，要求选择使用一种硬件平台。

问题

是否将系统建立在扩展 8051 系列微控制器基础之上？

背景知识

正如我们已经了解的有关标准 8051 和精简 8051 的知识，这些经久耐用的微控制器系列能够长盛不衰，在一定程度上是因为它们提供了许多种“标准器件”（涉及传统的 8 位微控制器领域）和一系列的“精简器件”（与 4 位微控制器领域相一致，并向诸如小型 PIC 领域的器件提出挑战）。

标准 8051 和精简 8051 主要都是针对低成本、低性能的应用领域，要求有限的存储器而首要的考虑是“成本、成本和成本”。但是毫无疑问，并不是所有项目都是这样。

为了开发需要特殊硬件或大量存储器的系统，可以转为选择 16 位或 32 位微控制器平台。然而，这样的改变可能要求对人员、培训和开发工具做较大的投资。一个备选方案是使用各个厂家近几年推出的扩展 8051 器件。这样的器件保护了对 8051 系列的投资，同时，为这种微控制器系列开辟了新的应用领域。

通常，扩展 8051 提供了 8051 器件所能提供的最广阔的特性。例如，Infineon 的 C505C 和 C515C 包含了一些有用的片内硬件模块（包括对 CAN^③总线的支持），这使得这些器件被用于规模巨大的汽车市场。

C505C 和 C515C 都保留了标准 8051 对存储器的限制。相比之下，其他的扩展 8051，诸如 Dallas 的 80C390（图 3.6）和 AD 公司的 ADμC812（图 3.7）能够在线性地址空间内存取更大容量的存储器。

与许多 16 位微控制器相比，扩展 8051 通常比较便宜。然而，它们不可避免地要比标准 8051 或者精简 8051 的备选方案更昂贵。

解决方案

是否在系统中使用扩展 8051 微控制器？

③ 我们将在第 28 章中讨论 CAN 总线的使用。

- 8051 兼容的内核
 - 兼容 8051 指令集。
 - 五个 8 位输入输出端口。
 - 三个 16 位定时/计数器。
 - 256 字节中间结果暂存 RAM。
 - 每个机器周期 4 个时钟 (8051 为 12 个)。
 - DC 能够以最大 40MHz 的时钟频率运行。
 - 采用倍频器以降低电磁干扰。
 - 以 100ns 执行单循环的指令。
 - 总共 16 个中断源，包括 6 个外部中断源。
 - 两个全双工硬件串行端口。
 - SIESTA 低功耗模式。
- 存储器
 - 4KB 内部静态存储器，可用作程序/数据/堆栈存储器。
 - 外部寻址高达 4MB。
 - 默认配置为与真正的 8051 存储器模式兼容。
 - 用户使能的 22 位程序/数据计数器。
 - 16 位/22 位分页/22 位连续模式。
 - 用户可选的复用/非复用存储器接口。
 - 可选的 10 位堆栈指针提供 64 引脚 QFP、68 引脚 PLCC 和 64 引脚 QFP 封装。
- 硬件数学运算支持
 - 16/32 位数学协处理器。
- 两个全功能的 CAN 2.0B 控制器
 - 每个控制器有 15 个消息中心。
 - 标准的 11 位或扩展的 29 位标志模式。
 - 支持 DeviceNet、SDS 和高层 CAN 协议。
 - 在 autobaud 期间中止发送器。
- 可编程 IRDA 时钟
- 其他特性
 - 掉电复位。
 - 掉电预警中断。
 - 可编程看门狗定时器。
 - 振荡器失效检测。
- 封装

图 3.6 Dallas 的 80C390 微控制器的特性

- 8051 兼容的内核
 - 标称 12MHz 操作 (最大 16MHz)。
 - 三个 16 位定时/计数器。
 - 32 个可编程输入/输出引脚。
 - 端口 3 提供大电流驱动能力。
 - 九个中断源，两个优先等级。
- 电源
 - 允许 3V 和 5V 运行。
 - 正常、空闲和省电模式。
- 存储器
 - 8KB 片内 flash/电可擦除程序存储器。
 - 640 字节片内 flash/电可擦除数据存储器。
 - 片内升压电路 (不需要外部 VPP)。
 - 256 字节片内数据 RAM。
 - 16MB 外部数据地址空间。
 - 64KB 外部程序地址空间。
- 模拟输入/输出
 - 8 通道高精度的 12 位模数转换器。
 - 片内 40ppm/oC 的电压基准。
 - 200kSPS 高速采样。
 - 用于高速从 ADC 捕捉到 RAM 的 DMA 控制器。
 - 两个 12 位电压输出数模转换器。
 - 片内温度传感器功能。
- 片内“外围设备”
 - UART 串行输入/输出。
 - I²C 和 SPI 串行输入/输出。
 - 看门狗定时器。
 - 电源监控器。
- 封装
 - 52 引脚的 PQFP 封装。

图 3.7 AD 公司的 ADμC812 微控制器的特性

性能问题

扩展 8051 具有较好的性能。例如，正如在前面介绍的，Dallas 的 80C390 比最初的 8051 的性能高十倍。此外，几种扩展 8051 具有硬件数学运算单元，显著地提高了包含大量数学运算程序的运行速度。但是，Dallas 的 89C420（一种标准 8051）的功能比现有的任何一种扩展 8051 器件都更加强大。

存储器问题

一些扩展 8051 支持使用大量的外部存储器。这里详细介绍的是 Dallas 的 80C390、AD 公司的 80μ812 和 Philips 公司的 80C51MX。（详细说明参见第 6 章）

片内可用的硬件模块

一些扩展 8051 片内的可用硬件模块如下所示：

- 有几种扩展 8051 具有片内模数转换器（一般最多 8 个通道，10 位精度）。我们将在第 32 章中讨论这种转换器的用法。
- 有几种扩展 8051 具有数学运算硬件支持，能够保证（例如）浮点数学运算相对快速地运行。例如，参见 Infineon 的 C517、C537、C509 和 Dallas 的 80C390 的数据手册（包含在 CD 中）。
- 一些扩展 8051 支持 I²C 总线。我们将在第 23 章中讨论这种重要的串行总线协议。
- 一些扩展 8051 芯片内支持 SPI 总线。我们将在第 24 章中讨论这种重要的串行总线协议。
- 一些扩展 8051 支持控制器区域网（CAN）总线，我们将在第 28 章中讨论 CAN 总线。
- 一些扩展 8051 具有片内数模转换器。我们将在第 34 章中讨论这种转换器的使用。

引脚数量

许多扩展 8051 具有非常多的可用端口引脚。例如，C509 具有 9 个 8 位端口。甚至当使用外部存储器时，仍有 6 个完整的 8 位端口可用。

功耗

关于 8051 系列的三种主要的运行方式的详细说明参见“标准 8051”一节。扩展 8051 具有大量的片内硬件模块，因此，它对基本电流的要求不可避免地要比标准 8051 高。此外，如果使用外部存储器，则对电流的要求最好由试验电路实测得到。

作为一个基本的参考，具有代表性的一些扩展 8051 在不同的运行模式下对电流的典型要求在表 3.3 中给出。

硬件资源

根据不同的器件选择，扩展 8051 提供了一系列不同的硬件资源：

- 在所有情况下，内核是 8051 兼容的。

- 在大多数情况下，片内包括许多附加的外围设备。在大多数情况下，具有高性能的CPU。
- 在一些情况下，可以直接存取大容量的外部存储器。

表 3.3 一系列不同的扩展 8051 器件的典型电流消耗值

Device	Normal	Idle	Power down
Dallas 80C390	13.1mA	4.8mA	1μA
Infineon C509	31mA	19mA	30μA
Infineon C515C	24mA	14mA	50μA

注意，这些数值是近似的，并随振荡器频率的变化而变化，每种情况都假定振荡器频率为 12MHz。

可靠性和安全性

没有数据可以证明扩展 8051 比其他微控制器系列的可靠性更高（或更低）。然而，需要注意的是，许多扩展 8051 要求使用外部存储器。与仅使用内部存储器的典型系统结构相比，这有可能降低整个系统的可靠性，其原因在“标准 8051”一节中已经讨论。

可移植性

由于有非常多的不同 8051 器件可供选择，因此基于扩展 8051 的设计在本质上是可移植的。然而，正如已经讨论的，不同的扩展 8051 只在内核上通用，引脚并不兼容。这意味着内核程序（例如调度器）可以比较容易地移植，而其他模块需要根据具体的微控制器进行修改。

优缺点小结

总之，扩展 8051 有以下优点和缺点：

- ◎ 基于 8051 体系结构，这样具有许多标准 8051 的优点。
- ◎ 在大多数情况下，有大量外部端口引脚可用。
- ◎ 在大多数情况下，有一些附加的片内硬件模块可用。
- ◎ 在大多数情况下，能够存取大量的 ROM 和 RAM 存储器。
- ◎ 尽管如此，在本质上，它仍然是一种 8 位器件。对于更高的性能要求，32 位器件可能是一种较好的选择。

相关的模式和替代解决方案

在本节中我们考虑三种替代方案。

使用两个（或更多）的标准 8051

假设需要一种具有以下性能的微控制器：

- 多于 60 个端口引脚。
- 六个定时器。
- 两个 USART。
- 128KB ROM。
- 512B RAM。
- 成本大约\$2.00。

通过使用扩展 8051 可以满足这些条件中的大部分。然而，这将是所要求的\$2.00 成本的五到十倍。相比而言，图 3.8 中的“微控制器”非常符合这些要求。

图 3.8 展示了两个标准 8051 微控制器通过一个端口引脚连接在一起。正如在第 26 章的“共享时钟中断调度器（时标）”一节中介绍的，可以通过少量的软硬件开销连接两个处理器。结果将得到一个灵活的硬件平台，具有 62 个端口引脚、5 个定时器、2 个 USART 等等模块可用。注意，以后可以容易地增加更多的微控制器，通过一根引线（再加上一根地线）进行通信来保证所有的处理器上的任务完全同步。

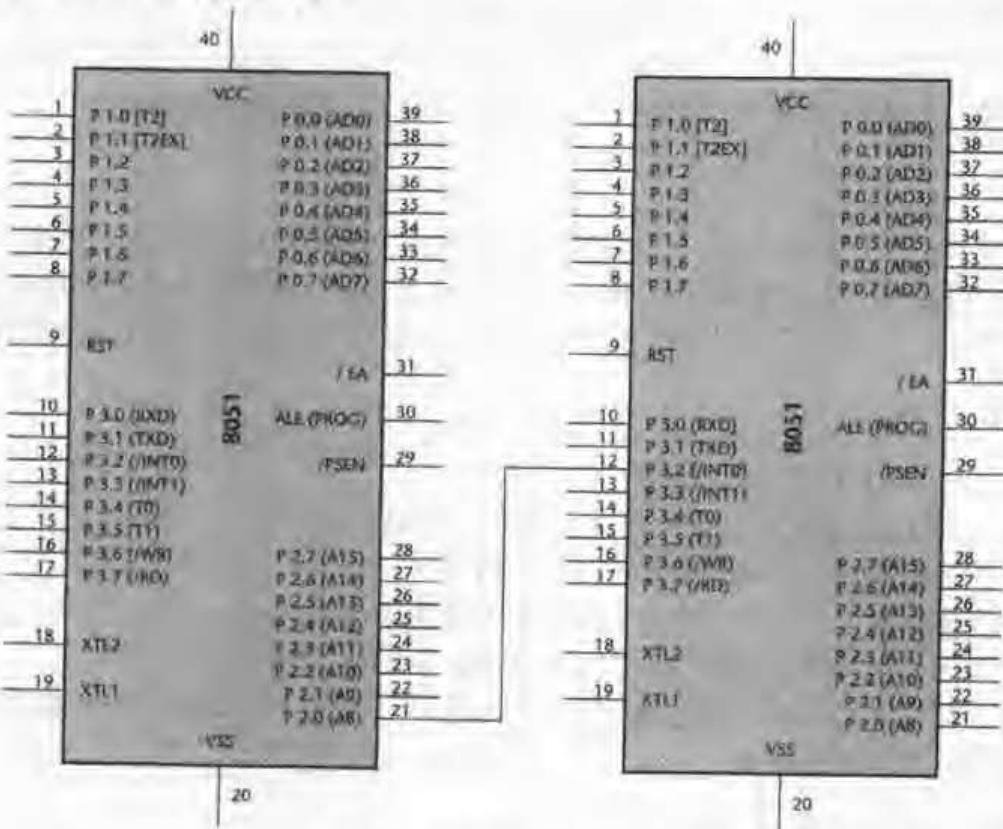


图 3.8 由两个标准 8051 创建一个扩展 8051

建立自己的 8051 器件

如果现有的所有扩展 8051 都不能满足要求，则可以考虑创建自己的 8051 器件。具体地说，

Xilinx Foundation^④提供了针对现场可编程门阵列（FPGA）及专用集成电路（ASIC）编程的一组完整的工具。可以从 Dolphin Integration^⑤购买一些与这些工具兼容的8051核。这些内核比较昂贵（大约\$16000），然而它们的效率很高（每个振荡周期执行一条指令），而且使用这样的技术可以为专用的微控制器增加硬件模块来满足特殊的要求。

创建和使用这样的8051器件超出了本书的当前版本所介绍的范围，然而相关公司的WWW站点将提供详细资料。使用这些技术需要熟悉VHDL^⑥。Yalamanchili（2001）提供了一个很好的入门起点。

值得注意的是，正是因为8051微控制器的结构非常成熟才会有厂商提供这些8051核。

使用XA系列器件

这里讨论的最后一种扩展8051的替代方案是由Philips公司生产的所谓8051XA系列。

当英特尔公司开发251系列（在“标准8051”一节中讨论）时，选择了生产一系列在很大程度上与最初的8051程序代码和硬件都兼容的器件。Philips公司在开发XA系列时，选择了一种不同的路线。其目标是开发一种新的、16位的8051器件，保持了对8051源码的兼容性，而不考虑其他方面的兼容性。

XA系列的特性包括两个16MB地址空间（程序和数据）和快速的硬件乘除功能。它还包括两个USART、一个片内模数转换器和对I²C总线的硬件支持。

值得注意的是，最近的扩展8051器件提供了相似的功能。而与扩展8051不同的是，XA系列要求开发人员购买不同的软件工具（编译程序等等）。此外，XA系列没有被特别广泛的使用，因此，开发工具和开发板不能很方便地得到。

关于XA系列的详细介绍请参阅Philips公司的WWW网站^⑦。

例子：使用扩展8051

贯穿于本书将给出使用扩展8051器件的许多例子。

进阶阅读

光盘中包含了经过整理的一系列扩展8051器件的数据手册。

④ www.xilinx.com

⑤ www.dolphin.fr

⑥ VHDL代表VHSIC硬件描述语言。缩写VHSIC代表Very High-Speed Integrated Circuit（超高速集成电路）。这些名词起源于美国国防部的开发新一代高速芯片的计划。VHDL的第一个版本于1985年发布，最近的版本是IEEE标准（1076-1993）。

⑦ www.philips.com

Chapter 4

振荡器硬件

引言

所有数字计算机系统都是由某种形式的振荡时钟电路驱动的。这种电路是系统的“脉搏”，是正确运行的关键。如果振荡器失灵，系统将完全无法运行；如果振荡器运行不规律，系统执行的所有有关时间的计算都会有误差。

因此，选择一个合适的振荡器电路是硬件设计的重要部分。对于大多数基于微控制器的系统，有两种主要的振荡器选择，分别由本章中的两个模型所描述：

- 晶体振荡器
- 陶瓷谐振器

晶体振荡器

适用场合

- 正在研制使用 8051 系列微控制器的嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

什么时候以及如何使用石英晶体为 8051 系列微控制器建立一个振荡器？

背景知识

石英是一种普通的矿物，是大多数砂粒的主要成分。它有一个有用的压电特性，这指的是如果在一块石英上加压，它将产生一定频率的电流。对于某些材料，可以倒转过来：施加电场将导致该材料的机械偏移。

可以利用这种特性作为实用的振荡器的基础，通过使用电场（在石英的表面安装触点并施加电流产生）在晶体内部产生机械振荡，随后变为晶体表面可测量的电压波动。可以通过把石

英切割为特别的大小与形状来精确控制这种波动的频率。一种被称为“AT 切割”的特殊切割方法生产起来相当廉价，并能够以合理的成本制作温度稳定性好的高频率晶体。

为了建立一个完整的振荡器，需要更多的组件。图 4.1 展示了怎样用晶体来形成一种被称为皮尔斯振荡器通用振荡器电路。

皮尔斯振荡器的变型在 8051 系列通用。建立这样一个振荡器的大多数组件已经包含在微控制器内部，有时这些组件一起被称为振荡反相器。使用这种器件一般只需提供晶体和两个小电容来实现完整的振荡器。将在这个模型的“解决方案”一节中做更进一步的讨论。

注意，在一些情况下，使用一个完整的、独立的外部晶体振荡器模块（基于类似图 4.1 说明的电路）来驱动微控制器更好。将在“可靠性和安全性问题”一节中讨论这种设计。

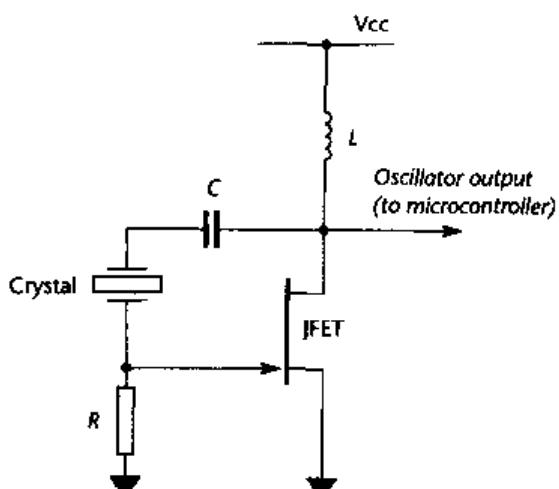


图 4.1 由石英晶体驱动的皮尔斯振荡电路（由 Horowitz 和 Hill, 1989 改编）

振荡器频率和机器周期之间的联系

为 8051 系列器件选择合适的振荡器时，振荡器频率的选择是真正所关心的：机器周期。也就是说，我们关心指令的执行速度。

正如在第 3 章讨论的，8051 系列的各种微控制器的机器周期和振荡器周期之间有不同的关系。例如：最初的 8051 系列微控制器的机器周期使用了 12 个振荡器周期。后来的芯片，诸如 Infineon 的 C515C，一个机器周期使用 6 个振荡器周期；更新的器件诸如 Dallas 的 89C420，每个机器周期只需要一个振荡器周期。因此，运行在同样的时钟频率下，后来的 8051 系列微控制器执行指令迅速且多。

通常，现代 8051 的性能提高是一件好事。然而，在计时起关键作用的场合，必须小心保证在每个特定的器件上，所有与时间有关的计算都被正确地实现。详细说明参见硬件延迟和合作式调度器。

为什么应该使时钟频率尽可能地低

按一般规律，系统的运行速度直接由振荡器频率决定。在大多数情况下，如果振荡器频率

加倍，系统将以两倍的速度运行。

根据我们的经验，许多开发人员选择振荡器/谐振器的频率为接近或达到该器件支持的最大值。例如，Infineon 的 C505/505C 可以运行在 2~20MHz 的晶振频率，而许多人自然而然地选择接近或达到这个范围的最高值，以获得最高性能。

这是不正确的，原因如下：

- 许多系统不需要现代 8051 器件提供的那么高的性能。
- 在大多数现代（基于 CMOS）的 8051 中，在振荡器频率和电源电流之间有着几乎线性的关系。因此，通过使用尽可能低的频率可以减少功耗要求。这在许多系统中都是非常有用的。
- 如果芯片以较低的速度运行，存取低速外设（诸如低速存储器或 LCD 显示）的编程和硬件设计将极大地简化，而外部器件（诸如存储器的锁存器）的成本也将降低。
- 电路产生的电磁干扰（EMI）随着时钟频率的增加而增加。

总之，在满足系统所需的性能要求的前提下，应使用尽可能低的振荡器频率。

0MHz 运行频率？

有几种现代的 8051 系列芯片能够运行在低至 0Hz 频率。例如：Atmel 的 89C52 器件的工作范围从 0~24MHz。通过使系统运行在极低频率下（运行在 kHz 甚至 Hz，而不是 MHz），可以节约大量的电能。将在一年调度器中使用这种特性。

在一些系统中，甚至 0Hz 也是有用的。初看起来这可能毫无意义：在 0Hz 时，器件将不再运行和执行代码。然而，即使时钟频率降低，设计为可以工作在低频下的器件的系统状态也将被保持。这意味着时钟频率可以降为 0 来节约电能。此外，这意味着如果时钟暂时失效（无论何种原因）然后恢复，系统也将随之恢复。

解决方案

这个模型的目标是帮助决定是否应该使用石英晶体为 8051 微控制器提供时钟，如果使用，如何连接这样的器件。本节直接回答了这个问题。

稳定性问题

为系统选择振荡器的一个主要因素是振荡器的稳定性问题。在大多数情况下，振荡器的稳定性是用诸如“±20ppm”这样的数值来表示的，即百万分之二十。

通过具体例子来说明这个问题，考虑一年中有大约 3200 万秒。^①每隔 100 万秒，晶体将可能超过（或滞后）20s。因此，一个基于 20ppm 晶体的时钟将在一年中超过（或滞后）大约 $32 \times 20s$ ，即大约 10 分钟。

^① $(365 \text{ 天}) \times (24 \text{ 小时}) \times (60 \text{ 分钟}) \times (60 \text{ 秒}) = 31\,536\,000 \text{ 秒}$ 。

标准的石英晶体一般标称为 $\pm 10\sim \pm 100\text{ppm}$ ，因此每年将可能超过（或滞后）大约 5~50 分钟。注意这些数值也适用于外部的振荡器模块。如果要求更高的准确度，参见“相关模型”。

成本问题

晶体的价格大约是陶瓷谐振器的两倍，并与晶体的稳定性有关。

如何将晶体与微控制器连接在一起

晶体振荡器的基本连接在图 4.2 中给出。

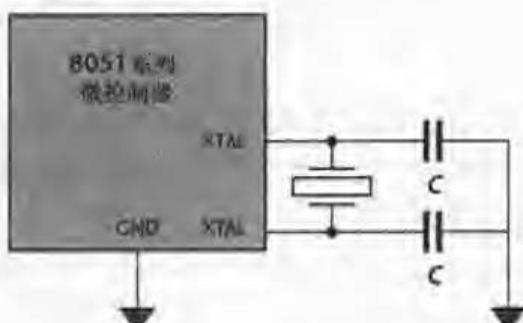


图 4.2 一种简单的晶体振荡器电路

电容值将随微控制器和晶体频率而变化。将在后面的例子中提供一系列不同的 8051 器件的电容推荐值。更加详尽的资料请参阅所选择的微控制器的数据手册。在没有特定信息的情况下，30pF 的电容值将在大多数情况下正常工作。

硬件资源

在大多数情况下，晶体振荡器的使用和系统中的 CPU 或内存需求没有直接的关系。然而，如果希望进行温度测量来增加振荡器的稳定性（参见“可靠性和安全性问题”），将占用 CPU 和存储器开销。

同时注意，正如在背景知识一节中讨论的，系统的性能直接与晶体频率有关系。如果系统不能足够快地工作，可以考虑提高振荡器频率。也可以考虑使用更现代的 8051 设计，例如，Dallas 或 infineon 的那些采用更少周期数来执行每条指令的微控制器。

可靠性和安全性

在本节中讨论一些与晶体振荡器相关的可靠性和安全性问题。

系统脉搏

振荡器形成了所有数字计算机中的“脉搏”。如果这些脉搏停止，系统就会停止。如果这些脉搏有变化，定时回路、延迟、形成的波形等等也会有变化。因此，嵌入式系统的正常运行依赖于提供一个不易受干扰而又有规律的时钟输入。

玻璃的主要成分

石英在一些物理性质上和玻璃相似，尤其是它易碎。

如果要求振荡器工作在那些有很大振动的环境下，那么石英未必是理想的选择。如果在这些环境中使用石英晶体，就需要将系统封装起来以避免振动影响系统的运行。

振荡器的起振时间

如果电路中的（晶体）振荡器的起振有延迟，那么复位周期可能会在振荡开始之前结束。如果发生这种情况，芯片将不会复位。^②晶体振荡器的起振时间取决于它是否被正确地安装并配有合适的电容。典型的起振时间是 0.1~10ms (Mariutti,1999)。

使用外部晶体振荡器模块

正如在“背景知识”中介绍的，可以使用独立的外部晶体振荡器模块（基于图 4.1 所示的电路）来驱动微控制器。这种技术的主要优点是振荡器能够保证起振。如果系统必须非常可靠地运行，那么这将是一种比较好的解决方案。

连接振荡器模块非常简单。图 4.3 展示了一个可以用在所有 8051 系列成员上的电路。注意，正如该图所示，引脚 XTAL1 应该被驱动，而 XTAL2 悬空不接。

特别是当使用较高的时钟频率 (>12MHz) 时，使用振荡器模块将提高系统的可靠性。然而，振荡器模块也的确有几个缺点：

- 振荡器模块的价格大约是晶体振荡器的两倍，大约是陶瓷谐振器的四倍。
- 振荡器模块对电流的要求一般与一个 8051 微控制器相当：15~35mA。在电池供电的系统中，这意味着非常大的功耗。
- 可能不容易找到不常用频率的振荡器模块，比如，11.059MHz。正如在“相关的模式和替代解决方案”中讨论的，这种频率对涉及串行接口的基于 8051 的设计非常有用。

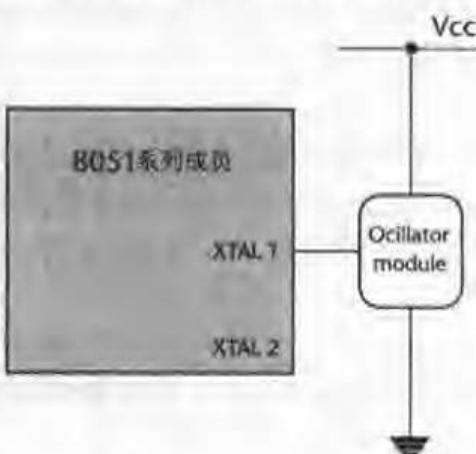


图 4.3 使用外部晶体振荡器模块

^② 详细说明参见阻容复位。



提高晶体振荡器的稳定性

正如已经讨论的，典型的晶体振荡器的稳定性在 $\pm 20\sim 100\text{ppm}$ 之间。如果使用这样的振荡器控制实时时钟，可能超过或滞后高达50分钟/年，即高达1分钟/星期。这种结果不只是针对8051系列，甚至在用于台式计算机网络的昂贵服务器中，这种现象也很明显。相比而言，大多数使用晶体振荡器的“石英”手表成本极低而时间很准。这是因为它们已经配置了很复杂的温度控制系统，以保持每天有大约16个小时运行在35°C的温度下。这个温度控制系统就是你的手腕（依靠生物学的机制）。

如果需要普通的由晶体控制的嵌入式系统的时间也很准确，可以把该设备放到固定温度的烘箱（或冰箱）中并调整软件来使时间准确。然而，这很不实际。相反，可以得到具有易于使用的封装的“温度补偿晶体振荡器”（TCXO）。它提供了晶体振荡器和补偿温度变化的电路。这样的器件提供高达 $\pm 0.1\text{ppm}$ （或更高）的稳定性。在时钟电路中，将超过或滞后至多大约1分钟/20年。除了那些要求最苛刻的系统，这种级别的准确度适用于几乎所有系统。然而，有一个不利因素，温度补偿晶体振荡器的单价可能超过\$100.00甚至好几倍。这样的价格使它们难以应用到大多数嵌入式项目中。

一种实用的备选方案是确定所选定晶体的温度-频率特性并将这个信息包含在系统中。一个简单的温度传感器的价格大约为\$2.00，以此为代价，能够跟踪温度并按要求调整时间。这是稳定的调度器的基础。

另一个备选方案是使用原子钟。例如，铯钟采用原子跃迁作为晶体振荡器的基准，能够提供百万万分之几的准确度（即，大约 $1/10^{12}$ ）。这相当于大约1分钟/百万年的准确度。这种设备的使用可能听起来像是一种奇异而昂贵的解决方案，然而现在可以通过使用全球定位系统（GPS）接收器或全球定位系统芯片组来访问位于不同卫星上的原子钟。这种技术越来越多地用于移动式电话公司（蜂窝电话）的基站上。

可移植性

这种技术能够并且已经在各种各样的微控制器和微处理器上使用。

注意，正如在“解决方案”一节中讨论的，所使用电容的容量取决于晶体频率和所使用的微控制器的要求。微控制器厂家的数据手册将提供推荐值。

优缺点小结

- ⊕ 晶体振荡器是稳定的。一般 $\pm 20\sim 100\text{ppm}=\pm 50\text{分钟/年}$ （高达1分钟/星期）。
- ⊕ 大多数基于8051的设计使用这里给出的简单晶体振荡器电路。因此，开发人员熟悉基于晶体振荡器的设计。
- ⊕ 大多数常用频率的石英晶体价格合理。要求的其他元件通常仅仅是两个小电容。总体来说，晶体振荡器比陶瓷谐振器更昂贵。
- ⊖ 晶体振荡器对振动敏感。

- ⑧ 稳定性随使用时间下降。

相关的模式和替代解决方案

一个替代方案

外部晶体振荡器的主要替代方案是外部陶瓷谐振器，参见陶瓷谐振器。

使用片内振荡器

正如在第3章看到的，精简8051器件，诸如流行的Atmel的89C4051，设计为取代（由晶体管、电阻、电容等等装配的）分立电路灵活而高性价比的方案。然而，这种器件仍然要求外部振荡器和复位电路。由于振荡器和复位元件加在一起的价格和这种精简的微控制器一样多，并且极大地增加了电路板的尺寸，因此将振荡器和复位电路包含在微控制器内部应该是很有意义的。

现在，这种8051系列的器件已成为现实，诸如Philips的87LPC764。这种20引脚的器件包含一个片内复位电路（参见第5章），和一个片内阻容（RC）振荡器。因此能够在没有任何外部元件的情况下使用。

阻容振荡器（又名“驰豫振荡器”）越来越多地成为可用的片内模块。这种振荡器的简单实现在图4.4中进行了说明。

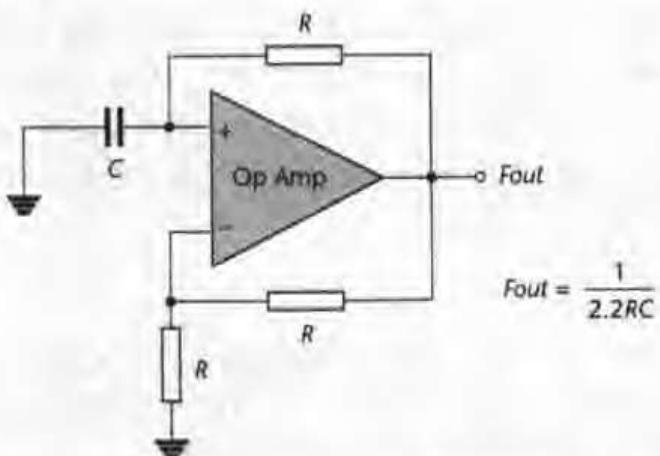


图4.4 一个简单的基于运算放大器的阻容振荡器（由Warne, 1998改编）

注意，其他实现方式可能使用少数几个逻辑门。

这种简单振荡器的其他实现方式可能使用少数几个逻辑门。然而，无论怎么实现，RC解决方案的问题都是这种振荡器不能很稳定地工作，这主要是由于电阻的阻值将随温度的变化而变化。这明显表现在许多实际的实现中。例如，87LPC764（以及类似器件）上的阻容振荡器的稳定性只有±25%。这对于许多系统是不够的。例如，如果用来产生串行接口的波特率，这种级别的稳定性将意味着通信不可能成功。

计时问题

不要仅仅因为微控制器可以运行在很大的频率范围就认为能够自由选择这个范围里的任何频率。振荡器频率的选择将对系统中所有与时间相关的方面产生较大的影响。

例如，无数基于 8051 的系统设计使用 11.0592MHz 的晶体频率。使用这个频率的理由是：在标准 8051 器件中使用这种晶体频率，可以由内嵌的串行端口很容易地产生标准波特率（诸如 9600 波特率）。使用其他频率（例如 10MHz、12MHz）的振荡器则难以产生这种标准波特率。在第 18 章中将更深入地讨论这个问题。

同样，振荡器频率规定了系统中的硬件定时器的增加频率。例如，如果需要调度一个任务以精确的每分钟运行，若选择了不恰当的振荡器频率，则将很难实现（详细说明参见第 14 章）。

注意，这种特殊的数字不仅仅局限于 8051 系列。“石英”数字手表使用 32.768kHz 的频率，因为通过 2^{15} 分频，将得到 1Hz “时标” ($2^{15}=32\,768$)。

例子：将晶体振荡器连接到 Atmel 的 89C2051

Atmel 的 89C2051 连接大多数石英晶体时的推荐电容值如图 4.5 所示。

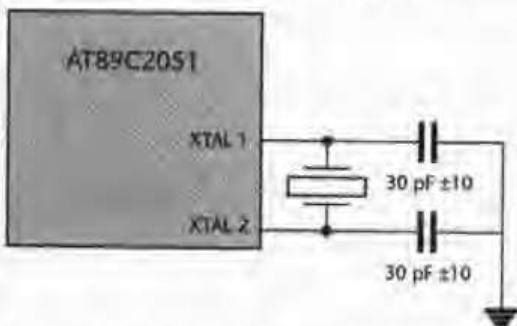


图 4.5 将晶体振荡器连接到 AT89C2051

例子：将晶体连接到双处理器电路板

请注意，大多数晶体或振荡器模块将驱动多于一个（一般高达五个）微控制器。数据手册将规定这种“扇出”值。这在多处理器设计时是很有用的，因为这意味着两个微控制器（假定都是 8051）将始终“同步”（详细说明参见第 6 篇）。

图 4.6 说明如何将两个芯片连接到同一个晶体。注意同样的方法能够适用于所有的微控制器组合。如果这两个芯片要求不同的电容值，那么选择一个中间值。

进阶阅读

参考所选择的微控制器厂家的数据手册来保证在晶体振荡器电路中使用要求的电容值。

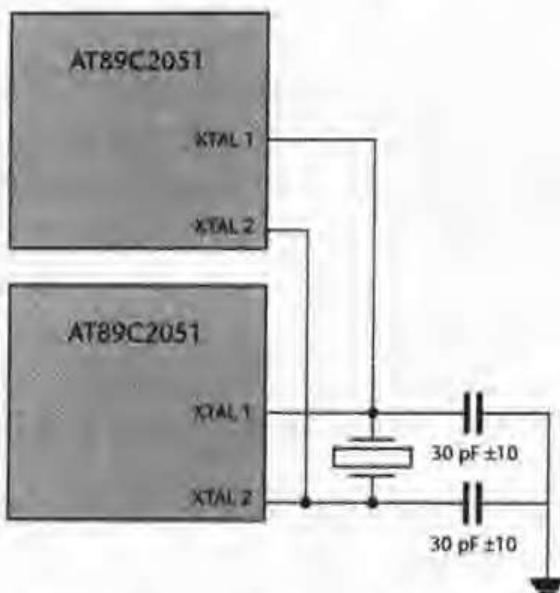


图 4.6 一个晶体一般能够驱动几个微控制器

陶瓷谐振器

适用场合

- 正在研制使用 8051 系列微控制器的嵌入式系统。
- 正在为系统设计一个合适的硬件基础。

问题

什么情况下以及如何使用陶瓷谐振器为 8051 系列微控制器提供时钟信号？

背景知识

陶瓷谐振器和石英晶体振荡器一样基于压电材料。这种材料（正如其名称所暗示的）是一种压电陶瓷。更多的背景材料参见晶体振荡器。

解决方案

这个模型的目标是帮助决定是否应该使用陶瓷谐振器为 8051 微控制器提供时钟，如果使用，则应如何连接陶瓷谐振器。本节直接回答了这个问题。

稳定性问题

正如在晶体振荡器中讨论的，为系统选择振荡器的关键因素在于振荡器的稳定性。晶体振荡器的稳定性通常用百万分之几的方法表示。而陶瓷谐振器与此不同，其稳定性通常按百分比

表示。1%的稳定性数值是普通的。一天有 1 440 分钟，这样，基于 1% 陶瓷谐振器的时钟将超过（或滞后）大约 14 分钟/天。显而易见，这种器件不适合用于要求长期精确计时的系统。然而，如果需要采用这种谐振器来作为 30s 延迟的基准，那么可能的超过或滞后是 0.3s，但这不一定成为一个问题。

成本问题

陶瓷谐振器的价格是晶体振荡器的一半。

外部电容器

大多数陶瓷谐振器包含内部电容器。因此它们可以不需要外部电容器而直接连接微控制器。这使得它们更易于使用且更进一步降低了成本和对电路板尺寸的要求。

硬件资源

在大多数情况下，陶瓷谐振器的使用和系统中的 CPU 或内存需求没有直接的关系。此外需要注意的是，系统的性能与谐振器频率直接相关。如果系统不能足够快地工作，则可以考虑提高频率。也可以考虑使用一种更现代的 8051 设计，例如 Dallas 或 infineon 的那些需要更少周期数来执行每条指令的器件。

可靠性和安全性

关于振荡器的可靠性和安全性问题的一般讨论参见晶体振荡器。总体说来，陶瓷谐振器是所讨论的在物理上最不易受干扰的振荡器类型。

可移植性

陶瓷谐振器能够并且已经在各种各样的微控制器和微处理器上使用。

请注意陶瓷谐振器通常不应被用来直接替换晶体振荡器。这两种解决方案要求不同的电容（如果有的话）。

优缺点小结

- ☺ 比晶体振荡器更便宜。
- ☺ 物理上不易受干扰：比晶体振荡器更不易受物理振动（或设备坠落等等）的破坏。
- ☺ 许多谐振器包含内置电容器，能够在没有任何外部元件的情况下使用。
- ☺ 尺寸小。大约是晶体振荡器尺寸的一半。
- ☹ 相对比较低的稳定性。一般不适用于需要（在长时间里）精确计时的系统。一般 $\pm 5000\text{ppm} = \pm 2500 \text{分钟}/\text{年}$ （高达 50 分钟/星期）。

相关的模式和替代解决方案

晶体振荡器是主要的替代方案。

例子：将陶瓷谐振器连接到 8051 微控制器器

对于许多不需要精确计时的简单的消费类应用，价格是关键，这类系统可以使用陶瓷谐振器。在大多数情况下，使用的是有内部电容器的陶瓷谐振器。同样的谐振器能应用于所有 8051 系列微控制器（图 4.7）。

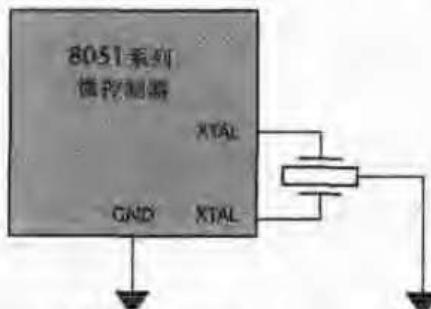


图 4.7 一个简单的陶瓷谐振器电路，该谐振器含内部电容

注意，当没有内置电容器时，应使用“石英晶体”电路（图 4.2）。推荐的电容值参考微控制器数据手册。不同类型的谐振器很好区分：那些没有电容的谐振器有两个引脚（像晶体）；而那些有电容的谐振器具有第三个引脚。当有第三个引脚时，中间的引脚应该接地。

进阶阅读

Chapter 5

硬件复位

引言

所有微控制器的启动流程都不通用。由于硬件的复杂性，必须运行一段由厂家定义的短小的“复位程序”来使硬件处于一种正确的状态，然后再开始执行用户程序。运行这个复位程序需要时间并且要求微控制器的振荡器已经运行。

当系统由可靠的电源供电时，一旦通电，电源迅速地到达额定输出电压；一旦关电，电源迅速地下降到 0V，并且在接通的时候电压不会降低。这时能够可靠地使用基于一个电容和一个电阻的低成本复位硬件。这种形式的复位电路称为阻容复位。

如果电源不够可靠，而系统涉及安全性，这种简单的阻容解决方案就不合适了。

可靠的复位讨论了一种更为可靠的替代方案。

阻容复位

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

怎样为 8051 微控制器创建一个低成本的复位电路？

背景知识

正如在本章的引言中介绍的，要求复位流程在执行其他所有代码之前结束，而运行复位流程又要求微控制器的振荡器正在运行。早期的 8051 系列芯片具有一个“复位”引脚用于触发复位操作。当引脚维持为逻辑 0 时，芯片将正常运行。当振荡器正在运行的时候，如果这个引脚被置为逻辑 1 并保持两个（或更多）机器周期，则该微控制器将被复位。

注意，如果复位操作没有正确结束，微控制器通常将根本不运行。它在极少情况下会运行，然而其运行不正确。无论发生这两种情况中的哪一种，通常在软件上都无法恢复对系统的控制。因此，显而易见，保证正确的复位操作是所有应用程序的关键。

解决方案

有多种技术可以用于 8051 系统上电时的自动复位流程。使用最广泛的是使用外部电容和电阻。这里将详细讨论这种技术。

阻容复位电路

一个典型的阻容复位电路如图 5.1 所示。

图 5.1 中的电路运行如下。假定 V_{cc} 最初为 0V（即系统没有上电）而且电容 C 完全放电。当上电时，电容开始充电。最初，电容两端的电压为 0V，因此电阻两端的电压（以及复位引脚上的电压）为 V_{cc} ，这是逻辑 1。逐渐地，电容将充电而电压将上升，最后为 V_{cc} 。此时，复位引脚上的电压将为 0V。

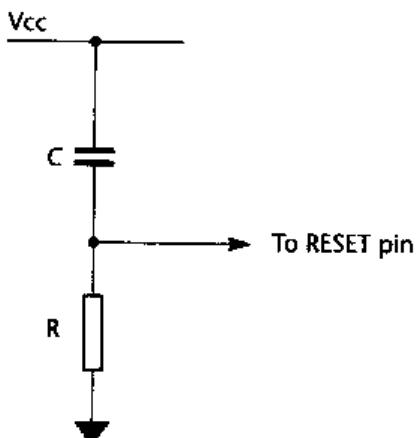


图 5.1 一个（高有效的）阻容复位电路

在实际系统中，微控制器输入电压的阈值大约为 1.1~1.3V^①。低于这个阈值的输入电压被认为是逻辑 0，而超过这个阈值的电压被认为是逻辑 1。这样，复位操作将延续直到复位引脚上的电压降到大约 1.2V。

根据这些知识能够计算出 R 和 C 的要求值。图 5.1 中的电容在开始充电后将在时间 t 秒内达到电压 V_{cap} ，它们之间的关系由等式 5.1 给出，并将以这个等式为基础计算。

$$V_{cap} = V_{cc}(1 - e^{-t/RC})$$

等式 5.1 图 5.1 中电容两端的电压随时间变化的函数

^① 如果需要的话，所选择的微控制器的数据手册将提供精确的值。

注意：等式 5.1 中假定电容由 0V 开始充电，而且电源电压瞬间从 0V 阶跃到 V_{cc} （而不是缓慢地斜线上升）。这些假定通常是不正确的。有关这个问题的讨论参见“可靠性和安全性”。

当使用这种形式的复位电路时，英特尔公司的 8051 数据手册推荐 8.2K 的电阻值和 $10\mu F$ 的电容值。将这些值代入等式 5.1，并将 500ms 内的结果绘图最终得到图 5.2。

如图 5.2 所示，所有的 8051 都在 24 个或更少的振荡器周期内完成复位操作。如果使用 12MHz 的振荡器，这段时间最大为 0.002ms。相比而言，推荐的复位电路需要大约 100ms 来完成复位操作。这个复位周期看起来像是过长，然而，由于在“可靠性和安全性”中讨论的理由，通常在实际系统中为复位提供大约 100ms 是很有必要的。

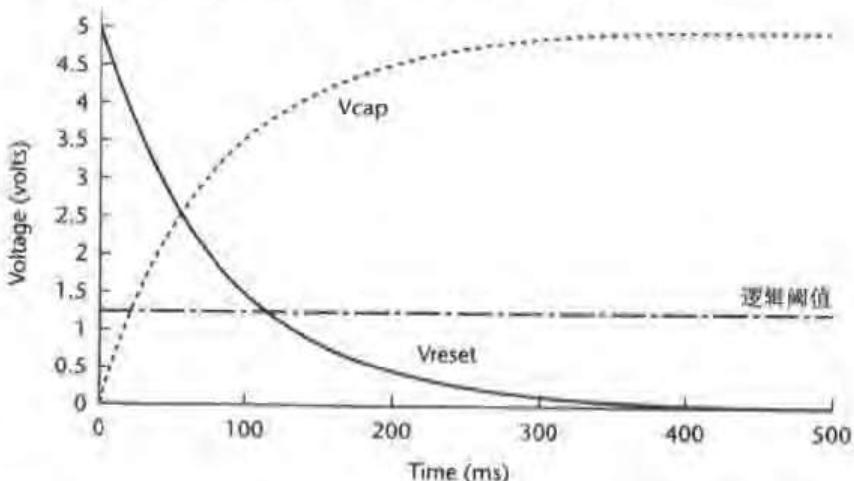


图 5.2 使用标准元件值和理想电源的阻容复位电路的特性的一个例子

选择 R 和 C 的值

如果在审阅了这个模式的所有方面之后，决定使用基于阻容的复位电路，那么应该采用多大的 R 和 C 的值呢？

与其试图由等式 5.1 直接确定 R 和 C 的值，倒不如将其简化。注意，R（欧姆）乘以 C（法拉）被称为这种形式的阻容电路的“时间常数”(s)。这个时间常数是电容充电到最终电压的 60% 时所需的时间。这样，图 5.1 中的电路在 5V 电源下时，电容电压达到 3V 时所花的时间即时间常数。因此，复位引脚上的电压达到 2V（即 $V_{cc}-3V$ ）。这对于保证芯片处于复位模式仍然是足够高的（正如已经讨论的，因为它大于 1.2V）。只要芯片保持在这种模式直到电源达到 V_{cc} （一般在上电后 100ms 左右，参见“可靠性和安全性”）后大约 1ms，该芯片被正确复位。

因此，一个基本的经验法则是，阻容时间常数应该为大约 100ms，满足这个条件的 R 和 C 的值通常将保证有效的复位操作（等式 5.2）：

$$RC \geq 100\text{ms}$$

等式 5.2 计算合适阻容值的一个经验法则

一个满足这个条件的合适的阻容复位电路如图 5.3 所示。

本节的主要内容概述如下：

- 在阻容复位电路中，10K 电阻和 10μF 电容组合得到 100ms 的时间常数。记住阻容复位电路的一般限制（参见“可靠性和安全性”），这个值适合于大多数基于 8051 的系统。
- 标准的 8K2，10μF 阻容复位组合得到 82ms 的时间常数。这通常是足够的。
- （在一些书上出现的）1K 和 10μF 值只提供 10ms 的时间常数。这些值将不能在所有电源的情况下提供可靠的复位操作。

增加一个复位按钮

在一些系统中，增加一个复位按钮有助于进行人工硬件复位。这很容易实现。图 5.4 展示了一个合适的电路。

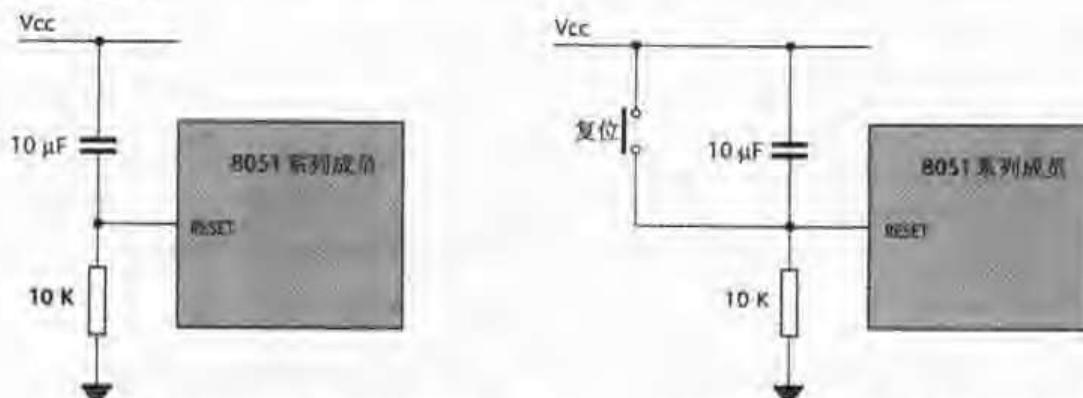


图 5.3 一个合适的（高有效的）阻容复位电路 图 5.4 一个具有复位按钮的（高有效的）复位电路

注意，复位按钮将复位引脚拉到 Vcc（假定为高有效的，参见“可移植性”）。此外，注意这个按钮同时对电容放电，保证当按钮释放时将执行正确的复位流程。

硬件资源

这个模式不涉及 CPU 或存储器的使用。

可靠性和安全性

使用阻容复位电路有许多可靠性和安全性的问题。主要问题将在本节讨论。

然而，总的说来，做出如下面框内所示的建议。

许多嵌入式系统的可靠性方面的问题可以追踪到复位电路的故障。如果成本是惟一的考虑因素，可以选择阻容复位；如果可靠性是考虑因素之一，则应选用可靠的复位。

电源达到稳态所需的时间

假设正在开发一个嵌入式工业控制系统，想保证在上电之后尽可能快地开始运行。在“解决方案”中注意到使用 12MHz 振荡器的 8051 微控制器的复位流程将需要 0.002ms。这样得出结论，提供 1ms 的复位周期（而不是前面推荐的 100ms）将提供足够的裕度。

假设调整阻容值使复位周期减少到大约 1ms。例如，图 5.5 展出了使用 $0.1\mu F$ 电容和 $6K7$ 电阻的结果。

这个组合值有时能工作。然而在大多数系统中都将出错。原因是实际的电源无法立即从 0V 切换到它们的额定输出电压。实际上，当第一次接通时，许多电源需要 50ms 或 100ms 才能达到这个电压。必须在设计中考虑到这种“斜线上升”的电压输入。

如果电源电压慢慢地增加，那么阻容复位电路中的电容将相对快速地充足电并简单地“跟随”不断增加的供电电压。结果， V_{reset} 在芯片达到它的工作电压（5V）之前将有许多 ms 维持在逻辑 0。因此，芯片只有在复位信号完成之后才准备运行它的复位程序，而复位将无法执行。系统将因此无法正确启动。

如果确实需要迅速的复位，你又能控制电源的设计，则有多种方式可以解决这个问题。例如，可以增加变压器容量或降低滤波电容器的值。然而需要注意的是，将设备限制为运行在特定电源上将导致该系统设计的可移植性大大降低。例如，如果贵公司后来决定把电源转包给其他公司或在一系列不同芯片上使用同一个电源，则将很快陷入困境。

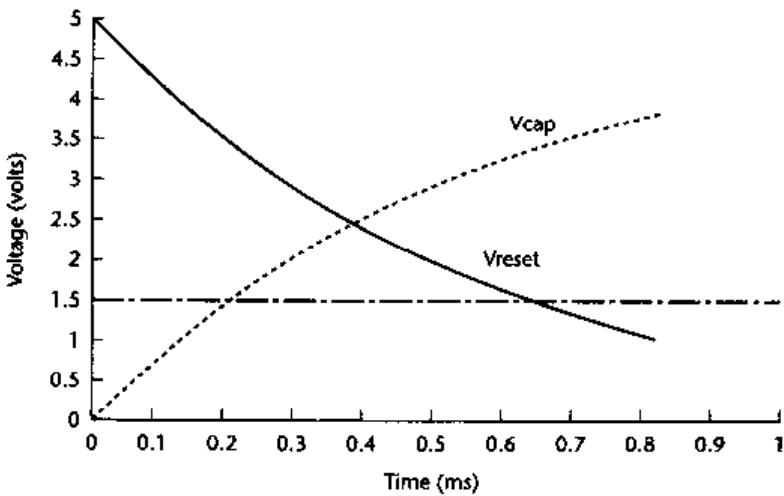


图 5.5 使用快速阻容复位电路

振荡器的起振时间

如果电路中的晶体振荡器的起振有延迟，那么阻容复位周期可能会在振荡开始之前结束。如果发生这种情况，芯片将不能复位。

晶体振荡器的典型启动时间是 0.1~10ms。然而，晶体振荡器的起振时间取决于它是否被正确地安装并配有合适的电容。这个问题将在晶体振荡器模式里详细讨论。

解决电源电压过低等问题

不幸地是，复位电路的潜在问题并不是只有在嵌入式设备第一次上电时才出现。例如，图 5.6 显示了在系统电源电压（标称 5V）变化时出现的两个问题。第一个（当时间=4s 时）是简单的电源“瞬变”，电源电压暂时地降到 0V。第二个问题（从第 14s 开始）是“电压过低”的状态。这指的是（市电电源）供电电压在一段时间以后显著地降低，但是不完全失灵。这类故障在由市电电源供电的系统中比较常见。

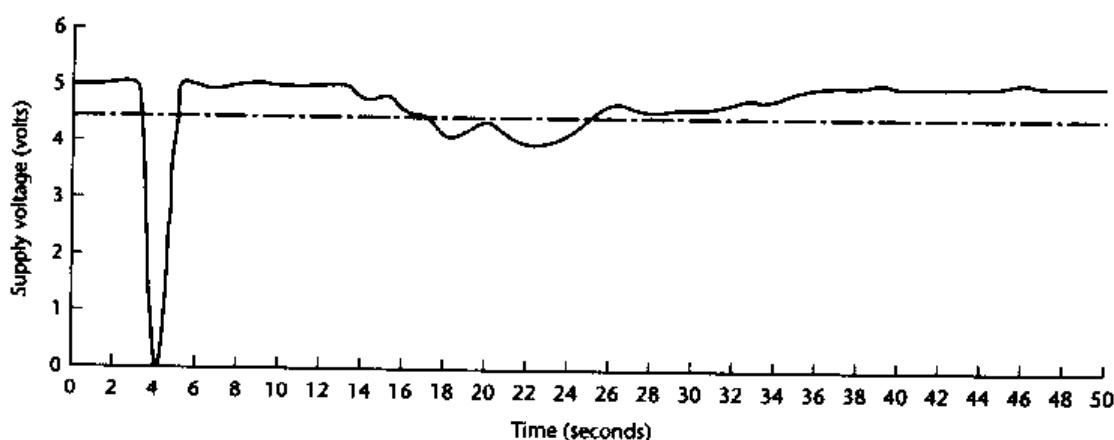


图 5.6 由于“电压瞬变”和“电压过低”而产生的电压波动的例子

为了保证系统以一种可预测的方式运行，要求其必须能够处理所有的电源问题。在大多数系统中，电源瞬变不会造成严重的危险。当电源失灵时，电压迅速地下降（到 0V），而系统停止运行。当电源恢复时，系统将以通常的方式复位。

电压过低的潜在问题更严重。如果电源电压降低到最小的工作电压之下（虽然略有不同，但对于 8051 系列大多数芯片一般为 4.5V），微控制器将停止运行。如果电压接着再次上升，微控制器将再次开始运行。然而，如果使用简单的阻容复位，芯片将不能复位。其结果很难预计，因此在电压有可能过低的情况下，阻容复位电路既不可靠又不安全。一些相关替代方案的技术参见“相关的模式和替代解决方案”。

可移植性

关于与各种不同性能的电源相关的可移植性问题，已经在本模式的其他地方讨论，在此不做详细讨论。

此外注意，正如在下面的例子中讨论的，并不是所有的 8051 系列微控制器都是“高电平有效”的复位。有一些是“低电平有效的”。虽然基本原则是一样的，但“高有效”和“低有效”复位的接线从根本上是不兼容的（参见例子：处理低电平有效复位）。

优缺点小结

- ◎ 阻容复位电路实现起来比较便宜。

- ⑥ 阻容复位容易理解并广泛用于其他微处理器和微控制器系统。
- ⑦ 如果系统由市电供电，而并不关注可靠性和安全性问题（主要考虑价格因素），这种技术就是一种好的解决方案。
- ⑧ 如果系统电源性能未知、发生变化或经常电压过低，复位操作并不总能成功。阻容复位通常不宜用于由市电供电且要求必须是可靠或安全的系统。

相关的模式和替代解决方案

可靠的复位模式描述了一种更加昂贵但是通常更加可靠的复位解决方案。

例子：Atmel 的 89C2051 使用晶振和阻容复位的最简电路

图 5.7 显示了 Atmel 的 89C2051 使用阻容复位的最简电路。该振荡器电路的详细资料请参见晶体振荡器模式。

注意，Atmel 的芯片不支持外部存储器，所以不提供/EA 引脚。详细资料参见“存储器模式”（第 6 章）。

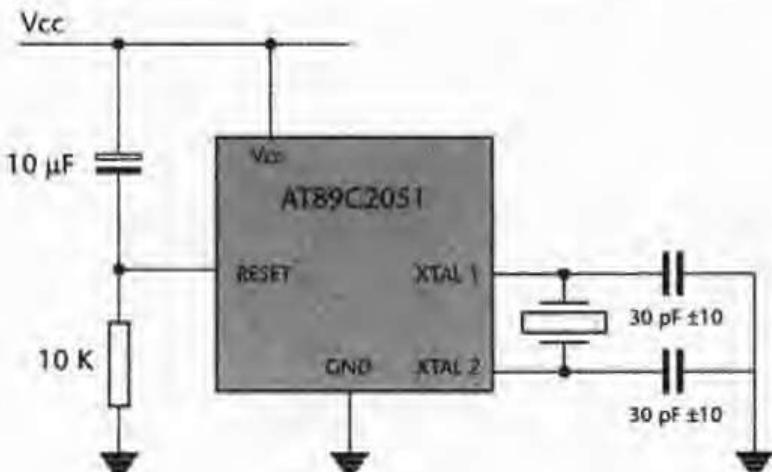


图 5.7 使用阻容复位的 Atmel AT89C2051 的最简电路

例子：处理低电平有效复位

迄今为止讨论的复位电路都具有“高电平有效”特性。这指的是正常情况下复位引脚将保持在低电平 (0V)。为了执行一次复位，必须在振荡器正在运行时将复位引脚拉高 (Vcc)。

然而，一些 8051 芯片具有“低电平有效”输入。这可以通过复位 (/RESET，而不是 RESET) 引脚的标注形式来识别。正如名字所暗示的，这些引脚在正常运行期间保持在高电平，而必须拉低（通常保持 24 个时钟周期）来执行复位。具有低电平有效输入的 8051 芯片的例子包括：Infineon 的 C509、C515C 和 C517A。这些都是广为使用的芯片。

低电平有效的复位电路的接线很简单。图 5.8 显示了一种对于各种低电平有效芯片的可行

电路。

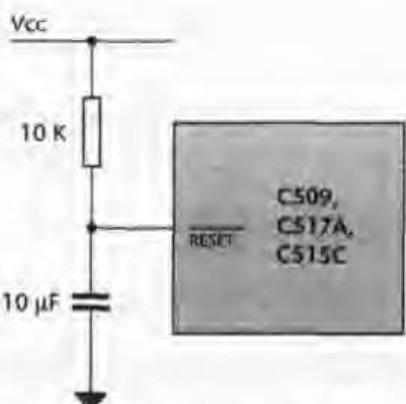


图 5.8 创建一个“低电平有效”的阻容复位电路

进阶阅读

可靠的复位

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

怎样为 8051 微控制器创建一个非常可靠的复位电路？

背景知识

关于复位电路的背景材料参见阻容复位。

解决方案

正如阻容复位讨论的，设计可靠的嵌入式系统的一个关键问题是：电源很少是理想的。例如，由市电供电的电源当第一次接通时能够在 100ms 左右达到正常的工作电压，之后可能受到电压变化（包括电压过低）的影响，这种变化在某种程度上是由于市电电源其他用户的负荷变化造成的。另一方面，1.5V 的普通电池在刚开始使用的时候常常有高达 1.7V 的电压，而在使用一段时间之后会降到 0.9V 或更低。

尽管仔细设计电源能够缓解这些问题，但是把系统设计为万一电源出现问题仍然能够正确复位以降低系统故障的风险常常更加可取。由 Dallas 和 Maxim 提供的可靠的复位电路，直接

解决了这个问题。

这些有用的芯片执行两个基本操作：

- 当系统“冷”上电时，为微控制器提供合适的（高电平有效或低电平有效）复位信号，保持至少 100ms 使振荡器（如果使用的话）和电源达到稳定的运行状态。
- 如果在正常运行期间电源电压低于某个预定值，复位周期就将开始，直到电源恢复为标准值之后 100ms 结束。这种特性既解决了（长时间或短时间的）电源彻底故障的问题，又解决了电压过低的问题。

总的说来，这些芯片很有效并且便宜。只要不是特别关注价格，它们还是非常值得推荐使用。

硬件资源

这个模式不涉及 CPU 或存储器的使用。

可靠性和安全性

通常，这种形式的复位电路比所有基于阻容的复位电路更加安全。

可移植性

各种可靠复位芯片通常对电源在较宽范围内的变化不敏感。因此，使用这种芯片将使设计更加具有可移植性，并且更少依赖于某种特别的电源。大多数芯片既有“高有效”又有“低有效”的版本，因此能与各种各样的 8051 芯片兼容。

优缺点小结

- ◎ 可靠的复位即使在“缓慢上升的”电源和发生电压过低的情况下也都能提供可靠性能。
- ◎ 比阻容复位的可选方案更昂贵。

相关的模式和替代解决方案

对于产品成本比可靠性和安全性更重要的应用场合，阻容复位提供了便宜的替代方案。

此外，获得可靠复位特性的两种更进一步的方式将在这里讨论。

片内复位电路

对于基于微控制器的系统，复位电路始终是一种挑战。既然可以设计小的复位芯片，为什么不将这种电路简单地包括在微控制器内呢？

实际上，在比较新的 8051 中已经包含了这种电路。例如，参见 Dallas 的 DS87C520/DS83C520 和 Philips 的 87LPC764。下一节中给出了一个 Dallas 的 87C520 使用内部复位模块的电路的例子。

微控制器/微处理器监视芯片

这里讨论的可靠的复位电路是简单而经济的选择。然而，在一些系统中，同时需要其他外部设备。Maxim 特别开发了一系列不同的微控制器监视芯片，这些芯片不仅包含了可靠的复位电路，而且具有更多的不同功能组合，诸如看门狗定时器甚至 RS-232 收发器。这里简单介绍了这种芯片的三个例子，详细说明请查阅 Maxim 的 WWW 网站^②。

- Max6330/I：复位控制，以及 3V/3.3V/5V 电源调节器。提供了既用于电源调整又用于复位电路的便宜的单个芯片。
- Max819：复位控制加上看门狗和电池切换。
- Max3320：复位控制，加上 RS-232 收发器。

此外，“1232”电源监视器（由 Dallas 和 Maxim 提供各种版本）将可靠的复位特性和看门狗定时器组合在一起。正如将在第 12 章看到的，在许多系统中，这是一种非常有用的合作。

例子：在基于 8051 系列芯片的系统中使用 Dallas 的 DS1812 “Econoreset”

作为本书介绍的标准 8051 芯片使用 DS1812 可靠复位的许多例子中的一个，请参阅第 26 章中的图 26.10。

例子：Infineon 的 C515C-8E 使用 Dallas 的“econo resets”

Infineon 的 C515C-8E 使用 Dallas 的 DS1811 的可靠复位的最简电路，如图 5.9 所示。

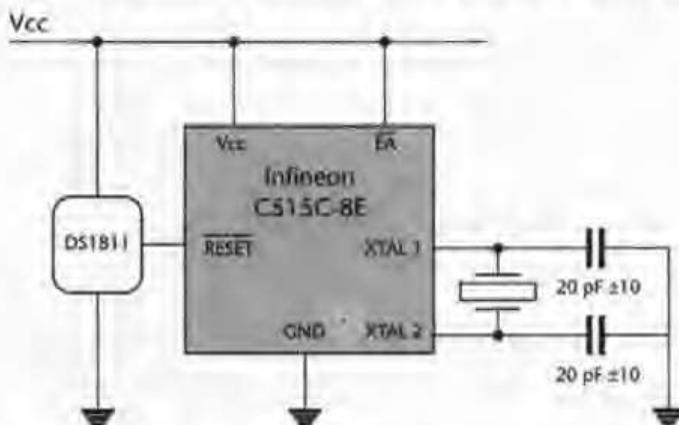


图 5.9 Infineon 的 C515C-8E 使用 Dallas 的 DS1811 的可靠复位的最简电路

例子：使用 Max810M 的 Infineon 的 C501-1E

Infineon 的 C501-1E 使用 Maxim 的 810M 的可靠复位的最简电路，如图 5.10 所示。

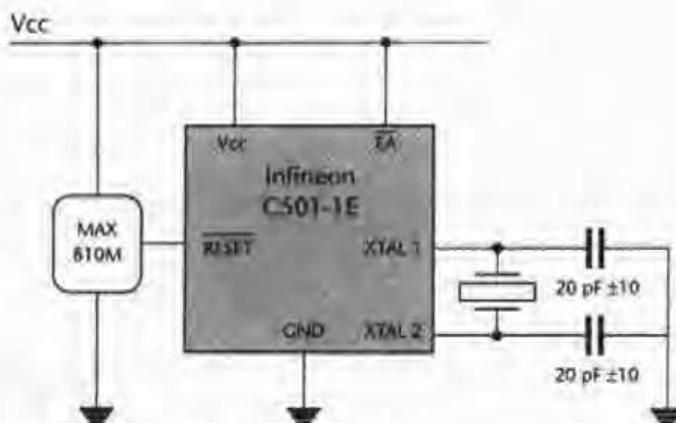


图 5.10 Infineon 的 C501-1E 使用 Maxim 的 810M 的可靠复位的最简电路

例子：使用片内复位电路的 Dallas 的最简电路

如前所述，一些比较新的 8051 芯片具有片内复位电路。

作为一个例子，将讨论 DS87C520（参见图 5.11），其电路运行如下。上电时，内部监视电路保持复位状态直到 V_{CC} 上升并超过“复位”电平。一旦超过这个电平，监视电路将允许振荡器输入并计数 65 536 个时钟周期，之后退出复位状态。这个上电复位（POR）时间间隔使电源和振荡器足以稳定。此外，在正常运行期间如果电源电压跌落，电源监视器将自动产生复位并保持。

注意，这种解决方案可以配合许多 Dallas 的高速以及超高速芯片使用，包括 80C320、80C323、83C520、87C520、87C530、87C550 和 89C420。

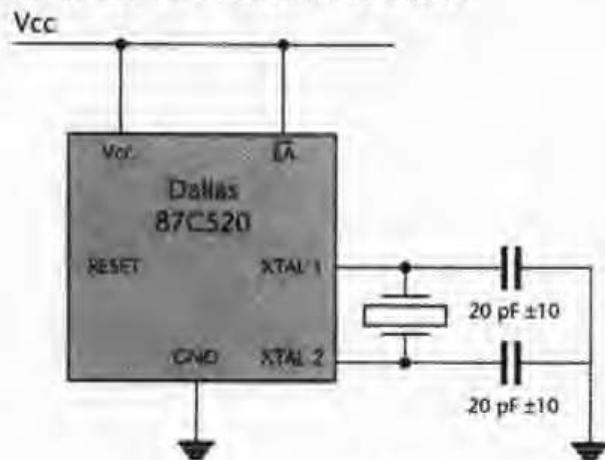


图 5.11 87C520 的最简电路

注意，不再需要外部复位电路。

进阶阅读

Chapter 6

存储器问题

引言

所有基于微控制器的实际系统都需要某种形式的非易失性程序存储器（用来存储程序代码）和某种形式的易失性存储器（用来存储数据和堆栈）。

大多数情况下，不扩展外部存储器而创建应用系统是有可能的。本章的第一个模式（片上存储器）讨论了如何通过有效地使用 8051 系列芯片的各种存储区来实现这个目标。

在一些系统中，必须扩展外部存储器。本章的其余模式（片外数据存储器和片外程序存储器）讨论了如何更好地为基于 8051 的系统扩展外部存储器。

请注意，本章主要涉及基于标准 8051 存储器体系结构的芯片。一些更新的 8051 芯片，诸如 Dallas 的 80C390、AD 公司的 AD μ C812 和 Philips 的 80C51MX 能够支持比早期的 8051 芯片更大容量的外部存储器。本章中将简要地介绍这种支持更大存储器的芯片，但是因为每个厂家都有其专用的解决方案，所以不再详细讨论它们。

片内存储器

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

怎样设计只使用内部存储器的 8051 电路？

背景知识

本节将介绍关于存储器的一些背景资料。

直接寻址与间接寻址

在关于微控制器的存储器的讨论中，常常看到“间接寻址（indirect addressing）”和“直接寻址（direct addressing）”这两个术语。这两个术语经常引起混淆，然而它们并不难理解。

想想无论使用哪种语言（例如，C语言或汇编语言）来编程，代码最终都必须转换为能够在所选择的微控制器上运行的机器代码指令。这套机器指令由硬件厂家定义。通过这个流程，即使高级语言中的很复杂的语句最终也被分解为基本操作，诸如“将数据块从某个存储区复制到另一个”。最终实现的机器指令的形式为“将存储器地址X中的内容移到（move）寄存器Y中”。

微控制器和微处理器中实现这种基本的“move”指令主要有两种方式：

- 使用直接寻址，存储单元的地址（即上面例子中的存储器地址X）作为指令的一部分而明确的给出。
- 使用间接寻址，存储单元的地址并不作为指令的一部分被明确的给出，而是在指令中包括另一个存储单元或寄存器的地址，该地址中所存储的内容是存储器地址X。

因为使用间接寻址意味着需要两个步骤来查找所需存储单元的地址，所以看起来它似乎比直接寻址要慢。然而，在（具有16位地址空间的）8位体系结构中普遍使用直接寻址，意味着所有“move”指令都必须包含两个字节的地址信息，因此将花费更多的时间对存储器进行存取。通常，许多芯片（包括8051）的折中方案都是有一小块存储区域能够直接寻址而大多数存储区域必须是间接寻址的。

注意，直接寻址和间接寻址之间的区别还有其他作用。例如，在8051系列芯片内部，有一块“特殊功能寄存器”存储区和另一个通用的存储区。两个存储区的大小相同（128字节），而且共用同样的地址范围。表面上，两个存储区域具有同样的地址是错误的，但在这种情况下并没有问题。其中的一块存储区只能间接访问，而另一块只能直接访问。因此，当编译程序翻译某个“C”语句时，必须选择合适的机器指令来保证对存储区的正确访问。在大多数情况下，这个流程对于程序员是完全隐藏的。

存储器类型

对于桌面系统，大多数设计人员和程序员可以忽略使用的存储器类型，而不存在潜在的危险。而在嵌入式工作平台上却很少是这样，因此简要地回顾一些不同的存储器类型。

首先是一个简短的历史回顾，用来解释一个重要的缩写的来源。在早期的主机和台式机系统上，数据的长期存储使用计算机磁带。对磁带的读写所需要的时间变化很大，取决于是绕动整个磁带还是仅仅绕动几个厘米。在这种背景下，出现了新的存储装置，用于当计算机正在运行的时候存储数据，但是在电源掉电时数据就会丢失。这种可读写的存储器芯片称为“随机存取存储器”（RAM）芯片。因为与基于磁带的系统不同，访问“随机选择的”存储单元需要同样的时间。

如今，磁带基本上已经消失了，但是缩写RAM仍然用来表示那些既能读又能写的存储装置。然而，自从RAM出现之后，不断出现新型存储装置，包括各种形式的ROM（只读存储

器)。因为 ROM 芯片在本质上也是“随机存取”的，缩写 RAM 现在最好的解释为“可读写存储器”。

动态随机存储器 (DRAM)

动态随机存储器是一种使用微电容来存储信息的读写存储器技术。由于电容将迅速地放电，所以必须经常刷新以保持需要的信息，由芯片上的电路负责进行这种刷新操作。和大多数当前的 RAM 类型一样，当电源掉电时信息将丢失。

通常，DRAM 简单并且比较便宜。

静态随机存储器 (SRAM)

静态随机存储器是一种使用双稳态电子电路存储信息的读写存储器技术。它不需要刷新，但是电路比 DRAM 更复杂，而且价格是 DRAM 的几倍。然而，存取时间是 DRAM 的三分之一。

掩模只读存储器 (ROM)

完全的只读存储器 (ROM) 毫无价值。从设计人员的角度来看，实用的 ROM 的最基本类型是只读的，然而，厂家能够在制造芯片时候按照该芯片的用户提供的“掩模”来写这种存储器。因此，这种芯片有时被称为“工厂可编程只读存储器”或“掩模只读存储器”。工厂或掩模编程并不便宜，因此不适用于小批量生产。因此，如果出错其代价会非常高，为最初的掩模编写代码将是一个非常磨练人的过程。ROM 的存取时间通常比 RAM 要慢，大致是 DRAM 的 1.5 倍。

值得注意的是，即使在强电磁干扰下，掩模 ROM 也能保持它们的内容。这种特性与一些可擦除芯片形成鲜明的对比，可擦除芯片存在数据损坏的风险。例如，强紫外线的照射（紫外线可擦除可编程只读存储器 UV-E PROM，参见本节后面的内容）或强电场（电可擦除可编程只读存储器 EEPROM，参见本节后面的内容）都是导致数据损坏的诱因。

许多 8051 系列芯片都有片内掩模可编程只读存储器。

可编程只读存储器 (PROM)

可编程只读存储器的名称听起来是自相矛盾的，然而它确实是这样。它实际上是一种“一次写、多次读”(WORM) 或“一次性可编程”(OTP) 存储器。实际上，使用 PROM 编程器来烧断芯片中微小的“熔丝”。一旦烧断，这些熔丝将不能被修复。这种芯片比较便宜。

许多现代的 8051 系列芯片都有 OTP 只读存储器。

紫外线——可擦除可编程只读存储器 (UV-E PROM)

与 PROM 一样，UV-E PROM 也是电可编程的。与 PROM 不同的是，它们有一个石英窗用来将芯片内部暴露在紫外线下，以擦掉存储器中的内容。这种擦除过程需要几分钟，在擦除之后石英窗将被一个对紫外线不透明的标签覆盖。这种形式的 EPROM 能够经得起数以千计的编程/擦除周期。

虽然 UV-E PROM 比 PROM 更加灵活，而且曾经非常常用。然而和 EEPROM 相比，UV-E PROM 显得比较原始。它们可用于样机设计，但如果用于生产就相对昂贵了。

许多老的 8051 系列芯片都有片内 UV-E PROM。

电可擦除可编程只读存储器 (EEPROM, E²PROM)

EEPROM 是一种更加方便使用的 EEPROM 类型，它既能够电编程又能够电擦除。这并不意味着它们能够在所有情况下简单地用于替换 RAM。这不仅是因为写 EEPROM 是一个非常慢的过程，而且其执行写操作的次数也是有限的。

许多 8051 系列芯片都有片内 EEPROM。

闪存

闪存 (Flash ROM) 不仅仅是用来对越来越长而且越来越不准确的缩写的简化，也是目前最常用的 ROM 类型。正如其名称所暗示的，它通常能够比 EEPROM 更加迅速地编程。此外，许多 EEPROM 常常需要较高的 (12V) 编程电压，而闪存器件通常能在标准电平 (3V/5V) 下编程。

一些 8051 系列芯片具有片内闪存。

解决方案

图 6.1 显示了 8051 系列的存储器映像。

为了更好地使用这些内部存储器，或者为系统选择更合适的芯片，需要理解不同存储区的含义。现在就讨论这方面的问题。



图 6.1 8051 系列的主要存储区的原理示意图

注意，虚线边界内显示的部分并不适用于所有的系统。同时注意，随着 8051 系列的发展，一些扩展 8051 芯片增加了额外的存储器区域。

程序代码区

正如名称所暗示的，8051 系统中的程序存储器的主要任务是存储（可执行的）程序代码。这些代码由 C 语言编译程序自动生成。注意，程序代码在 ROM 中“就地”运行，而不是复制到 RAM 中运行。

除程序代码之外，代码区还能用于存储（只读）数据，诸如查表数值。这通常是一种非常好的想法。许多 8051 芯片都具有相对多的 ROM 可供利用（64KB 很常见），但是只有少量的

RAM 可用（通常至多 4KB，一般不超过 256B）。只要有可能，将只读数据放在 ROM 中通常是个好办法。

使用 ROM 来存储数据表在提高性能上也很有意义。例如，如果程序代码需要计算正弦或余弦，则在大多数系统上都需要进行大量的 CPU 操作（一般是超过 3000 次 CPU 操作）。如果能够将相关计算的结果放在一个数组中（使用查表法），将使 CPU 的负荷降低为原来的千分之一左右。

将数据存储在 ROM 中很容易做到。例如，使用 Keil 的 C51 编译程序以及程序关键字，能够采用以下方法在 ROM 中存储一个大数组：

```
int code CRC16_table[256] =
{0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7, 0x8108,
 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef, 0x1231, 0x2101, }
 // etc...
```

DATA、BDATA 和 IDATA 区

可以得到最多 256B 的内部数据存储器，具体取决于使用的 8051 芯片。内部数据存储器的访问速度通常很快，因为只需要使用 8 位地址即可寻址。

内部数据区分为三个重叠的区域：DATA、IDATA 和 BDATA。下面逐个讨论这些存储区域。

使用 DATA 区

DATA 区指向内部数据存储器的前 128 个字节。这里存储的变量可以使用直接寻址很快访问。

如果使用 Keil 推荐的小内存模式，C51 编译程序默认设置为使用 DATA 区。也可以使用 data 关键字来明确规定某个变量应存储在 DATA 区。例如：

```
char data Input1;
unsigned int data Loop_Control;
```

使用 IDATA 区

IDATA 区指向内部数据存储器的所有 256 个字节（包括与之重叠的 DATA 区的 128 个字节）。这里存储的变量使用间接寻址，访问速度比 DATA 变量要慢。idata 关键字可以用来明确地声明变量将存储在 IDATA 区，方法如下：

```
char idata Input2;
unsigned int idata Loop_Control2;
```

注意，IDATA 区通常最好用作堆栈区。一般而言，最好将这个区域留给编译程序使用。

使用 BDATA 区

BDATA 区与 DATA 区重叠。具体而言，它指向内部 DATA 区中 16 个字节大小的可位寻址存储器（地址 0x0020~0x002F）。

通过 bit、bdata 和 sbit 关键字来使用这个区域，并声明能够按位访问的数据类型。参考以下例子：

```
// 定义无符号的字符型变量 Bit_addressable
```

```

// 可以取值 0~255
unsigned char bdata Bit_addressable;
// 定义位变量 Flag
// 可以取值 0 或 1
bit Flag;
// 声明变量 Bit_addressable 中的位
// 注意关键字 sbit 的使用(“而不是” bit)
// 注意，这个声明不占用存储器
sbit Bit0 = Bit_addressable^0;

```

特殊功能寄存器 (SFR) 存储区

如图 6.1 所示，所有的 8051 都提供最多 128B 的存储器作为特殊功能寄存器 (SFR)。SFR 是位、字节或字（2 个字节）长度的寄存器，用来控制定时器、计数器、串行输入/输出、端口输入/输出和各种外设。端口 SFR 将在端口输入/输出模式中讨论。

注意，正如在“背景知识”中简要介绍的，128~255 字节的 IDATA 区和 SFR 区共用同样的地址空间。然而，这两个存储器区域使用不同的寻址方式访问。128~255 字节的 IDATA 区只能间接寻址，而特殊功能寄存器只能直接寻址。

外部数据存储器

外部数据存储器最大可以为 64KB。与内部存储器相比，对外部存储器的访问比较慢。

外部数据存储器可以被分成两个区域。XDATA 区可以指向所有 (64KB) 数据地址空间中的任意地址。PDATA 区指向这个地址空间的前 256 个字节。如果正确地编程，访问 PDATA 变量比访问外部数据空间的其他变量的速度更快。

内部的“外部”存储器

值得注意的是，尽管 XDATA 存储器映射为 XDATA 区，但它并不是必须在物理上位于微控制器之外。实际上，许多现代的芯片都包含片内 XDATA RAM。例如，Dallas 的 83C520 包含 1KB 的“XRAM”，而 infineon 的 C509 包含 3KB 的 XRAM。正如后面“控制对内部和外部存储器的访问”一节讨论的，这些 RAM 区的使用方法和外部存储器一样。

避免各种程序和数据区之间的混乱

注意，许多 8 位微控制器都具有单个 (64KB) 存储区，由程序和数据共用。就 8051 而言，最多有 64KB 的程序区和 64KB 的数据区可用。因为这两个区域共用同样的地址空间，所以芯片（和编译程序）需要一种正确访问各个区域的方法。

区别对程序和数据访问的主要方式是使用/RD、/WR 和/PSEN 引脚。

与直接寻址及间接寻址一样，这些过程对于工作在高级语言环境下的程序员通常是隐藏的。

取决于复位时“外部存取”(/EA) 引脚的状态，大多数 8051 系列芯片可以工作在两种工作模式下。当/EA 为低时，片内程序（而不是数据）存储器被禁用，64KB 程序地址空间全部供外部存取使用。当/EA 为高时，使用片内程序存储器。在这种情况下，只有当

程序需要存取超过片内存储器地址范围的存储单元时，才会访问外部存储器。

忘记将/EA引脚拉高，是8051的初学者在设计中常犯的错误。

控制对内部和外部存储器的访问

注意，与程序存储器不同，/EA引脚在复位时的状态不会影响片内数据RAM。片内数据RAM始终是使能和可存取的。另一个区别和片内RAM的存在对外部数据存储空间的影响有关。对于有最多256字节的片内(IDATA)RAM的CPU，完整的64KB外部数据空间同样是可用的。对于具有片内XDATA存储器的芯片，该区域将与外部存储器重叠。注意，对于不同的芯片，片内XDATA存储器映射的地址空间也不同。大多数芯片将这些存储器映射到地址0x00，但有一些芯片使用不同的地址。

注意，这些讨论不适用于精简8051芯片，因为那些芯片不支持外部存储器，所以不需要/EA引脚。此外，一些8051系列的芯片（特别是8031及其变型，和一些扩展8051）没有片内ROM，所以总是需要使用外部程序存储器。

不同8051芯片上的内部存储器

一系列8051芯片上的各种存储器模块的例子在表6.1中给出。

表6.1 一系列8051系列芯片上可选择使用的存储器

芯 片	基 本 RAM (DATA 和 IDATA)	扩 展 RAM (XDATA)	ROM (CODE)	备 注
Analog Devices ADμC812	256×8-bit	640×8-bit data Flash EEPROM	8K×8-bit Flash EEPROM	16M×8-bit 外部数据地址 空间； 64K×8-bit 外部数据地址 空间
Atmel 89C1051	128×8-bit	0	1K×8-bit Flash EEPROM	
Atmel 89C2051	128×8-bit	0	4K×8-bit Flash EEPROM	
Atmel 89C4051	128×8-bit	0	12K×8-bit Flash EEPROM	
Atmel 89S53	256×8-bit	0	16K×8-bit OTP EEPROM	
Dallas 83C520	256×8-bit	1K×8-bit	16K×8-bit mask ROM	
Dallas 87C520	256×8-bit	1K×8-bit	0	
Dallas 80C390	256×8-bit	4K×8-bit	8K×8-bit OTP EEPROM	4M×8-bit 外部数据地址 空间； 4M×8-bit 外部数据地址 空间

续表

芯 片	基本 RAM (DATA 和 IDATA)	扩展 RAM (XDATA)	ROM (CODE)	备 注
Infineon C501-1E	256×8-bit	0		
Infineon C501-1R	256×8-bit	0	8K×8-bit mask ROM	
Infineon C501-L	256×8-bit	0	0	
Infineon C505C-2R	256×8-bit	256×8-bit	16K×8-bit mask ROM	
Infineon C505C-4EM	256×8-bit	256×8-bit	32K×8-bit OTP ROM	
Infineon C509-L	256×8-bit	3K×8-bit	0	
Infineon C515C-8E	256×8-bit	2K×8-bit	64K×8-bit OTP EEPROM	
Intel 8031	256×8-bit	0	0	
Intel 8032	128×8-bit	0	0	
Intel 87C51FA	256×8-bit	0	8K×8-bit UV-EEPROM	
Philips 80C51MX	256×8-bit	?	?	
Philips 83C751	64×8-bit	0	2K×8-bit mask ROM	8M×8-bit 外部数据地址 空间; 8M×8-bit 外部数据地址 空间
Philips 87C751	64×8-bit	0	2K×8-bit UV-EPROM	
Philips 87LPC764	128×8-bit	0	4K×8-bit EEPROM	

注意：最近推出的 80C390 分别支持最多 4MB 的外部程序和数据区。同时还要注意，有关 80C51MX 的数据是基于 Philips 在正式推出第一个芯片之前发布的初步数据。

硬件资源

片内存储器是纯粹的硬件（存储器）资源提供者。

可靠性和安全性

假定有两个相同的系统，使用几乎相同的软件，但是一个只使用内部存储器，而另一个使用外部存储器（用于存储程序及数据）。其他东西都相同，很有可能“内部”系统将被证明是更加可靠的。这在一定程度上是因为使用内部存储器降低了接线或设计错误的机会，还有部分是出于减少外部接线（“相当于天线”）使系统更不易受到电磁干扰的影响，而且在某种程度上

是因为在振动及高湿度的情况下（使用外部存储器的方案）每一个焊点都存在失效的风险。

此外，ALE 引脚将是电磁干扰的来源。这些引脚是访问外部存储器所必需的，但在一些芯片中，如果使用内部存储器，这些引脚的操作都能够被禁止。这将降低该系统导致其他系统出错的可能。

在几乎所有的情况下，“内部”解决方案不但生产起来便宜而且结构上简单。结论是显而易见的，尽可能地使用内部存储器。

可移植性

一般而言，最具可移植性的 8051 程序假定只使用少量的程序存储器（比方说 4KB 的某种 ROM）和 128 字节的 RAM。即使在大多数最简单的 8051 中，这种组合也是可以得到的。

在许多现代的 8051 中，有更多的存储器（一般 256 字节的 RAM）可用。一些现代的芯片还有片内 XRAM。然而，这些功能使用的越多，程序就越难以移植到 8051 系列的其他微控制器上。

优缺点小结

使用内部存储器（代替外部存储器）可能有以下影响：

- ◎ 更低的系统成本。
- ◎ 增加硬件可靠性。
- ◎ 降低电磁辐射（如果能够禁止 ALE 功能）。
- ◎ 在大多数情况下，可用的数据存储器都是有限的。

相关的模式和替代解决方案

参见片外数据存储器和片外程序存储器。

例子：Philips 的 8XC552 上的内部存储器

作为一个可用存储器选择的例子，下面讨论 Philips 的 8XC552。

8XC552 包含 8KB 的片内程序存储器，通过使用外部存储器，程序存储器可以扩展到 64KB。当 EA 引脚为高时，8XC552 从内部 ROM 取指令，除非地址超过 1FFFH。2000H~FFFFH 的存储单元将从外部程序存储器读取。当 EA 引脚为低时，所有指令来自于外部存储器。ROM 地址的 0003H~0073H 用于中断服务程序。

内部数据存储器被分成三个部分：RAM 的低 128 字节、RAM 的高 128 字节和 128 字节的特殊功能寄存器区。RAM 的低 128 字节可以直接和间接寻址。而 RAM 地址的 128~255 字节和特殊功能寄存器区共用同样的地址空间，它们通过不同的寻址方式来访问。128~255 字节的 RAM 区只能间接寻址，而特殊功能寄存器只能直接寻址。内部 RAM 的其他方面和 8051 相同。通过设置 8 位的堆栈指针，堆栈可以位于内部 RAM 的任何地方。堆栈深度最大为 256 字节。

除程序计数器和四个寄存器组之外，特殊功能寄存器（只能直接寻址）包含了 8XC552

所有的寄存器。这 56 个特殊功能寄存器中的大多数用来控制片内外设硬件。其他寄存器包括运算寄存器 (ACC、B、PSW)、堆栈指针 (SP) 和数据指针寄存器 (DHP、DPL)。SFR 中的 16 个寄存器包含了 128 个直接寻址的位地址。

例子：不同存储区的访问速度的比较

各种存储区中所存储数据的典型访问时间（按指令周期数计算）如下：

- 对 DATA 区的访问需要一个周期
[直接存取]
- 对 IDATA 区的存取需要 2 个周期
[复制 8 位地址到寄存器 (1 个周期), 然后 1 个周期的移动指令]
- 对 PDATA 区的存取需要 3 个周期
[复制 8 位地址到寄存器 (1 个周期), 然后 2 个周期的移动指令]
- 对 XDATA 区的存取需要 4 个周期
[复制 16 位地址到寄存器 (2 个周期), 然后 2 个周期的移动指令]
- 对代码区的存取需要 4 个周期
[复制 16 位地址到寄存器 (2 个周期), 然后 2 个周期的移动指令]

虽然这些数字是典型的，但是很难精确预计比较对不同存储区中的变量的访问时间，部分是因为结果取决于使用不同的编译程序选项带来的变化。

例子：使用 Infineon 的 C515C 的内部 XRAM 存储器

Infineon 的 C515C 是一种强大的扩展 8051 芯片，广泛用于一系列不同的系统，特别是它具有良好性能和片内 CAN 模块。

除了标准 (256 字节) IDATA RAM 之外，C515C 具有 2KB 的片内 XRAM。

使用这些存储器前，必须知道（例如与 Dallas 的 520 不同）这些存储器不是从地址 0x0000 开始映射的，其起始地址是 0xF800。

如果使用 Keil 编译程序，则必须通过“Size / Location”菜单项为链接程序提供这些信息。应该在 XDATA 的起始地址一栏输入“0F800”。对于现有版本的编译程序，如果遗漏了前面的“0”将导致问题的产生。

进阶阅读

片外数据存储器

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。

- 需要为该系统设计相应的硬件。

问题

如何为标准 8051 微控制器扩展最多 64KB 的外部 RAM?

背景知识

在为 8051 扩展外部数据存储器之前，必须首先理解存储器接口。图 6.2 简要地显示了这些接口，在随后的讨论中将以此为参考。

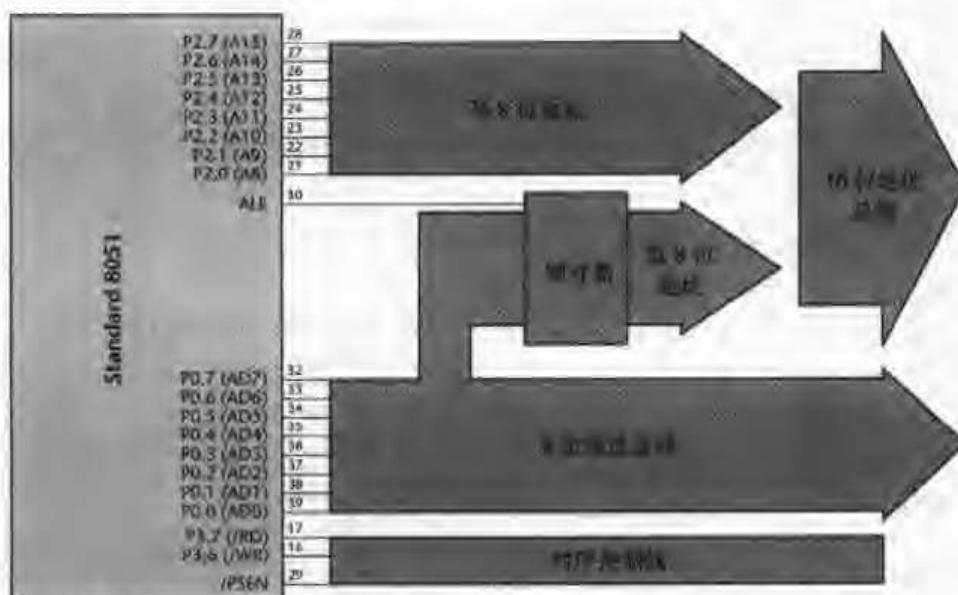


图 6.2 8051 的存储器接口

地址总线 (P0.0~P0.7)

端口 0 被用作多路复用的地址/数据总线，输出低 8 位地址 (A0~A7) 和输入/输出 8 位数据 (D0~D7)，同时用于程序和数据存取。

数据总线 (P2.0~P2.7)

P2 输出高 8 位地址 (A8~A15)，用于所有的外部程序存取。访问 16 位地址的外部数据同样如此。

注意一个特例：那些使用片内程序存储器和只有 256 字节外部数据存储器的系统，能够使用 P2 作为通用的输入/输出。

ALE (地址锁存选通)

ALE 用来解复用 AD0~AD7 总线。在外部存取周期开始时，ALE 为高则 CPU 产生 A0~A7，当 ALE 变低时 AD0~AD7 总线应该被外部锁存。

注意，在大多数基于 8051 的系统中，即使在访问内部程序和数据期间，ALE 也总是有效的。然而，在一些更现代的 8051 设计中，如果不需要访问外部存储器，则可以禁止 ALE 功能。这将有助于降低电磁辐射。

同时还要注意，当不需要访问外部存储器时，ALE 可以作为一个运行在六分之一振荡器频率的连续时钟。这个输出能被用于控制外部回路的定时。

PSEN（程序存储器使能）

PSEN 是外部指令（程序存储器）访问的读选通。与 ALE 不同，PSEN 在内部存取期间无效。将在片外程序存储器模式讨论 PSEN 的使用。

RD（读数据）

RD 是用于访问外部数据的读选通，而且类似 PSEN，在内部存取期间无效。

WR（写数据）

WR 是用于访问外部数据的写选通，类似于 PSEN 和 RD，在内部存取期间无效。

解决方案

有了存储器接口的基本知识（参见“背景知识”），只要在选择元件时再稍加注意（参见图 6.3），为 8051 微控制器扩展外部数据存储器还是很容易的。

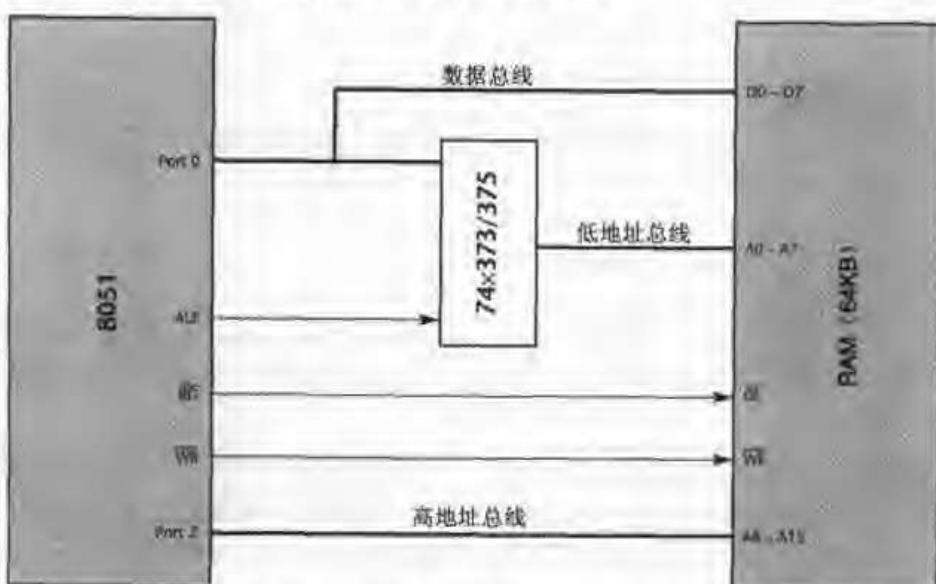


图 6.3 为 8051 微控制器扩展外部数据 (RAM) 存储器

注意，如果微控制器不是 CMOS 芯片而存储器件是 CMOS 芯片，则需要在端口 0 接上拉电阻（图中没有显示）。使用 DIL（或类似的）10K 排阻很容易实现。

注意，74×373 和 74×375 功能上相同，但是具有不同的引脚排列。通常，375 的排列便于接线。两个锁存器都可以有不同的速度等级。毫无疑问，价格随速度的提高而增加。

用于各种时钟频率的锁存器和(RAM)存储器的推荐组合在表 6.2 中给出。这些是从 Dallas 的应用手册 89 中摘录的。然而，能与 Dallas 芯片配合运行的锁存器和存储器组合同样适用于大多数其他（通常更慢的）8051 芯片。

硬件资源

使用外部存储器有较大的资源问题。它需要使用两个端口 (P0 和 P2)，加上端口 3 的两个引脚 (P3.6、P3.7)，而可用的端口引脚数从 32 个降低到 14 个。更多的相关讨论参见下面内容。

可靠性和安全性

正如在片内存储器讨论的，扩展外部存储器可能降低系统的可靠性。尽可能使用片内解决方案。

然而，因为许多 8051 没有足够的 RAM 来支持大规模系统，在一些系统中将被迫使用片外 RAM。当使用片外程序或数据存储器时，请注意，缺乏经验的 8051 开发人员最常见的一个错误是：当使用外部存储器时，仍然使用 P0、P2 或 P3 作为普通输入/输出口。千万不能这样做（原因在“硬件资源”中讨论）。

简而言之，如果使用外部存储器，则不能将 P0 和 P2 用于其他功能，并且必须在写端口 3 时谨慎小心。

例如，任何类似这样的语句：

```
P3 = AD_data;
```

都有可能导致灾难。

正确的做法是使用 sbit 变量来保证只写“安全的”端口引脚，详细说明参见端口输入/输出。

可移植性

外部存储器的硬件设计通常是可移植的。然而，如果从一个“低速的”处理器升级到一个“高速的”处理器（或仅仅增加晶体频率），则必须确保外部锁存器和存储器模块足够快速。有关建议参考表 6.2。

使外部存取设计保证可移植的惟一方法是：总是使用尽可能快速的外部模块。这种方法有成本上的问题，但是，如果能够生产 100 000 个单一的通用（高速的）电路板而不是小数量的各种“低速的”、“中速的”和“高速的”版本，你将发现“单一通用的”解决方案不但可靠而且性价比高。

优缺点小结

- ◎ 许多 8051 都没有用于支持大系统的足够 RAM，本模式解决了这个问题。
- ◎ 正如在片内存储器讨论的，扩展外部存储器可能会降低系统的可靠性。应尽可能地

使用片内解决方案。

表 6.2 用于各种时钟频率的锁存器和 RAM 存储器的推荐组合（由 Dallas 应用手册 89 改编而来）

时钟频率 (MHz)	推荐的锁存器	推荐的存储器速度
33.0	74F373/375	55 ns
29.5	74F373/375	55 ns
25.0	74F373/375	80 ns
21.0	74F373/375	100 ns
20.0	74F373/375	120 ns
19.8	74AC373/375	120 ns
18.4	74AC373/375	120 ns
16.0	74HC373/375	120 ns
14.7	74HC373/375	120 ns
14.3	74HC373/375	150 ns
12.0	74HC373/375	170 ns
11.1 (11.059)	74HC373/375	200 ns
7.4	74HC373/375	200 ns
<=1.8	74HC373/375	200 ns

相关的模式和替代解决方案

关于片内存储器的使用的讨论参见片内存储器。

例子：C509 使用外部 RAM 和内部 XRAM

Infineon 的 C509 微控制器有 3KB 的内部 XRAM。默认时，当芯片复位后 XRAM 是禁止的，可以存取完整的 64KB 的外部数据存储器。

然而，如果使能 XRAM，内部 XRAM 和外部 RAM 存储区将重叠。就 C509 具体而言，XRAM 映射到高位的 3KB 的数据存储器，如图 6.4 所示。

内部 XRAM 通过特殊功能寄存器 SYSCON1 使能。具体而言，PRGEN 位必须设置为 0 而 SWAP 位必须设置为 1。

例子：Dallas 的 8XC520 使用外部 RAM 和内部 SRAM

Dallas 的 8XC520 微控制器有 1KB 的内部“SRAM”。默认时，当芯片复位后 SRAM 是禁止的，可以存取完整的 64KB 的外部数据存储器。然而，如果 SRAM，内部 XRAM 和外部 RAM 存储区将重叠。就 8XC520 具体而言，SRAM 映射到低位的 1KB 的数据存储器，如图 6.5 所示。

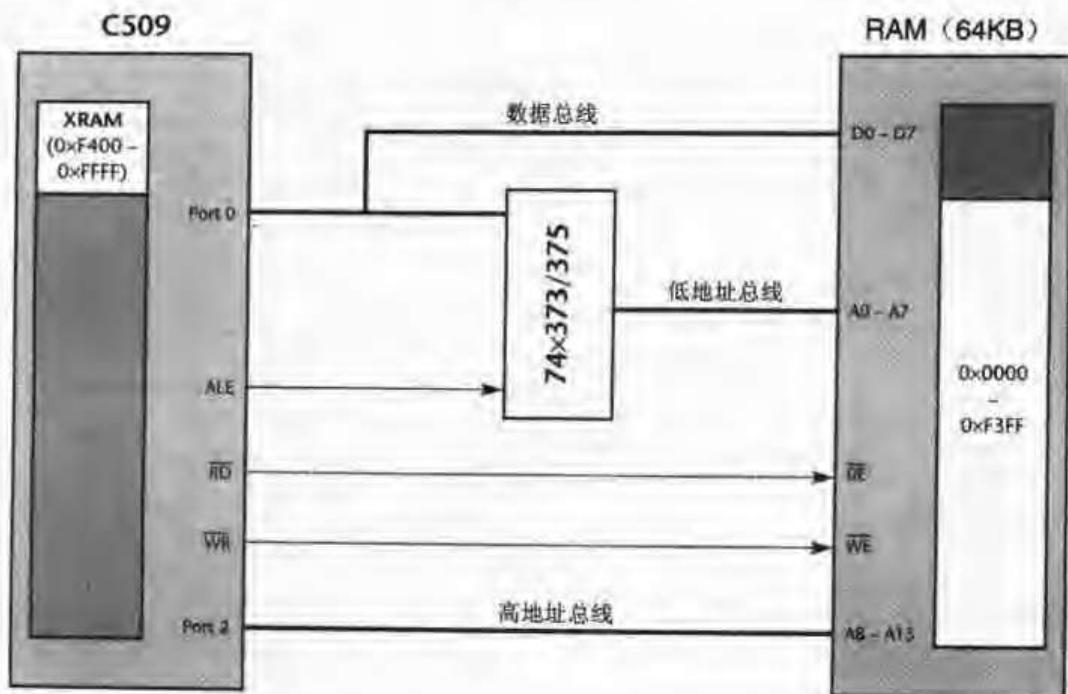


图 6.4 Infineon 的 C509 微控制器扩展外部 RAM

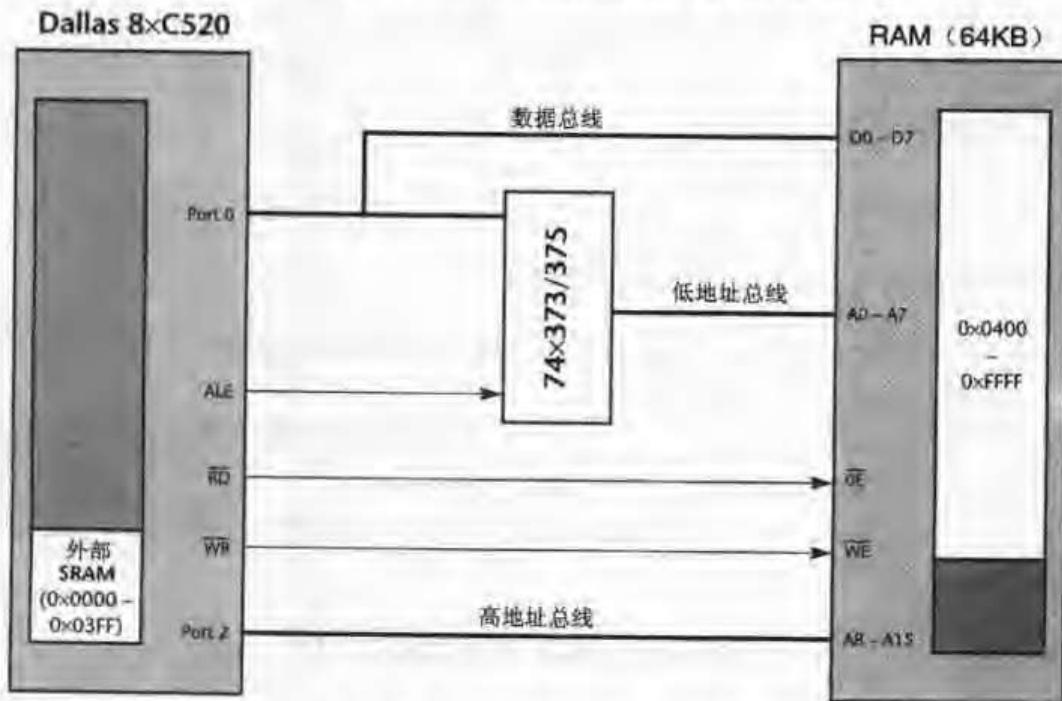


图 6.5 Dallas 的 8XC520 微控制器扩展外部 RAM

进阶阅读

片外程序存储器

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

如何为 8051 微控制器扩展最多 64KB 的外部程序存储器（通常为 ROM）？

背景知识

背景资料参见片外数据存储器。

解决方案

如果已经决定必须要使用外部程序存储器，其扩展是简单的。图 6.6 给出了基本电路。

用于各种时钟频率的锁存器和(ROM)存储器的推荐组合在表 6.3 中给出。这些是从 Dallas 的应用手册 89 中摘录的。正如在片外数据存储器提到的，能与 Dallas 的芯片配合运行的锁存器和存储器的组合同样适用于大多数其他（通常更慢的）8051 芯片。

表 6.3 用于一系列时钟频率的锁存器和(ROM)存储器的推荐组合（由 Dallas 应用手册 89 改编而来）

时钟频率 (MHz)	74F373/375	74AC373/375	74HC373/375
33.0	55	55	N/A
29.5	70	70	N/A
25.0	90	70	55
22.1	90	90	70
20.0	120	90	90
19.8	120	120	90
18.4	120	120	90
16.8	150	120	120
16.0	150	150	120
14.7	150	150	120
14.3	150	150	150
12.0	200	200	150
11.1(11.059)	200	200	200
7.4	250	250	250
<= 1.8	250	250	250

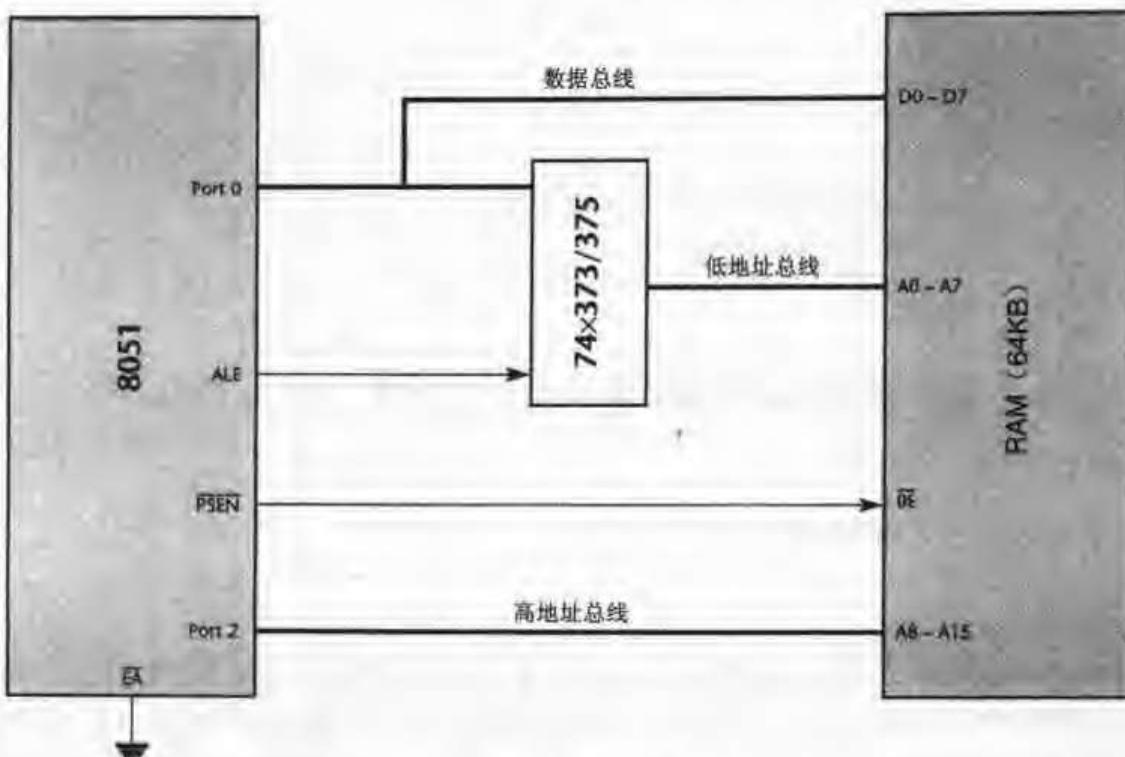


图 6.6 扩展外部 ROM 存储器

注意，如果微控制器不是 CMOS 芯片而存储器件是 CMOS 芯片，则需要在端口 0 接上拉电阻（图中没有显示）。使用 DIL（或类似的）10K 排阻很容易实现。同时注意，在 EA 引脚的控制之下，微控制器从外部存储器读取指令。

硬件资源

正如在片外数据存储器讨论的，使用外部存储器有较大的资源问题：它需要使用两个端口（P0 和 P2），加上端口 3 的两个引脚（P3.6、P3.7）。使可用的端口引脚数从 32 个降低到 14 个。更多的相关讨论参见下面内容。

可靠性和安全性

正如在片内存储器讨论的，扩展外部存储器可能降低系统的可靠性。应尽可能地使用片内解决方案。

大多数情况下，如果使用合适的有片内 ROM 的 8051 芯片，则可以避免扩展外部 ROM。详细说明参见“相关的模式和替代解决方案”。

可移植性

外部存储器的硬件设计通常是可移植的。然而，如果从一个“低速的”处理器升级到一个“高速的”处理器（或仅仅增加晶体频率）必须确保外部锁存器和存储器模块足够快速。有关建议参考表 6.3。

保证外部存取设计可移植的惟一方法是：总是使用尽可能快速的外部模块。这种方法有成本上的问题，但是，如果能够生产 100 000 个单一通用的（高速的）电路板而不是小数量的各种“低速的”、“中速的”和“高速的”版本，则会发现“单一通用的”解决方案不但可靠而且性价比高。

优缺点小结

② 有可能的话，使用内部程序存储器是最好的选择！

相关的模式和替代解决方案

在为基于 8051 的系统扩展 ROM 之前，应仔细地考虑一下。有许多具有大量片内程序存储器的 8051 芯片可用。作为一个例子，表 6.4 给出了一组 Philips 最新的 8051 芯片。

表 6.4 各种最近的 Philips 的 8051 芯片的可用的（内部）ROM 的例子

芯 片	Flash 大小	注 释
89C51	4K	Standard 12-clock machine cycle
89C52	8K	Standard 12-clock machine cycle
89C54	16K	Standard 12-clock machine cycle
89C58	32K	Standard 12-clock machine cycle
89CRC+	32K	Standard 12-clock machine cycle
89CRD+	64K	Standard 12-clock machine cycle
89CRB2	16K	6-clock machine cycle(12clock optional)
89CRC2	32K	6-clock machine cycle(12clock optional)
89CRD2	64K	6-clock machine cycle(12clock optional)

注意，其他厂家也提供类似的芯片，但是 Philips 的芯片最全面。

正如表中给出的，现在的 8051 芯片有大量的片内 ROM 可用（并具有良好的性能）。当然，其他厂家也有类似芯片。

扩展外部程序存储器有三个主要原因：

- 需要某个片内硬件模块（例如，CAN 总线、模数转换器及硬件数学处理），而找不到一个既有足够的 ROM，又有所需的外设的 8051 芯片。
- 需要外部 RAM。因为已经建立了所需的外部存储器接口，所以再扩展外部 ROM 是经济的，而不使用一个有片内 ROM 的芯片。
- 需要更新系统的运行程序。例如，如果系统运行在一个相对昂贵的（扩展）8051 芯片上，当需要更新程序时，替换一个小的 ROM 芯片而不是替换整个微控制器或许更经济合算。

例子：扩展 ROM 和 RAM 存储器

图 6.7 说明了如何为系统扩展程序存储器和数据存储器。

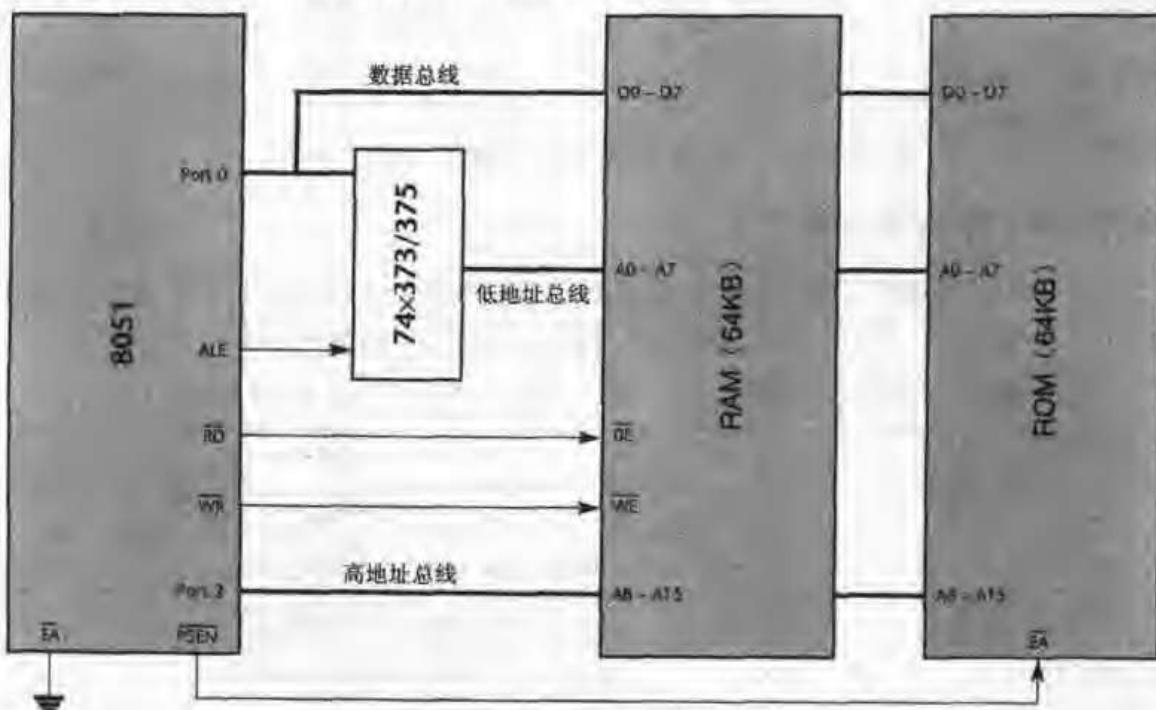


图 6.7 为 8051 设计扩展外部 ROM 和 RAM 存储器

例子：减少元件数量

如果空间很紧凑或者试图通过限制接线的数量来提高基于 8051 的系统的可靠性，有两个基本的选择：

- 使用有片内程序存储器的微控制器。
- 使用那些不需要锁存器的（ROM）存储器芯片。

第一个选择并不总是有效。例如，强大的 C509 微控制器运行飞快并有一些有用的特性（包括一个与 80C517 兼容的硬件数学单元），但没有片内 ROM。

当扩展不需要使用锁存器的外部存储器时，可以考虑使用 Atmel 的 AT27C520（64KB）EPROM。它有一个内部锁存器，因此不需要其他的元件。

例子：扩展多于 64KB 的程序存储器

除了一些最新的芯片（参见表 6.1），8051 系列的各种芯片仅仅直接支持最大 64KB 的外部程序存储器。这对于大型程序或需要使用大量只读变量的场合未必足够。在这种情况下，“分段切换”是一种选择。

表面上看，分段切换存储器方案的想法很简单。在一个简化了的方案中，系统有（例设）八段 64KB 的存储器。对每组存储器中的存储单元的访问由通常的 16 位地址线来控制。为了在八个存储块之间选择，使用微控制器上的八个附加的输出引脚来按照需要选择相应的存储段。

而实际情况稍显复杂。

首先考虑中断。当中断发生时，微控制器将跳转到当前存储段的某个地址。保证这个地址包含相关指令的一个方法是将中断函数复制到所有的存储段中（注意，正如将在后面看到的，编译程序通常能够自动地处理）。

同样，库函数也需要在各个地方都可以存取。因此通常将其放在一个公用区中。

保存在代码区的只读变量也需要被复制。除非能够保证这些常数将仅仅在某个存储段中调用，一般需要将它们放在一个公用区中。

最后，存储段切换的程序本身也必须在公用区中。

解决了这些问题之后，需要考虑当芯片复位时将会发生什么。此时，所有的端口输出均设置为 0xFFFF。如果某个端口正被用作分段切换，该端口的输出将控制哪个存储段的程序首先开始运行。需要通过存储段切换硬件来保证首先执行正确的存储段中的程序。

安全警告！

这些讨论表明，分段切换是复杂的而且容易出错。即使了解所做的一切，后来的系统维护人员未必能够充分理解如果改变程序时将产生的后果。

可能的话，应尽量避免分段切换。如果系统需要多于 64KB 的程序存储器，使用 Dallas 的 80C390、AD 公司的 ADμC812 或 Philips 的 8051MX（所有这些芯片都支持多于 64KB 的程序存储器）是更好的解决方案，也可以考虑第 3 章介绍的从 8051 升级的两种途径（使用 80251 或 8051XA）中的一种。

如果决定使用分段切换。首先，需要一个合适的地址解码机制，其次，需要一种实现公共代码区的方法。

有两种基本方法可用来解决“公用区”问题。最简单的方法是在每个程序存储段中都复制所有的公共代码。例如，图 6.8 显示了使用 4 个 64KB 的 ROM 芯片，每个之中复制了（大约 32KB 的）公用区。

也可以通过使用 5 个 32KB 的存储芯片实现类似的结果。排列为一个公用区和四个高地址的程序存储段（图 6.9）。

表面上看，大容量存储器芯片更有效率，其低成本意味着基于单存储器芯片的解决方案是经济合算的。此外，因为大多数本书给出的程序例子使用相对少的中断或库函数，所需的公用区常常长度较小（一般为几 KB），因此“复制公用区”解决方案比它最初看起来的效率要高。

例如，希望为 C509 微控制器扩展 512KB ROM 和 64KB RAM。大部分所需的硬件都是熟悉的：通过一个 74HC573 锁存器将一个外部 ROM 芯片和一个外部 RAM 芯片连接到微控制器上。这和前面图 6.7 给出的方案完全相同。

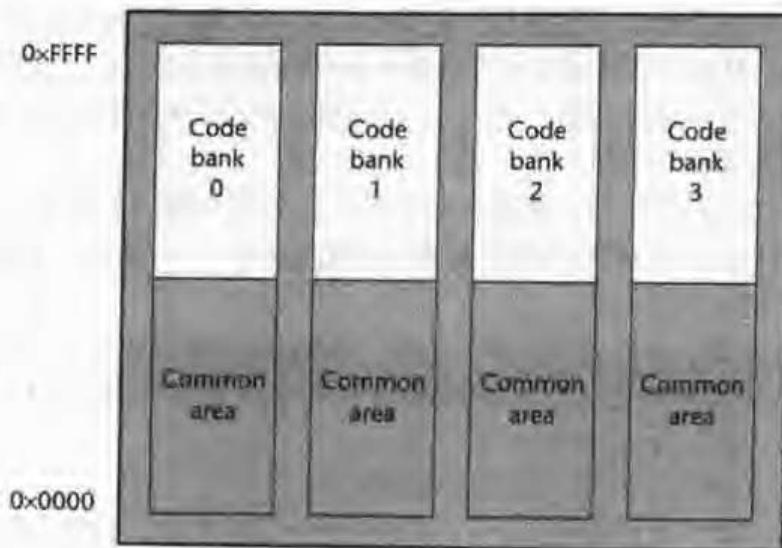


图 6.8 使用 4 个 64KB 的 ROM 芯片分段切换

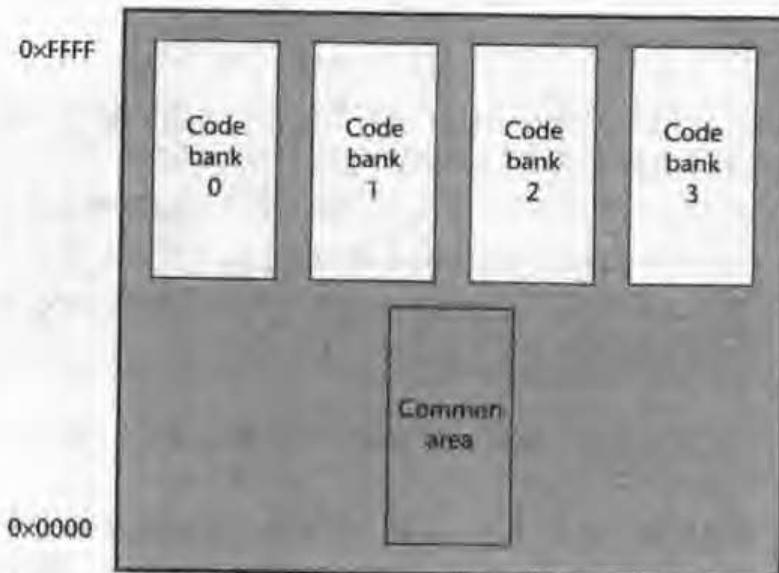


图 6.9 使用 5 个 32KB 的 ROM 芯片分段切换

当然，硬件的不同首先是在程序存储器的大小上，其次是 74HC257 的使用。74HC257 是一种（四路）2 选 1 数据多路复用器。它提供必要的“连接逻辑”连接端口引脚（可以使用任何可用的端口，在这个例子中是端口 3 的低 4 位）。

74HC257 连接的详细资料在图 6.10 和图 6.11 中给出。

注意，Keil 编译程序提供对分段切换的支持并且支持这里介绍的存储排列。详细资料参见 Banker/Linker 手册。

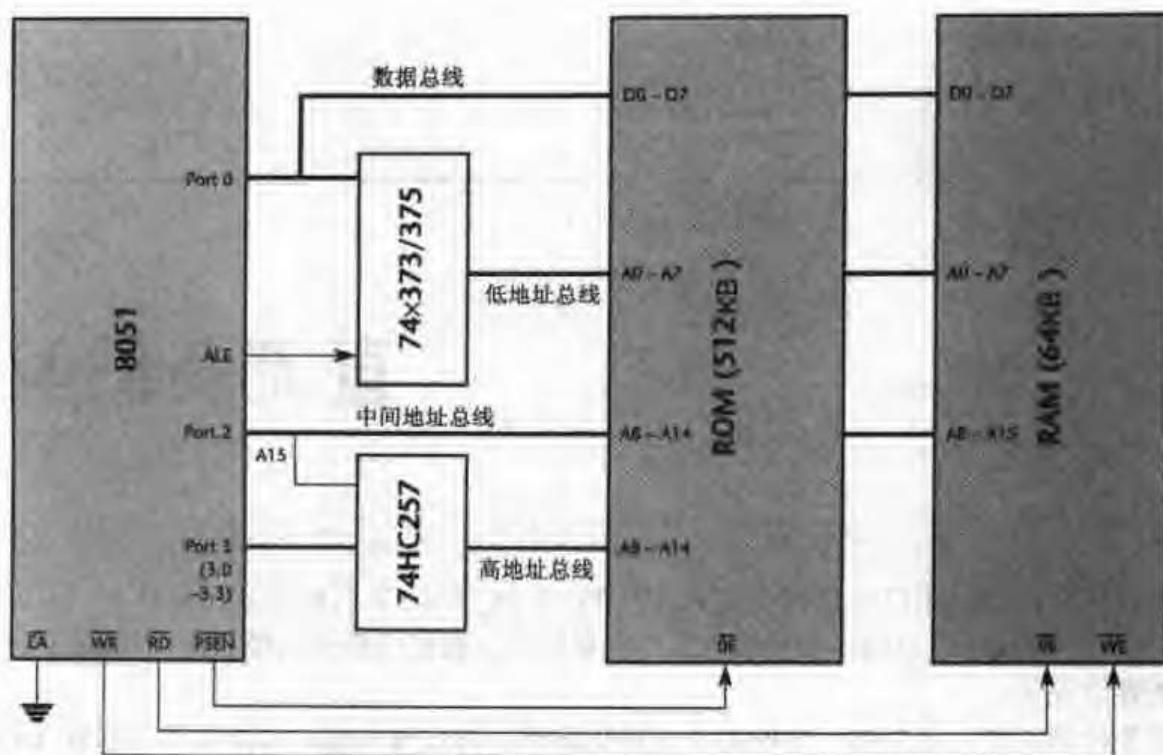


图 6.10 为 8051 设计扩展 512KB ROM

注意，这个例子中的硬件是由 Infineon 应用手册 AP0824 改编而来。

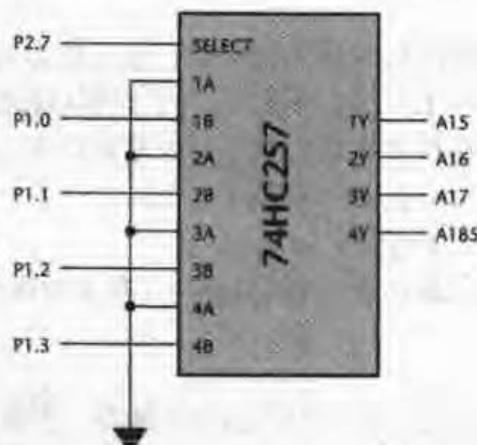


图 6.11 图 6.10 中的 74HC257 连接的详图

进阶阅读



直流负载驱动

引言

典型的微控制器端口可由软件控制置为 0V 或 5V（对 3V 系统是 0V 或 3V）。每个引脚一般可以灌入（或源出）10mA 左右的电流。本章将讨论通过这些引脚控制低功率或高功率直流负载的硬件设计。

需要注意的是，这些端口会被用来直接驱动低功率直流负载，例如发光二极管 LED（参见直接 LED 驱动）或小警报蜂鸣器（参见直接负载驱动）。当端口直接驱动多个这样的负荷，比如说 8 个 LED，通常就超过了微控制器的总端口驱动能力。在这种情况下，使用缓冲电路 IC 是一种划算的解决方法，参见 IC 缓冲器。实际上，即使是负荷较小的系统，通过缓冲器驱动也能改善其可靠性。

当然，许多输出装置都需要远超出微控制器端口输出能力的高电压和大功率驱动。例如，一个直流电动机的驱动就要求 12V/1A 的输出。为了控制这样的装置，需要用适当的驱动（或接线）电路将微控制器的输出转换到所需的水平。本章将研究三种驱动这类负荷的方法：

- 使用双极结型晶体管（参见 BJT 驱动）
- 使用驱动 IC（参见 IC 驱动器）
- 使用金属氧化物场效应晶体管（MOSFET）（参见金属氧化物半导体场效应晶体管驱动器）

请注意，本章主要关心的是接口电路的硬件细节。在端口输入输出中，将讨论控制硬件的相应软件。

直接 LED 驱动

适用场合

用 8051 系列的微控制器开发一种嵌入式系统。

需要为该系统设计相应的硬件。

问题

什么是用微控制器驱动少数几个 LED 的最便宜的方法？

背景知识

即使在最基本的系统中，一个一直发出红光或绿光的发光二极管（LED）也有让人安心的作用，至少表明系统已经上电了。其他需要数字显示的系统，可以使用成组的 7 段或 8 段 LED 显示^①。例如，显示当前的时间、电压及自动应答电话（语音邮件）系统接到的电话数目。总的说来，LED 是在嵌入系统中使用最广泛的用户界面元器件。

从 LED 的名字可以看出，在电气上，LED 像一个普通二极管一样工作。LED 有大约 2V 的正向压降，一般需要 5~15mA 电流才能达到高亮度显示（图 7.1）。注意，传统的硅二极管只要约 0.7V 的正向电压就能“接通”，这个差异是因为 LED 通常使用磷砷化镓制造而导致的（详情请参见 Horowitz and Hill, 1989）。

解决方案

这一方案的关键是低成本的 LED 驱动技术。

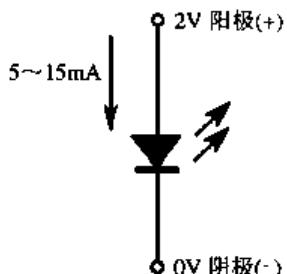


图 7.1 点亮单个 LED

驱动单个 LED

如图 7.2 所示，小的单个 LED 可以由微控制器端口直接驱动。注意，通常需要在电源和驱动端口之间用一个电阻和 LED 串联。当 LED 被接通时，这个电阻用于限制流进驱动端口的电流。

为了理解为什么这个电阻是必需的，需要知道，正如在“背景知识”一节中所讨论的，LED 上的电压降大约是 2V。假定 Vcc 是 5V，当驱动引脚的输出电压是 0V 的时候，电阻和 LED 上的电压降大约是 5V。这样，电阻上的电压降应该是 3V。如果没有这个电阻，3V 电压就将完全降落在一段电线上：从而导致产生非常大的电流（仅限制于电源容量），可能会立刻烧坏驱动端口、LED，甚至是整个微控制器。

^① 多段式 LED 将在 MX LED 显示中讲述。

图 7.2 中的等式（本质上是欧姆定律）表明了如何计算所需的电阻（R）阻值。所需参数的典型值如下：

- 电源电压， $V_{CC} = 5V$
 - LED 正向压降， $V_{diode} = 2V$
 - 所需 LED 电流， $I_{diode} = 10mA$ （注意，所选用的 LED 的数据手册会提供这个电流）
- 以上条件的计算结果是需要 300Ω 的电阻。

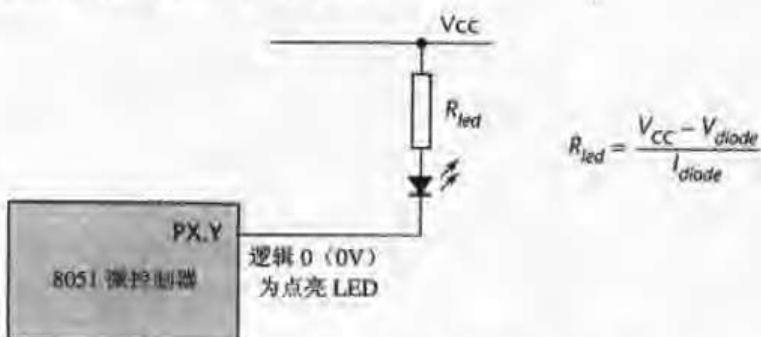


图 7.2 将单个 LED 直接连接至微处理器的端口

注意，当计算所需的 R_{led} 阻值时，电阻的单位是欧姆，电压的单位是伏特，电流的单位是安培。

上拉电阻的使用

在本章中，将在多处介绍如图 7.2 所示的驱动电路例子。

这个电路在微控制器内部有上拉电阻的端口引脚上才能正确工作，这一点对 8051 系列微控制器的大多数端口都适用，除了端口 0。另外，其他一些 8051 系列的微控制器（特别是 Atmel 的 89Cx051 系列）有少数引脚没有内部上拉电阻。

改造图 7.2 所示的电路以用于没有内部上拉电阻的端口非常简单：只需要增加一个外部上拉电阻，如图 7.3 所示——上拉电阻的阻值应该在 $1K\sim10K$ 间。这适用于本书中的所有例子。

在不知道驱动电路的端口是否内含上拉电阻时，最好的办法是加上一个 $10K$ 的上拉电阻。如果用于驱动的引脚已经内含上拉电阻，外加的上拉电阻不会对电路的正常运作产生什么影响。

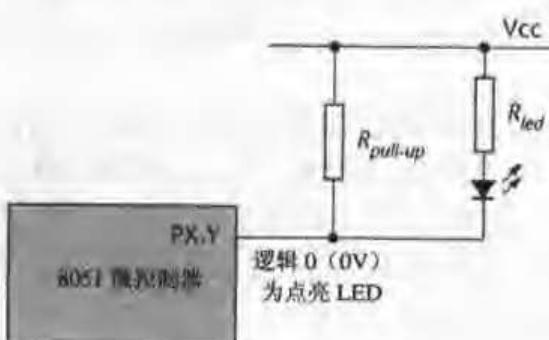


图 7.3 使用上拉电阻

硬件资源

这种模式的每个实现方案都至少需要一个端口引脚。

可靠性和安全性

使用这种模式涉及几个可靠性方面的要求。

LED 连接

如果将普通的 LED 连接至端口，切勿遗漏了电阻！省略电阻而导致的大电流也许不会使系统立刻失效，但将大大减少驱动端口的寿命，而且很可能减少整个处理器的寿命。

注意：现在已经有 5V 的 LED 可用了，但成本更高，这种 LED 内含了串联电阻。如果这种 LED 的驱动电流负荷要求，则可以安全地直接连接在端口引脚上使用。

是否应该使用缓冲器？

当超过 2 个的 LED 连接至同一端口时，缓冲几乎总是必需的。这是因为，在每个端口引脚的电流驱动能力是 10mA 的情况下，整个端口的总电流驱动能力可能只有 20mA 或者更小。这个问题将在 IC 缓冲放大器中讨论。

用 LED 作为报警器

LED（特别是闪烁的 LED）经常用作报警器。记住，在明亮的日光下，这样的警示几乎看不见，而且盲人或者有视力缺陷的人根本看不到。在一些系统中，增加附加的发声装置或者用发声装置替换 LED 是合理的。

LED 的一般用途

LED 消耗大量的电能（例如和液晶显示相比较），在许多电池供电的设计中必须小心使用。

可移植性

本模式中介绍的电路几乎能用于所有的微控制器和微处理器。然而请注意，如图 7.2 所示，本章介绍的大部分直流和交流负载驱动电路用的都是某种形式的“灌”电流设计。在这些电路中，电流流进处理器的驱动端口。这种驱动方式并没有考虑大多数微控制器的端口都是用某种形式的 MOS 工艺制造的，也能够“源”电流（提供电流），所以负载也可以用图 7.4 所示的方法驱动。

然而，一些其他用作缓冲或驱动的电路可能使用了不同的制造工艺，包括 TTL 工艺。TTL 器件的灌电流能力和 MOS 器件差不多，但是“源”电流能力却相差很多。所以，基于灌电流的硬件设计比基于源电流的硬件设计具有更好的可移植性（在不同的逻辑器件系列间）。因此，灌电流的硬件设计将用于本书的各种驱动模式中。

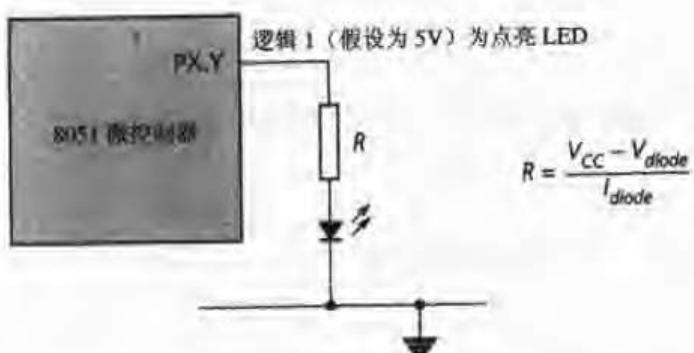


图 7.4 将一个 LED 直接连至处理器的端口

注意，在这里，端口作为一个电流源。这种驱动方式的详情及对其缺点的讨论参见上文。

优缺点小结

- ① 这一模式允许通过微控制器的端口驱动少数几个 LED，只使用最少的外部硬件。
- ② 这是一个低成本的解决方案。
- ③ 只适用于驱动少数几个 LED（典型情况是 2 个），驱动更多的 LED 必须使用缓冲器，见 IC 缓冲器。

相关的模式和替代解决方案

所需的软件详见端口 I/O。适用于驱动更高功率负荷的技术参见本章后续的其他模式。

例子：使用低电流 LED

为了强调一下所有的规则都可能被打破，在这个例子中将不使用缓冲器，而把 8 个 LED 直接连接至 8051 的端口（见图 7.5）。当然，正如已经讨论过的，不可能用这种方式连接这么多“普通”(10mA) 的 LED 而又不超过端口的驱动能力。然而，如果使用低电流 (2mA) LED，对大多数 8051 微控制器而言却是可能的。所需电阻的阻值计算如下：

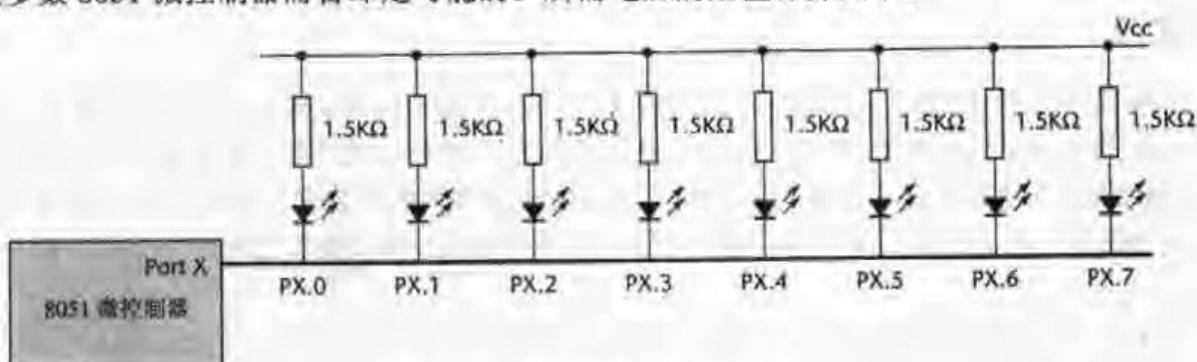


图 7.5 使用低电流直接驱动 LED

$$R_{led} = \frac{V_{cc} - V_{diode}}{I_{diode}} = \frac{5V - 2V}{0.002A} = 1.5k\Omega$$

注意，天下没有免费的午餐，2mA 的 LED 不会像 10mA 或者 20mA 的 LED 那样亮。

进阶阅读

Horowitz, P. and Hill, W. (1989) *The Art of Electronics*, 2nd edn, Cambridge University Press, Cambridge, UK

直接负载驱动

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

对于这样的系统，怎样把一个低电压（不大于 5V），低功率（不大于 100mW）的直流设备连接到 8051 微处理器的端口上？

背景知识

背景材料参见直接 LED 驱动。

解决方案

在直接 LED 驱动一节中，考虑了如何把 LED 连接至 8051 微控制器的端口。在这个模式中，将考虑更一般的情况——如何控制一般的小直流负载（图 7.6）。

图中所示为连接至端口的一般（非特定的）负载。

一般负载和 LED 的主要区别是负载上的电压降（典型地）是 5V（指在 5V 系统中，3V 系统中为 3V）。

和 LED 的 2V 压降相比，在这种情况下，电阻可以省略，因为所需的电阻阻值为：

$$R = \left[\frac{5.0V - 5.0V}{I_{load}} \right] = 0\Omega$$

硬件资源

这种模式的实现方案至少需要一个端口引脚。

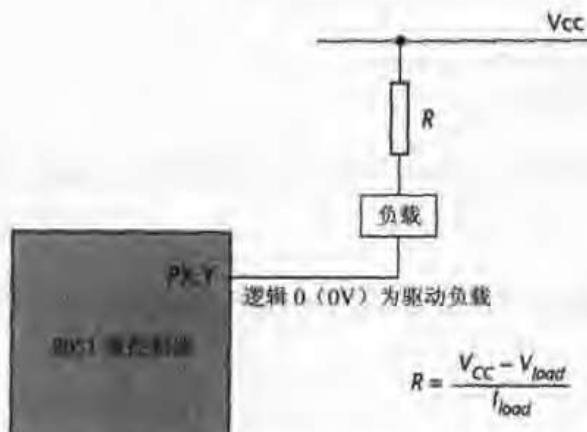


图 7.6 不使用缓冲器的低功率负载驱动

可靠性和安全性

IC 缓冲器讨论了缓冲驱动 LED 和其他小负载的技术，它们能增加一些系统的可靠性。

可移植性

所有微控制器都可以用这种方式控制小的负荷。

优缺点小结

- ① 本模式讨论的技术可以通过微处理器的端口用最少的外部硬件实现小负载的驱动。
- ② 这是个低成本的解决方案。
- ③ 只适用于小负载（和少数几个非常小的负载），否则缓冲器是必需的，参见 IC 缓冲器。

相关的模式和替代解决方案

相应的软件详见端口 I/O。

适用于从单个微处理器驱动 2 个或者更多小负载的技术参见 IC 缓冲器模式。

适用于驱动高功率直流负载的技术参见 BJT (双极结型晶体管) 驱动模式、IC 驱动和 MOSFET (金属氧化物场效应管) 驱动。

例子：蜂鸣器

某些压电蜂鸣器可在微控制器端口提供的电压（蜂鸣器可在 3~12V 间工作）和电流（蜂鸣器需要 3mA 的电流）下产生非常响的声音（70dB）。这一特性可用于制造非常好的告警设备，甚至是在电池供电的设备上（见图 7.7）。

正如在“解决方案”一节中讨论过的，这个 5V 负载不需要串联电阻。

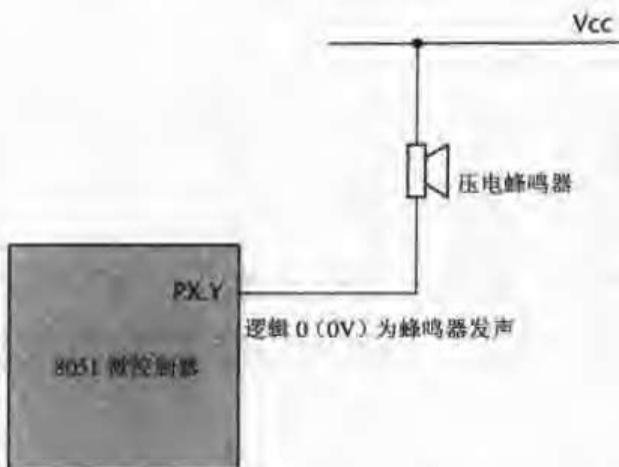


图 7.7 将一个蜂鸣器直接连至微处理器端口

进阶阅读

Horowitz, P. and Hill, W. (1989) *The Art of Electronics*, 2nd edn, Cambridge University Press, Cambridge, UK

IC 缓冲放大器

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

如何安全地从单个微控制器的端口控制一个或者多个小功率直流负载？

背景知识

假定用直接 LED 驱动模式中讨论的技术从一个端口控制 8 个 10 mA 的 LED。图 7.8 介绍了一种驱动的方法。

但是在几乎所有的情况下，这种方法都会失败。正如在直接 LED 驱动中讨论的，8051 系列的微控制器一般每个端口引脚都可以吸收或提供大约 10mA 的电流。但图 7.8 并没有交代清楚所有的情况。以一个 8051 系列的微控制器为例，Atmel 89C52 是一款“标准”的现代 8051 微控制器（40 个引脚、4 个端口），在本书中用于各种各样的例子。它的每个端口最大可以吸收 10mA 的电流。可是，P0 端口总共只能吸收 26mA 的电流（P0 端口所有引脚的总和），而 P1、P2、P3 每个端口总共能吸收 15mA 的电流。总的说来，整个芯片至多只能吸收

71mA 电流。

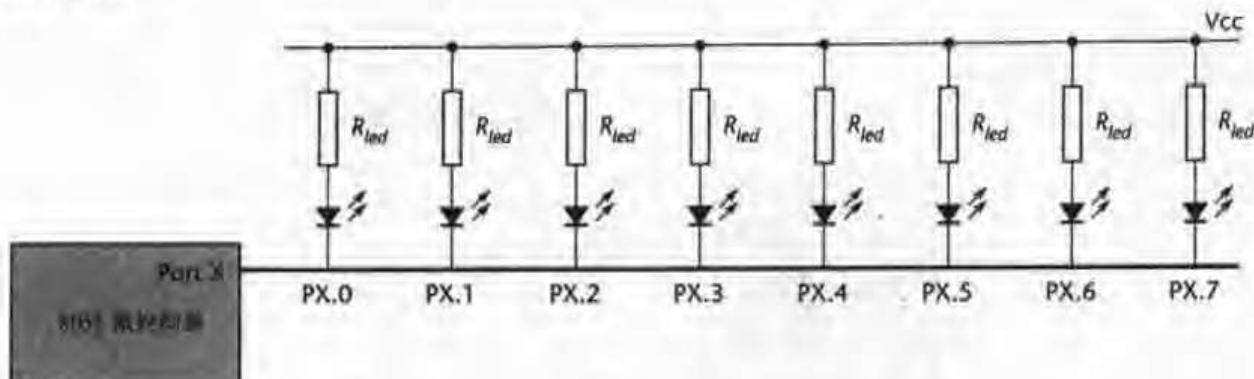


图 7.8 试图用一个微控制器端口直接驱动 8 个 LED

注意，大多数情况下，除非使用低电流 ($\sim 1\text{mA}$) LED，这种方法都将损坏端口（和微控制器）。

虽然各种 8051 系列微控制器之间有所不同，但 89C52 代表了多数微控制器的情况。如果图 7.8 使用的是普通的 LED，则其总电流大约为 80mA，超出了一般 8051 微控制器的驱动能力。因此，图 7.8 中的电路通常是不能使用的。

解决方案

和“背景知识”中论述的一样，如果需要从一个微控制器驱动多个小功率负载，缓冲器是必需的。即使端口能直接驱动这些负载，也可以通过在微控制器和负载间使用一片 IC 缓冲器，降低端口受损的危险和使设计更加可靠。

本模式将讨论，第一，各种可能被用作缓冲放大器的 IC；第二，缓冲放大器输出端是否需要上拉电阻。

寻找合适的 IC

许多 IC 都能被用作缓冲放大器。合适的非门 IC 非常容易得到，而且价格低廉。例如，使用非常普遍的 74x04，它包含 6 个封装在一起的非门（图 7.9）。

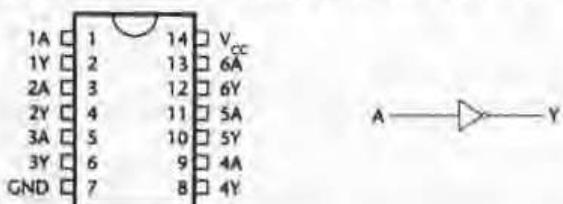


图 7.9 74x04 非门的管脚图和对应的逻辑图（正逻辑）（承蒙德克萨斯仪器允许转载）

注意，每个芯片包含 6 个独立的非门。

除 74x04 外的另一个选择是取反的缓冲放大器(74x240)和不取反的缓冲放大器(74x241)，8 个缓冲放大器封装在一个芯片内（参见图 7.10 和图 7.11）。这两个芯片对于缓冲整个端口非

常有用。

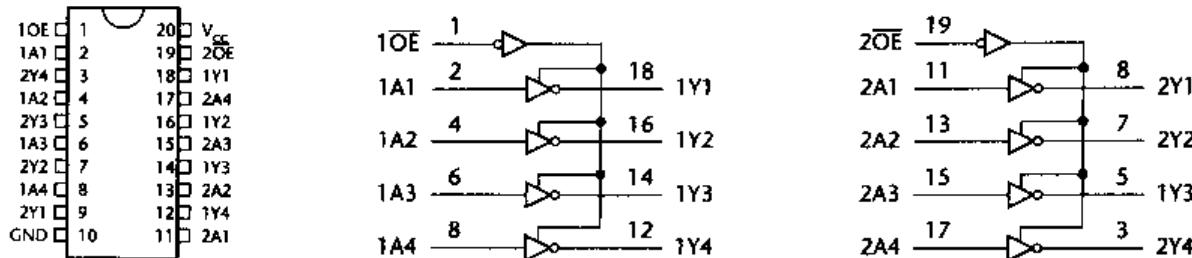


图 7.10 74x240 反相缓冲器的引脚图和逻辑图（承蒙德克萨斯仪器允许转载）

注意，每个芯片内含 8 个缓冲放大器，分成两组，每组四个（第一组，第二组）。所有的缓冲放大器都是三态器件。仅作为缓冲放大器使用时，必须通过在门 1 上加低电压（0V）使能（就第一组来说），或者在门 2 上加高电压（5V）使能（就第二组来说）。

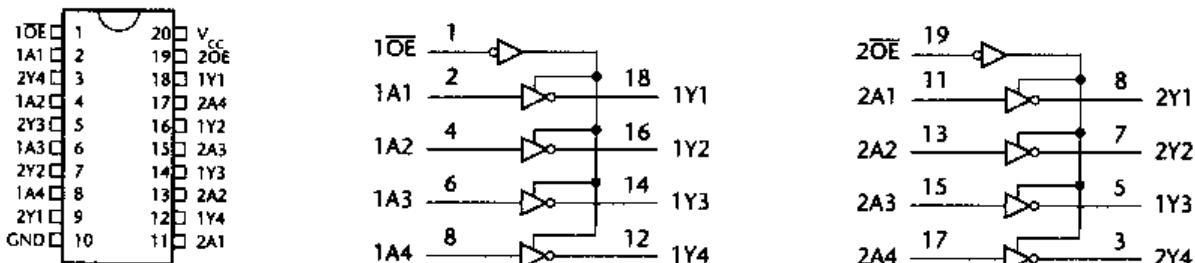


图 7.11 不反相缓冲器 74x241 和 74x244 的引脚引线（承蒙德克萨斯仪器允许转载）

注意，和 240 一样，每个芯片包含 8 个缓冲放大器，分成两组（第一组，第二组）。所有的缓冲放大器都是三态器件。仅作为缓冲放大器使用时，必须通过在门 1 上加低电压（0V）使能（就第一组来说），或者在门 2 上加高电压（5V）使能（就第二组来说）。

注意，所有的缓冲放大器都属于 74 系列的 IC。这些 74x 缓冲放大器每个引脚可以承受 20mA 的电流。240/241/244 的总驱动电流是 70mA，04 的总驱动电流是 50mA。如果负载满足每个引脚 20mA 的条件而不满足每个芯片总驱动电流的条件，则需要使用多个缓冲芯片。

这些缓冲放大器操作相当快，最大延迟时间大约为 1 微秒。

逻辑芯片系列

上文讲述的 74 系列缓冲放大器有数目众多的版本，包括“LS”（74LS04 等），“ALS”、“HC”、“HCT”、“VHCT”、“AHC”及“AHCT”等等。不同的版本有不同的开关速度、功耗、工作电压和价格。通常，可以选择任何能符合项目需要的缓冲放大器。

尽管 74 系列的缓冲放大器有众多的可选器件，但实际上只有两个逻辑芯片系列：

CMOS（互补金属氧化物半导体）和 TTL（晶体管-晶体管逻辑电路）。CMOS 器件通常在器件名称上带字母“C”（例如，74HC04），TTL 器件通常在器件名称上带字母“S”（例如，74ALS04）。下面介绍一下这两种器件的差别。

最初的 5V TTL 器件可以追溯到大约 1964 年。时至今日，TTL 器件已经历许多的改进，并且仍然被广泛使用。TTL 器件的主要优点是快速，主要缺点是相对较大的功耗。

TTL 的主要竞争者是 CMOS 器件。5V CMOS 器件的历史可以追溯到大约 1983 年，也经

过了许多的改进。CMOS 器件的主要优点是低功耗和速度已经大大地改善了（对较新的器件而言），看来也许 CMOS 逻辑器件将在今后几年代替 TTL 逻辑器件。

任何 8051 微控制器驱动 TTL 或 CMOS 逻辑器件都没有问题。但是，根据使用的技术不同，缓冲放大器的输出有重要的区别：

- 对(5V)TTL 而言，逻辑 0 输出电压范围是 0V~1.5V；逻辑 1 输出电压范围是 3.5V~5V。
- 对(5V)CMOS 器件而言，逻辑 0 输出电压是~0V；逻辑 1 输出电压是~5V。

这个差异有重要的意义。例如，考虑一下使用 CMOS 逻辑门缓冲放大 LED 输出的情况（图 7.12）。这种驱动方法可以有效地工作，因为电压摆幅足够大而且固定；可是，考虑一下将缓冲放大器换成 TTL 器件的情形（图 7.13）。

在图 7.13 中，指出了 TTL 缓冲放大器的两个缺点。第一，需要一个上拉电阻（通常阻值为 5K~10K）将输出“高”拉到 5V。第二，输出“低”在 0V~1.5V 的范围间变化。这样，很难选择适当的电阻阻值同时保证明亮的显示和不超过缓冲放大器或者 LED 的电流限值。

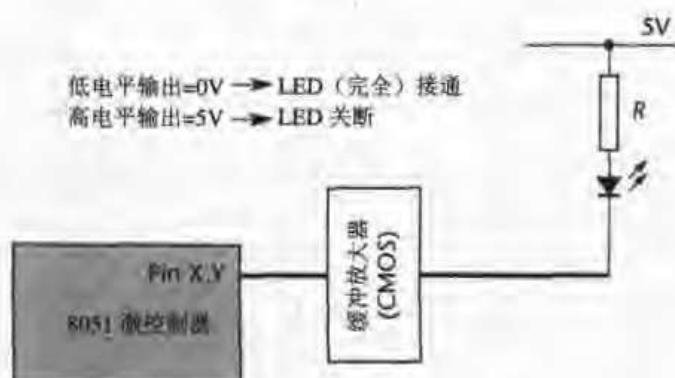


图 7.12 使用 CMOS 缓冲放大器

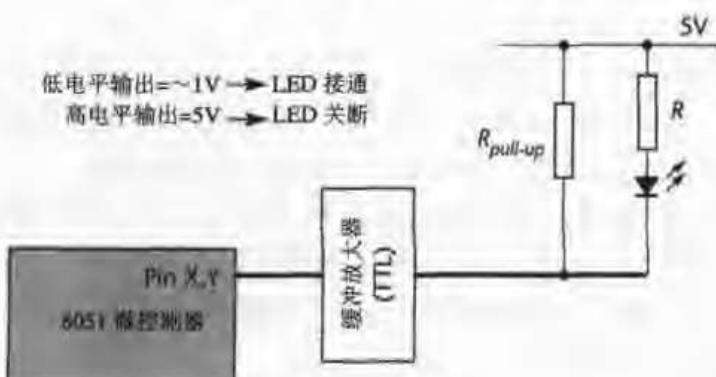


图 7.13 使用 TTL 缓冲放大器

使用 CMOS 缓冲放大器

正如上文中的讨论所建议的，只要有可能应该尽量在缓冲放大器设计中使用 CMOS 逻辑器件。同时也应该在设计文档中讲清楚，使用的是 CMOS 逻辑。

如果电路是别人设计的，且设计中在缓冲器的输出端使用了上拉电阻，则可以认为该设计使用的是 TTL 逻辑。当然，也可能使用的是 CMOS 逻辑；如果没有上拉电阻，使用的应该是 CMOS 逻辑。

在缓冲放大器的输入端使用上拉电阻

照例，如果端口的引脚没有内部的上拉电阻，则无论使用哪种逻辑，都需要在 IC 缓冲放大器输入端加入上拉电阻（10K 即可），详情请参见直接 LED 驱动。

硬件资源

这种模式的每个实现方案都至少需要一个端口引脚。

可靠性和安全性

驱动小输出功率设备需要做出的一个关键设计决定是是否使用缓冲放大器。

在一些情况下，省略缓冲放大器会潜在地降低系统的安全性。例如，如果嵌入式系统在面板上有一能被接触到的 LED，而有人希望与系统交互，则可能导致 LED 上产生高电压而损坏。如果 LED 直接连至微控制器的端口引脚，甚至有可能造成微控制器的损坏。如果 LED 和微控制器之间有缓冲放大器，微控制器损坏的可能性将很小。

使用缓冲放大器会增加生产成本。然而，如果产品在使用时输出装置可能会损坏，增加一个缓冲放大器确是个好的选择。一般情况下，更换一个烧坏的缓冲放大器（0.10 美元）总比更换一个烧坏的微控制器便宜（1 美元）。因此，对于考虑了维修的小批量或高成本的产品，在设计中使用缓冲放大器是个好的解决方案。

总的说来，如果可靠性比较重要，则应该使用缓冲放大器。否则，对极低价（或大批量生产）的产品，或者修理不太可能的情况，使用缓冲放大器只会增加生产成本。

可移植性

这种技术对所有的 8051 系列微控制器都是可行的（还包括大多数其他微控制器和微处理器系列）。

照例，如果用于没有内部上拉电阻的端口引脚，则需要在设计中加入这样的上拉电阻（10K 即可）；详情参见直接 LED 驱动。

优缺点小结

- ◎ IC 缓冲放大器允许从单个端口控制多个小负载。

- ⑤ 缓冲放大器可以改善可靠性。
- ⑥ 使用缓冲放大器增加了产品成本。

相关的模式和替代方案

本章的另一些模式（特别是 IC 驱动器）提供了替代方案。

例子：用一个 74HC04 缓冲驱动三个 LED

图 7.14，用 74HC04 缓冲驱动三个 LED。和在“解决方案”中讨论的一样，HC (CMOS) 缓冲放大器并不需要上拉电阻。

在这种情况下，假定这些 LED 的驱动电流是 15mA，不超过缓冲放大器的驱动能力（总共 50mA）。

所需的电阻阻值是：

$$R_{led} = \frac{V_{cc} - V_{diode}}{I_{diode}} = \frac{5V - 2V}{0.015A} = 200\Omega$$

软件部分请参阅端口 I/O。

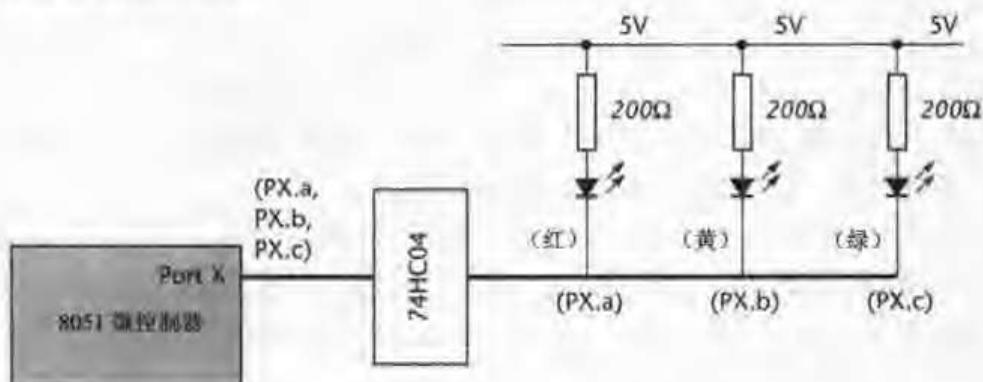


图 7.14 用 74HC04 非门驱动三个 LED

进阶阅读

BJT (双极结型三极管) 驱动器

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

怎样从 8051 微控制器的端口驱动需要 2A 电流的直流负载？

背景知识

双极结型晶体管（BJT）是 20 世纪 50 年代初发展出的一种三端半导体器件。考虑到和本章末将提到的 FET（场效应晶体管）相区别，将使用这个全名（或简称）。

就本书所关心的范围而言，双极结型晶体管最常用于作为电流控制开关。本书不关心 BJT 的内部运作细节，而将集中论述如何使用 BJT 作为微控制器驱动直流负载的手段。

解决方法

BJT 晶体管有两种基本规格：PNP 和 NPN（图 7.15）。

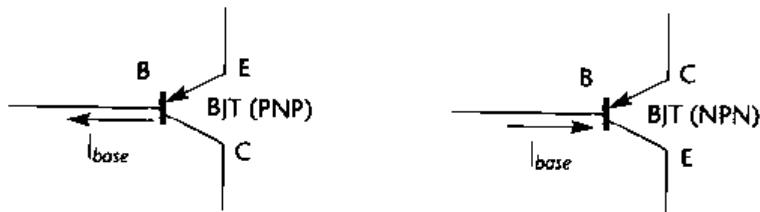


图 7.15 两种规格的 BJT [左] PNP, [右] NPN

注意，对于本书，BJT 的主要应用价值是作为受电压控制的开关使用。对于 PNP 器件，流出电流基极的电流用于控制大得多的流过负载的电流。对于 NPN 器件，流进基极的电流完成的也是同样的功能。

为了“接通”晶体管，需要根据下面的公式向 BJT 的基极提供电流（对于 NPN）或者吸收电流（对于 PNP）：

$$I_{base} \geq \frac{I_{collector}}{h_{fe}}$$

其中 h_{fe} 是晶体管的放大倍数。其数值大约在 15~800 之间。书中使用的一般是放大倍数为 100 的器件。前面提到过，8051 的引脚可以吸收或者提供大约 20mA 的电流（最大）。这样，使用合适 BJT 就可以控制最大约为 2A 的电流（20mA 乘以 h_{fe} ）。

这里将给出两个具体的例子，一个 PNP 晶体管开关和一个 NPN 晶体管开关。

PNP 晶体管开关

为了说明将在这个模式中论述的 BJT 应用方法，考虑图 7.16 的情况。这是一个小灯（5V, 500mA），连接在一个 PNP 晶体管上。注意，在这个电路中，几乎任何 PNP 晶体管都能工作，只要有大约 100 的 h_{fe} 值（在数据手册中可以查到），而且能够承受大约 500mA 的集电极电流。在这个例子中，使用的是 Zetex 的 ZTX 751 晶体管。这个晶体管可以控制最大 2A 的集电极电流，能适用于各类系统。这类器件并不昂贵，每个大约 0.30 美元（购买 1000

只的单价)。

简单地说，这个电路是这样工作的：当微控制器引脚是逻辑0时，电流将流出晶体管的基极，使器件进入饱和态。所需的负载电流流动起来，灯就被点亮了；当微控制器引脚处于逻辑1时，没有电流流出晶体管的基极，负载电流(基本上)将是0。

总的说来，这个电路的关键之处是用非常小的流出晶体管基极的电流(图中的 I_{base})控制比其大得多的流过灯的电流(I_{load})。这是所有BJT接口电路的基础。

上拉电阻

使用这个模式时，可能需要在硬件设计中加入上拉电阻。(参见直接LED驱动)。

输出端缓冲放大

使用BJT可以控制相对较大功率的负载，而负载可能发生损坏，例如，器件短路、电动机堵转或者整个器件物理损坏。在这种情况下，微控制器的端口就会被高电压和/或大电流损坏，可能导致整个微控制器彻底不能工作。

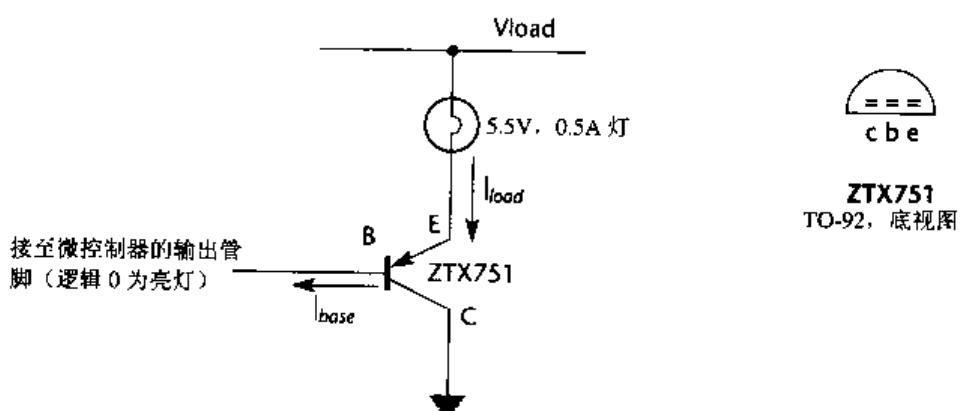


图 7.16 PNP 晶体管驱动一个灯，灯由 8051 微控制器控制

注意，上拉电阻详见直接LED驱动，这里可能也会用到上拉电阻。注意，灯的供电电压在这里假定为5V。实际上，负载的供电电压不必和微控制器的电源电压一样。

在许多情况下，万一负载或者BJT驱动器出现了问题，如果使用缓冲放大器保护，微控制器就会比较安全。图7.17展示了一个合适的电路。注意，其中使用了两个非门以达到正确的逻辑。一片74x04上有多个门电路，而且其缓冲放大器的成本很低，所以这是个不错的设计方案。也可以用图7.18中基于74x241的电路作为替代方案。注意，因为在IC缓冲放大器一节中已经讨论过的原因，在使用TTL技术时，这两种缓冲电路与晶体管基极的连接处都可能需要加上一个上拉电阻。

NPN型晶体管开关

利用NPN型晶体管可以设计出一个类似于图7.17的电路(图7.19)。

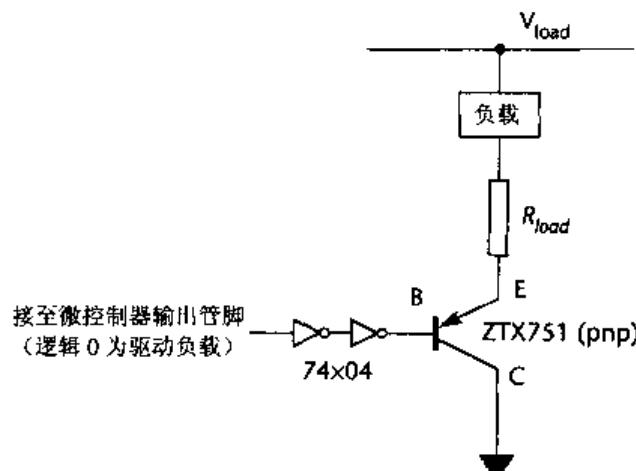


图 7.17 在微控制器和 BJT 驱动器间加入 74x04 缓冲放大器

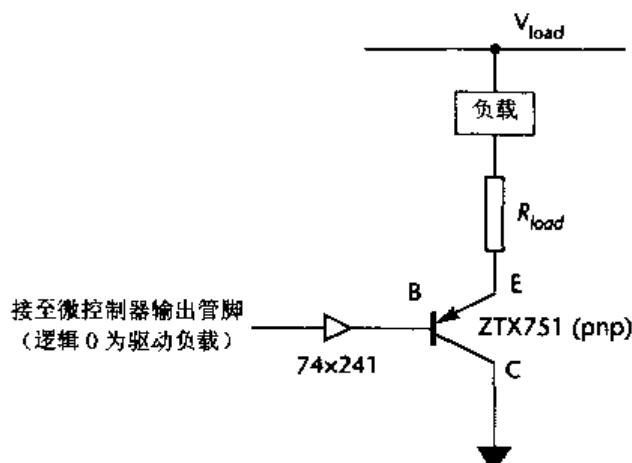


图 7.18 在微控制器和 BJT 驱动器间加入 74x241 缓冲放大器

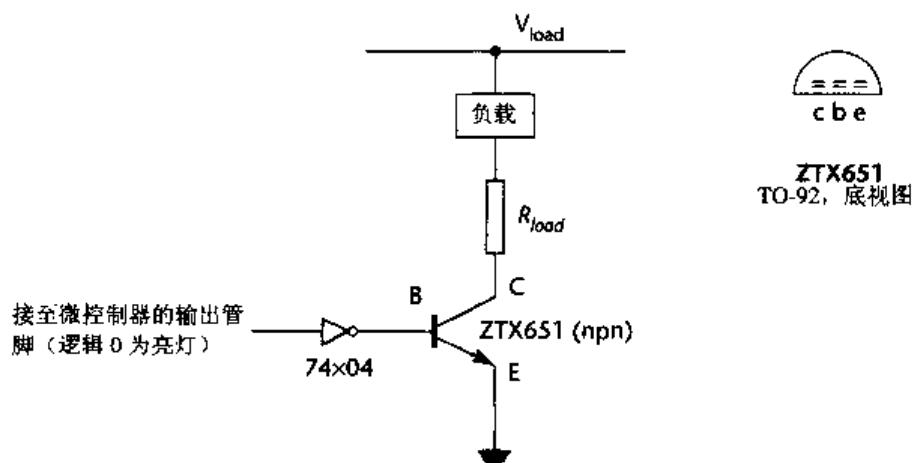


图 7.19 从 8051 的端口驱动 NPN 型晶体管的一种方法

注意，图中假定使用的是 CMOS 缓冲放大器，参见 IC 缓冲放大器。

在图 7.19 的电路中，使用了非门将端口的输出反相（所以现在要用逻辑 0 驱动负载），并在晶体管和微控制器之间提供一定程度的隔离。

注意：在图 7.19 中需要“提供”电流，也就是说，电流方向是流进晶体管的基极。在直接负载驱动中已讨论过，在一些情况下，源电流的要求可能会导致问题。因为这个原因，基于 PNP 晶体管的驱动器的可移植性略偏好些。

硬件资源

这种模式的每个实现方案都至少需要一个端口引脚。

可靠性和安全性

引脚复位后的初始电平

系统复位后，许多端口特殊函数寄存器（SFR）被设为 0xFF。这对安全性和可靠性非常重要。

设想一下，如果一个电动设备连接到端口，而这个设备是用逻辑 T 输出启动的，即由端口引脚上的 T 输出启动。当微控制器复位时，电动设备可能被启动并开始移动。即使端口的输出在程序一开始就被设为 0，电动机在程序执行到停止指令前仍可能被瞬间驱动。在一些系统中，这可能导致系统用户或者靠近系统的人员受伤甚至死亡。

因为输出引脚复位后为高，所以必须确保有安全性要求的设备连至微控制器时是低电平有效的，即相应端口引脚输出为 0 时启动设备。

接通灯和直流电动机

本章介绍了许多公式，用于计算如图 7.19 所示的驱动电路中的串联电阻阻值，在介绍这些等式时，我们假定负载阻值是固定不变的。

事实上，并不是所有的设备都表现为电阻性负载。例如，当控制灯或者直流电动机时，电路接通的初始电流非常大。这个冲击电流可能持续数百毫秒，才逐渐降低到稳态值。驱动电路要能够承受接通瞬间的冲击电流。

处理冲击电流的一种方法是提高驱动电路的额定设计参数。这意味着，为了承受冲击电流，稳态电流为 1A 的灯或者电动机的驱动电路的驱动能力应达到 10A 甚至更高。请注意，一般来说无法估测冲击电流的大小。请查对数据手册——一般其中都提供了相关信息。如果找不到精确的资料，应假定冲击电流至少为稳态电流的 10 倍。

解决这个问题的另一个方法是使用所谓的热敏电阻和负载相串联（图 7.20）。这种电阻的冷态阻值大约为 $1\sim2\Omega$ ；当热敏电阻变热时（当电流流过热敏电阻时），阻值降到大约 0Ω 。即使小的初始电阻也能大大地减小冲击电流。

请注意一个经常被忽视的问题。并不是所有的驱动电路在过大的负载下都会立刻失效，在测试时驱动电路可能可以可靠地工作。可是，让驱动电路超过其额定值工作将大大缩短它的使用年限，导致驱动电路在现场应用中失效。如果有任何疑问，则应选择功率足够大的二极管或电阻。

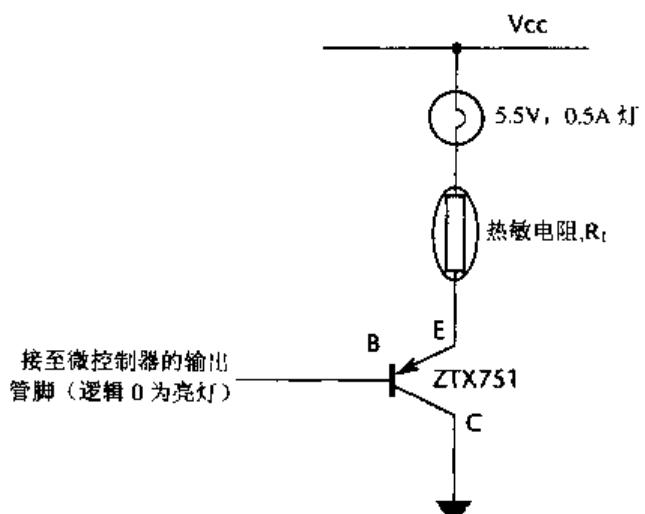


图 7.20 使用热敏电阻防止冲击电流导致的损坏

切断电感性直流负载

如果负载是感性的，安全地接通负载并不意味着所有的问题都得到了解决。

所有包含线圈的负载都是电感负载，经常遇到的例子就是电磁式继电器和电动机。切断这种负载必须非常小心，因为如果电流消失，电感两端的电压将急速上升。电感切断引起的电压上升遵循下面的公式：

$$V = L \frac{dI}{dt}$$

这里， V 是电感两端的电压， L 是电抗大小。 dI/dt 是电流的变化率。如果突然切断电流（也就是说，突然切断电感），急剧变化的电流将转化为电压的急剧上升。这种“感应冲击”足以损坏逻辑门电路或者任何其他形式的负载驱动电路。为了保护这个电路，可用一个二极管吸收“感应冲击”（图 7.21）。

请注意，二极管不仅要能承受稳态电流，而且要能承受断开时的感应电流——普遍使用的廉价二极管 1N4004 可以承受 1A 的电流，适合于许多电路设计。如果 1N4004 不够大，有各种其他容量的二极管可供选择，例如，1N5401 (3A, 100V)、40HF10 (40A, 100V) 或者 70UR60 (250A, 600V)。注意，价格随载流量的增加大幅上升，低功率的二极管只值几美分，而大功率的二极管的价格可能高达 10 美元。

注意，如果电流在被切断后迅速衰减，二极管可用电阻代替（图 7.22）。在切断时，电阻

两端的电压如下：

$$V_{resistor} = I_{turnoff} \times R$$

切断负载时，三极管（或驱动电路）两端的电压为：

$$V_{transistor} = I_{cc} \times V_{resistor}$$

必须选择合适的电阻阻值，以保证 $V_{transistor}$ 不超过允许的限值。

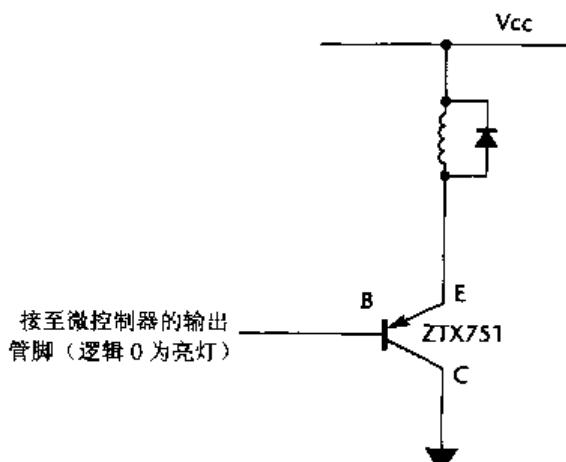


图 7.21 用二极管保护晶体管免受感应冲击

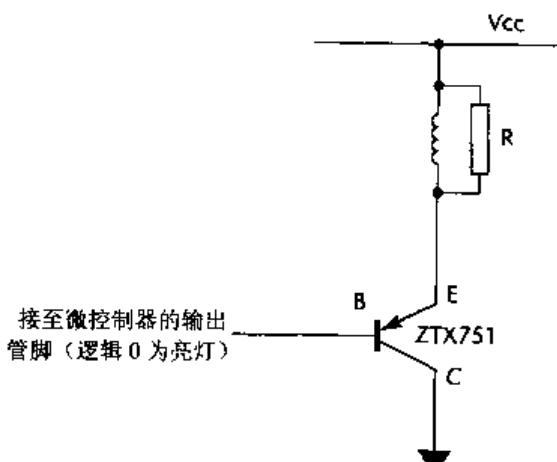


图 7.22 用电阻保护晶体管免受感应冲击

请注意一个经常被忽视的问题。并不是所有的驱动电路在过大的负载下都会立刻失效，在测试时，驱动电路可能可以可靠地工作。可是，让驱动电路超过其额定值工作将大大缩短它的使用年限，从而导致驱动电路在现场应用中失效。如果有任何疑问，则应选择功率足够大的二极管或电阻。

负载故障时的保护措施

热敏电阻、二极管和涌流抑制电阻是设计为在正常情况下保护开关电路。在驱动大功率的

负载时，也需要处理负载损坏引起的问题。特别是需要考虑负载短路的可能性。

这个问题在半导体开关装置（例如 BJT）上特别容易发生。半导体器件的热容量特别低，这意味着过大的电流（例如，由短路或者堵转的电动机引起的大电流）将使半导体器件温度飞快上升，超出安全工作范围（参见 Lander, 1993；或者 Rashid, 1993）。为了在这种情况下保护电路，必须使用保险丝和负载相串联。

注意“普通的”保险丝是不敷使用的，保险丝必须迅速地熔断以防半导体器件损坏。这种保险丝通常标识为“高速”的，并说明适合用于半导体器件保护。

可移植性

这种技术对所有的 8051 系列微控制器都是可行的（还包括大多数其他微控制器和微处理器系列）。

照例，如果用于没有内部上拉电阻的端口引脚，则需要在设计中加入这样的上拉电阻（10K 即可），详情参见直接 LED 驱动。

优缺点小结

- ◎ 软件开发人员似乎认为分立的三极管器件是“旧式”的，已经被相应的 IC 器件替代了。实际情况并非如此。在许多系统中，特别是在需要少量低功率或者中等功率的驱动器时，分立式三极管被广泛地使用，相当重要的原因是其价格只要 0.10 美元（大约是 IC 价格的一半），即使订货量很小。所以分立式三极管是性价比非常高的解决方法。
- ◎ BJT 可以在很低的电压下工作，与很多低电压的微控制器兼容。
- ◎ BJT 的最大放大倍数大约是 100，这意味着，如果只用单个 BJT，在大多数情况下，驱动电流被限制在 1A~2A。相比之下，一个 MOSFET 可以轻易地开关 100A 的负载。
- ◎ 三极管的开关速度（特别是关断速度）大约是 0.5 微秒。这个速度看起来很快，但是在一些脉宽调制应用中（参见硬件脉宽调制）可能还不够快。

相关的模式和替代方案

BJT 驱动的主要替代方案是 IC 驱动和 MOSFET 驱动。

例子：驱动大功率红外 LED 发射器

红外线（IR）LED 广泛地应用于家庭中的遥控，同时也用于许多保安系统。和普通的 LED 相比，红外 LED 通常需要较大的电流驱动，但其电路连接方式是一样的。

例如，西门子的 SFH485 红外 LED 要求 100mA 的电流驱动，正向压降约为 1.5V。如果要用 8051 微控制器的端口 1 驱动这个 LED，可以使用图 7.16 中电路的一个变体。

这里将使用一个廉价的 PNP 三极管，2N2905。这个三极管可以驱动最大 600mA 的电流。对 100mA 的 I_{LED} ，5V 电源电压，1.5V 的 LED 正向压降，所需的 R2 阻值为：

$$R_2 = \frac{5.0V - 1.5V - 0.4V}{0.100A} = 31\Omega$$

三极管 0.4V 的饱和电压 (V_{CE}) 是从数据手册中查得的。

这里将使用稍大的标准电阻器价值 33Ω 。虽然最接近的阻值是 30Ω ，但这要冒让 LED 工作在稍高的正向电压的风险，可能会减少 LED 的寿命。

最终的电路如图 7.23 所示。

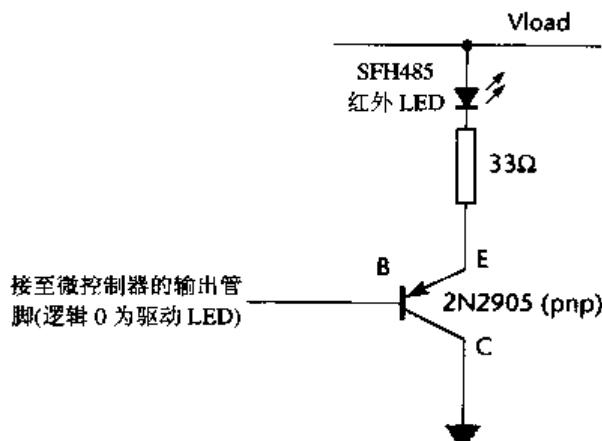


图 7.23 用三极管 (PNP) 驱动红外 LED

注意，逻辑 0 为驱动红外线输出。

例子：驱动一个大蜂鸣器

假设需要驱动一个很响的蜂鸣器用作警报系统的一部分。例如，Klaxon 的 KTB 蜂鸣器需要 $30mA$ 和 $12V$ 的驱动才能工作。图 7.24 展示了一个合适的驱动电路。

这里，再一次使用了 2N2905 作为低成本的设计方案。

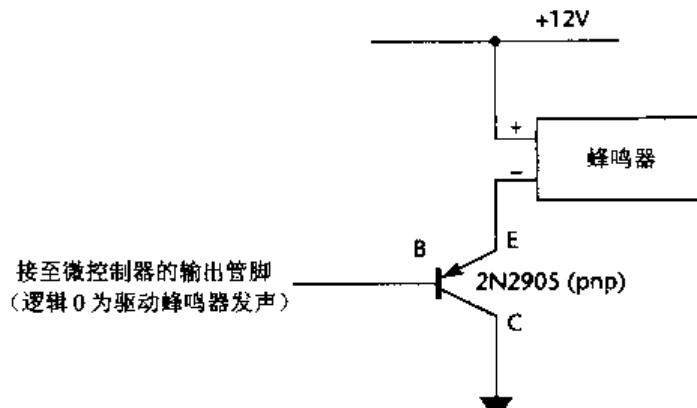


图 7.24 用 2N2905 控制 Klaxon 蜂鸣器

进阶阅读

IC 驱动器

适用场合

用 8051 系列的微控制器开发一种嵌入式系统。

需要为该系统设计相应的硬件。

问题

怎样从单个微控制器的端口安全地控制多个中等功率的直流负载？

背景知识

一般的背景知识参见 IC 缓冲放大器。

解决方案

在本节将考虑“吸收”和“提供”电流的 IC 驱动器。

通用灌电流 IC 驱动器

大多数通用“灌电流”IC 驱动器适用于开关 50V 的直流电压（也有可以开关更高电压的品种）。一般能吸收 0.5A~2A 的电流。

一个常用的驱动 IC 是 ULN2803 系列（许多厂商都生产）。例如，ULN2803A 包含 8 个驱动器，每个能开关 50V, 0.5A（直流）。注意，每个驱动器由一对达林顿管（一对连接在一起的 BJT）组成，达林顿管通过将两个三极管级联以增加电流增益（参见图 7.25）。

注意，这些器件包括了二极管以保护芯片免受“感应冲击”（参见“可靠性和安全性问题”一节），这样也减少了元件的数量。

同时也注意到：

- ULN2803A 不需要单独的电源接线。
- 引脚 9 应该接地。
- 如果用于开关电感负载，内部 8 个二极管的共阴极（引脚 10）应该接到大功率供电电源（可能高达 50V）的正端。
- ULN2803A 的开关速度大约是 1 微秒。

通用的源电流 IC 驱动器

ULN2803 系列是灌电流驱动 IC，可有些时候，需要的是电流源。这里有一个源电流驱动器件的例子，UDN2585A。UDN2585A 可以在 8 个输出引脚上为每个输出提供最大 120mA 的电流，而输出电压可高至 25V（参见图 7.26）。

UDN2585A 将被用于本书的许多例子中，作为多段 LED 显示，详情参见第 21 章。

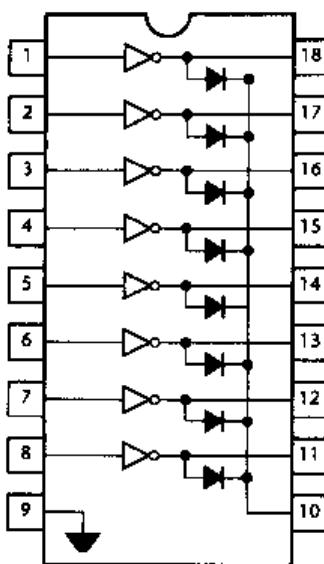


图 7.25 Allegro ULN2803A 的引脚引线和内部细节（灌电流缓冲放大器）
(承蒙 Allegro 微系统公司允许转载)

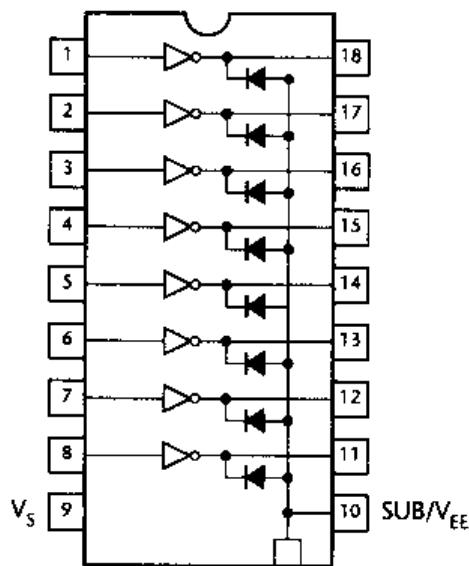


图 7.26 UDN2585A (源电流驱动器) (承蒙 Allegro 微系统公司允许转载)

同时也注意到：

- UDN2585A 的引脚 9 应该接到大功率供电电源（可能高达 25V）的正端。
- 如果用于开关电感负载，内部 8 个二极管的共阳极（引脚 10）应该接地（这样做至少没什么坏处）。
- UDN2585A 的开关速度大约是 5μs。

上拉电阻

在使用这个模式时，可能需要在硬件设计中加入上拉电阻，接在缓冲放大器的输入端，详情参见直接 LED 驱动。

例如，在小的 Atmel 器件上（比如 89C2051），引脚 P1.0 和 P1.1 都没有上拉电阻。使用这类器件时，需要增加外部上拉电阻。一般上拉电阻的阻值为 10K。在图 7.27 中图示了如何用 AT89C2051 控制 ULN2803A。

注意，通常缓冲放大器的输出端不需要上拉电阻。

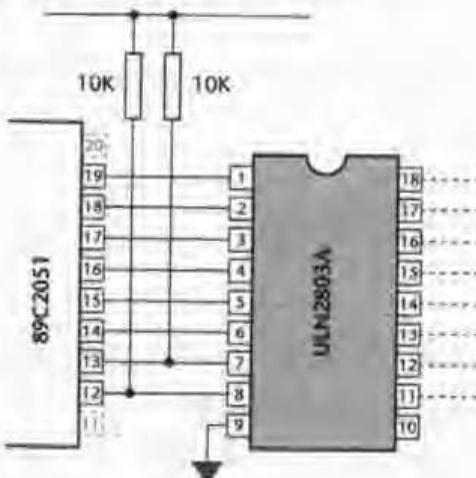


图 7.27 Atmel AT89C2051 输出引脚的外接上拉电阻

硬件资源

这种模式的每个实现方案都至少需要一个端口引脚。

可靠性和安全性

驱动大功率直流负载时需要考虑很多重要的可靠性和安全性问题。这已经在 BJT 驱动器的模式中论述过了，更加详尽的资料请参看 BJT 驱动模式。

注意，当在安全性至关重要的系统中开关大功率负载时，最好在微控制器端口和 IC 驱动芯片间使用一片 IC 缓冲放大器。

可移植性

这种技术对所有的 8051 系列微控制器都是可行的（还包括大多数其他微控制器和微处理器系列）。

照例，如果用于没有内部上拉电阻的端口引脚，则需要在设计中加入这样的上拉电阻（10K 即可），详情参见直接 LED 驱动。

优缺点小结

- ◎ 如果需要驱动两个以上中等功率直流输出，IC 驱动器是一个廉价而简单易用的设计方案。
- ◎ 和三极管驱动器不同，IC 驱动器的工作电压限制比较严格：多数情况下，IC 驱动器的工作电压范围比用在电池供电系统的 8051 还要小。这意味着，如果在电池供电设备中使用 IC 驱动器，则必须使用更复杂的（稳压）电池电源供应设计，这可能会显著增加成本。如果用于电池供电的设备，可能只在使用 IC 驱动器驱动四个以上的输出引脚时才是经济的。
- ◎ 大部分廉价 IC 驱动芯片需要 5V 电源，所以不能简单地用于 3V 的设计。

相关的模式和替代方案

本章的其他模式提供了可供选择的替代方案。

例子：显示错误代码

许多嵌入式系统的一个基本要求是要能让用户知道可能已经发生了的错误。这些错误可能包括：传感器出错、传动装置出错或者从节点出错。

报告这些错误的一个简单、廉价的方法就是利用一个“错误端口”：如果系统工作正常，这个端口将显示“正常”代码；否则，显示错误代码。

显示这些代码的硬件由 8 个 LED 组成，每个 LED 连接至一个端口引脚。为了确保在任何照明条件下，代码都能看得见，可能需要使用高亮度 LED。典型高亮度 LED 需要 25mA 的电流驱动：8 个 LED 的总驱动电流（200mA）远远超过了微控制器的驱动能力，大约是 IC 缓冲放大器驱动能力的 3 倍。

图 7.28 展示了一种错误代码显示的硬件设计，使用了一片 ULN2803A IC 驱动器。

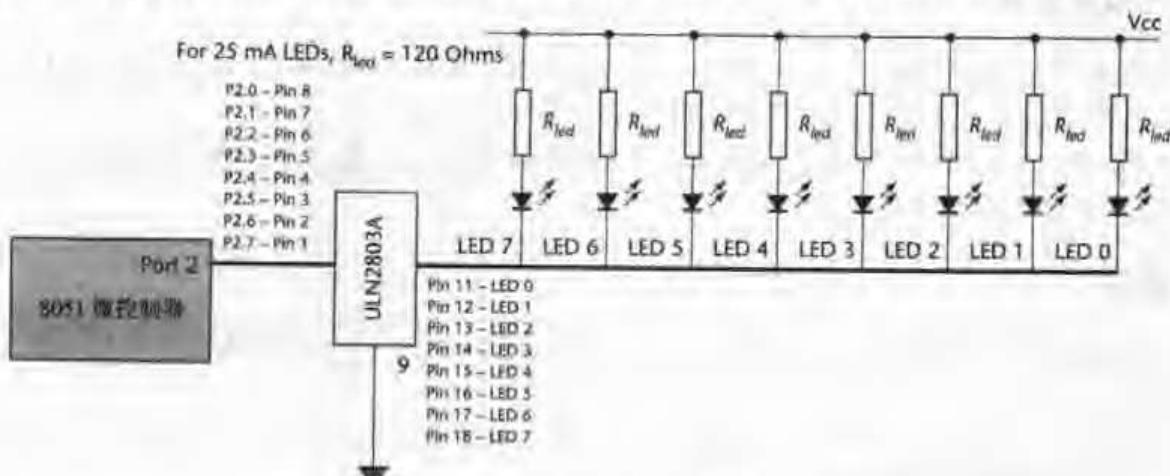


图 7.28 嵌入式系统的错误代码显示硬件

注意，相应的错误代码显示软件将在第 14 章讲述。

进阶阅读

MOSFET 驱动器

适用场合

用 8051 系列的微控制器开发一种嵌入式系统。

需要为该系统设计相应的硬件。

问题

如何用微控制器控制高电流的直流负载（高达 100A）？

背景知识

在 BJT 驱动器一节已经讲述过，双极结型晶体管是一个电流控制开关。对比之下，金属氧化物半导体场效应晶体管（MOSFET）是电压控制开关（图 7.29）。栅极电压为 0 时，MOSFET 将关断在漏极和源极间流动的电流，即使漏极和源极间加上了几百伏的电压。

MOSFET 的“导通”时间和 BJT 差不多，但是和 BJT 不同的是，MOSFET 的“关断”时间也很快（50 ns 左右）。因此，MOSFET 一般用于要求高开关频率的系统（最高大约 1MHz），诸如基于脉宽调制（PWM）的速度控制和脉冲频率调制系统（例如，参见硬件脉宽调制和硬件脉冲频率调制）。

MOSFET 另一个重要特性是氧化金属栅极电气上是和电流通道相隔离的，这意味着在信号周期的任一刻，栅极和电流通道间基本上都没有电流流动。这使得 MOSFET 具有极高的输入阻抗，同时意味着 MOSFET 对静电非常敏感，参见“可靠性和安全性问题”。

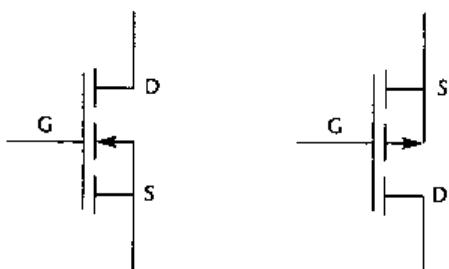


图 7.29 MOSFET 的两种类型，[左]N 沟道器件，[右]P 沟道器件

解决方案

到目前为止，已经讨论过的接口电路中没有一种是针对开关大直流电流设计的。MOSFET

为很多高功率的系统提供了一种灵活而经济的驱动方案，可以开关 100A 甚至更高的电流。MOSFET 通常用一种能有效使用散热片的方式封装，从而让正常工作中产生的热量能散发出去（图 7.30）。



图 7.30 功率 MOSFET 的典型封装 (TO-220)，为安装散热片做了准备

越来越多的 MOSFET 可以直接用处理器的端口驱动，可是在大多数情况下 MOSFET 必须用高电压 (12V+) 驱动，这大大超过了端口能提供的电压。另外，在 BJT 驱动器曾讨论过，通常在微控制器端口和任何大功率负载间加一个额外的保护层是一个好的设计方法。

在这种情况下，比较好的设计方案是在端口和 MOSFET 间用一个“电平转换”IC 作为接口，例如廉价的 74x06。

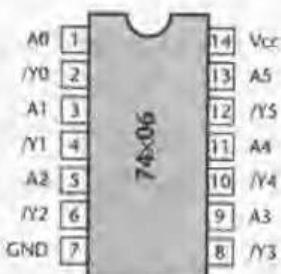


图 7.31 电平转换器 74x06 的引脚引线

注意，74x06 内含 6 个反相缓冲放大器：连接 A0（缓冲放大器 0 的输入）到/Y0（缓冲放大器 0 的输出），A1（缓冲放大器 0 的输入）到/Y1（缓冲放大器 0 的输出），依此类推。Vcc 应连接至 5V 电源；缓冲放大器的输出端电压可达 12V，就 74F06A 来说可达 30V，输出电流大约为 100mA。

例如，参见图 7.32。这幅图说明了一个很有用的 MOSFET 器件 IRF540 的用法，通过 74x06 连接至一个微控制器端口。注意，当 74x06 的反相特性使驱动负载的逻辑为“逻辑 0”，这也是一个好的设计方法。

还需注意，IRF540 可以开关最大 100V，3A 的负载（直流）。但对于 N 沟道版本，IRF540 有一个非常低的导通电阻 (0.04Ω)。这个电阻非常重要，因为 MOSFET 的功率损耗是由如下的公式决定的：

$$P = I^2 R$$

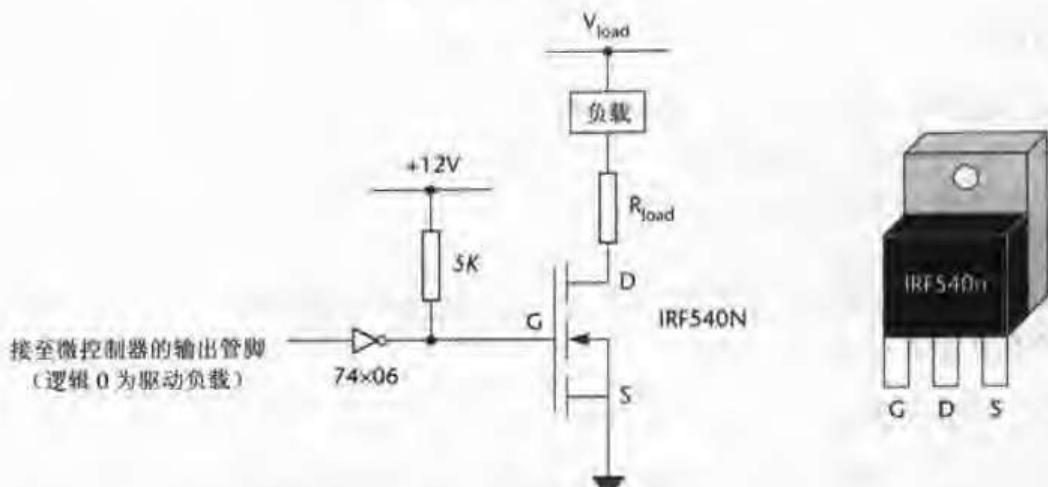


图 7.32 使用 MOSFET 驱动器

注意：逻辑 0 输出用于接通负载。当把 74x06 用作电平转换时，上拉电阻是必需的：上拉电阻应连接到大功率电源的正端。

在 30A 的负载电流下，即使很低的导通电阻也会转化为大量的功率损耗。记住，这些功率是要变成热量散发掉的，损耗越大，所需的冷却能力就越大。

上拉电阻

在使用这个模式时，可能需要在硬件设计中加入上拉电阻，详情参见直接 LED 驱动。

硬件资源

这种模式的每个实现方案都至少需要一个端口引脚。

可靠性和安全性

驱动大功率直流负载时需要考虑很多重要的可靠性和安全性问题，这些已经在 BJT 驱动器的模式中讲述过了。更加详尽的资料请参阅 BJT 驱动模式。

用手接触 MOSFET 器件时的注意事项

因为 MOSFET 器件栅极电极上的氧化层非常薄，所以 MOSFET 很容易因静电放电而被破坏。所以用手接触 MOSFET 时需要采取适当的预防措施。

可移植性

这种技术对所有的 8051 系列微控制器都是可行的（还包括大多数其他微控制器和微处理器系列）。

照例，如果用于没有内部上拉电阻的端口引脚，则需要在设计中加入这样的上拉电阻（10K 即可），详情参见直接 LED 驱动。

优缺点小结

- ◎ MOSFET 有大电压和大电流驱动能力。
- ◎ MOSFET 的开关速度很高。
- ◎ MOSFET 对静电敏感。
- ◎ 对小电流的驱动时，BJT 更加经济。
- ◎ 在系统的低压侧和高压侧之间带内置光电隔离的固态继电器，可能更加可靠，参见固态继电器驱动（直流）。

相关的模式和替代方案

本模式中使用的 MOSFET 可以使用 BJT 代替（详细背景资料参见 BJT 驱动）。但是，BJT 一般较贵，功率也较低。

BJT 的一个缺点是饱和时发射极/集电极间的电压降大约有 1V。在一些系统中（例如，用于控制直流电动机的 H-桥电路），桥的每个臂上都有两个三极管，所以电压降是大约 2V；对于典型的 12V 电源电压，这是个很大的损失。使用 MOSFET，电压降可以降低 10 倍。再加上 MOSFET 的低成本和高开关速度，所以 MOSFET 驱动的使用非常普遍。

对于大电流和/或高电压负载的驱动，另一个替代设计是绝缘栅极双极晶体管（IGBT）。IGBT 结合了 BJT 和 MOSFET 的一些最好的特性，它具有低的通态电阻和大约 1 微秒的关断和接通时间。

例子：点亮一盏灯

图 7.33 展示了如何用 MOSFET 驱动器控制一盏（12V，20W）电灯。

注意，即使是这样相对低电压的灯，也有很大的冲击电流，热敏电阻 R_t 和灯串联以减小冲击电流，详情参见 BJT 驱动器。

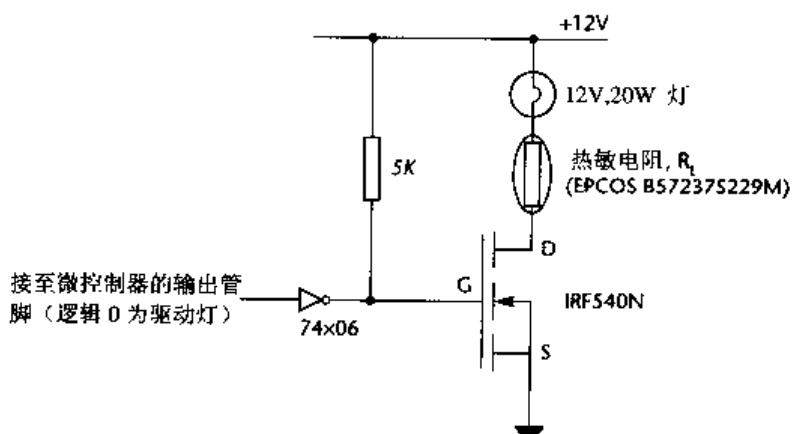


图 7.33 用 MOSFET 驱动器控制 12V，20W 的灯

例子：开环直流电动机控制

图 7.34 展示了如何用 MOSFET 驱动器控制一个 (12V, 2A) 直流电动机。

注意，其中使用了一个二极管用于关断电动机时保护驱动电路免受感应冲击，有关详细情形参见 BJT 驱动器。

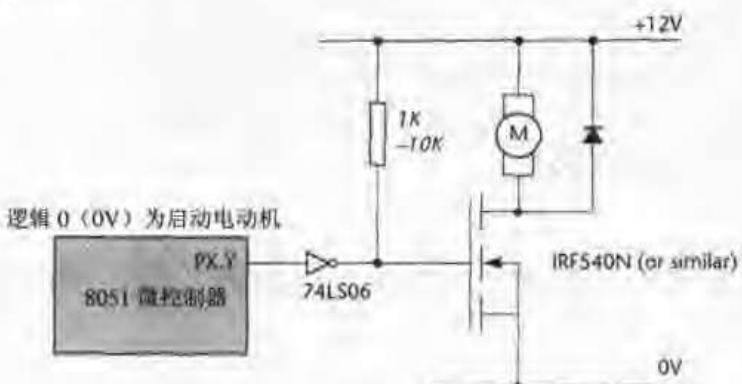


图 7.34 使用 MOSFET 驱动器控制直流电动机

进阶阅读

固态继电器驱动（直流）

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的硬件。

问题

怎样用微控制器“接通”或者“切断”一个高电压的（直流）设备？

背景知识

固态继电器是一种半导体器件，也就是说设计用于代替传统的电磁（EM）继电器。因为本书不推荐使用电磁继电器开关直流负载，因此在第 8 章讨论交流负载的开关前，不考虑电磁继电器。如果不熟悉电磁继电器，则应在考虑使用电磁继电器前查阅第 8 章有关电磁继电器的背景资料（特别是电磁继电器驱动）。

解决方案

和电磁继电器不同，固态继电器（SSR）本质上是纯粹的电子器件：没有活动的（机械）

开关触点。有两种固态继电器，直流和交流 SSR。注意，和电磁继电器不同，直流 SSR 不能开关交流电源，交流 SSR 不能开关直流电源。为什么会这样将在下文中解释。

通过使用光电技术，SSR 在“控制”和“开关”电路间提供了非常高的隔离。“控制”输入连接到一个（或多个）LED 上。这些 LED 将控制一个连接到开关电路的光敏三极管或者光敏二极管阵列，而 LED 和光敏器件间没有任何电的联系。就直流 SSR 来说，开关电路通常基于 MOSFET，所以其电流和电压开关能力与 MOSFET 类似。

SSR 的使用通常很简单：SSR 的输入端和微控制器的端口电压兼容，并且因为内置的光电隔离，通常没有必要在微控制器和 SSR 间增加附加的门电路。

下面的例子将说明如何使用 SSR。

上拉电阻

使用这个模式时，可能需要在硬件设计中加入上拉电阻，接在 SSR 的输入端。详情参见直接 LED 驱动。

硬件资源

这种模式的每个实现方案都至少需要一个端口引脚。

可靠性和安全性问题

驱动大功率直流负载时需要考虑很多常见的可靠性和安全性问题，这些已经在 BJT 驱动器的模式中讲述过了，更加详尽的资料请参看 BJT 驱动模式。

SSR 有多可靠？

在电气特性上，SSR 不如电磁继电器可靠：在过电压的情况下（例如由于电感负载的反电势）或者过电流的情况下（可能由于冲击电流），SSR 会损坏。如果怀疑 SSR 的可靠性，应使用更高的额定值。也就是说，在用 200V 的器件就可能够了的情况下，应使用 300V 的器件。

另一个问题：可能有人会设法并联使用多个 SSR，以增加电流输出。这种设计基本不能工作而且根本不可靠。主要的问题是没有办法保证所有的继电器精确地在同一时刻接通。当一个继电器首先接通，电源电压将降低，通常会低到使第二个（以及后续的）SSR 无法接通，直到第一个继电器烧毁，电源电压恢复正常。这样最有可能的结果是，在大约 1ms 的时间内，所有的继电器相继烧毁。

SSR 的名称意味着什么？

尽管名称相近，固态和电磁继电器间存在重要的区别，特别是在电路测试上。如果使用了电磁继电器，则可以通过用万用表测量开关触点的电阻来检查触点闭合了没有。当触点闭合时，电阻应该基本为 0；而触点断开时，电阻应该为无穷大。而且不用连接高压侧就能做这样的测试。

但是对基于 SSR 的电路没法做这样的测试：当用万用表测量时，大多数 SSR 总是显示电阻为无穷大。测试 SSR，需要在特定的工作电压下进行。所以初期的测试最好（小心地）使用一高电压负载进行（通常使用普通的家用灯泡）。

如果 SSR 故障，将发生什么？

SSR 的典型损坏方式为输出短路，这是非常危险的。

可移植性

SSR 能被用于任何处理器类型。然而，仍然存在其他需要考虑的可移植性问题。

最重要的（已经简略地提到过的）是交流 SSR 不能用于开关直流电。原因是交流 SSR 包含过零点传感电路（详情参见第 8 章）。因为直流电源电压从不过零，所以 SSR 永远无法接通。

同样，因为大多数直流 SSR 基于 MOSFET，而 MOSFET 不能用于开关交流电，所以 SSR 最多以整流器方式工作一会儿，然后就会过热。

优缺点小结

- ⊕ SSR 不会磨损（在正常使用下）。
- ⊕ SSR 抗冲击和振动。
- ⊕ SSR 具有很高的开关速度。
- ⊕ 通过使用光电技术，SSR 在“控制”和“开关”电路间有很高的隔离水平。
- ⊕ SSR 只产生很低的电噪声，并且工作时无噪音。
- ⊕ SSR 没有开关跳跃。
- ⊖ SSR 会立即且不可恢复地被过电压和/或过电流损坏。
- ⊖ SSR 的典型损坏方式是输出短路，这是非常危险的。
- ⊖ 开关侧有最低工作电压和工作电流的限制，而且可能相当的高，从而使初期的测试复杂化。
- ⊖ 电磁继电器通常可以开关更高的电压和电流。
- ⊖ 不像电磁继电器，SSR 的开关侧在断开状态下仍有一些漏电流。
- ⊖ 接通状态的电阻一般比电磁继电器大得多：这将导致无用的发热，而且还需要散热器散热。

相关的模式和替代方案

替代方案参见本章和第 8 章的其他模式。

例子：SSR 在通信中的应用

小的直流半导体继电器常用于电信设备（例如，调制解调器），代替大的电磁继电器。

实际上，因为电信市场的重要性，已经为电话线路电流传感设计了专用的 SSR（例如，Erg 的元件产品）。

在调制解调器和类似的装置中，SSR 提供大约 200~300V（直流）/200mA 的额定输出。这些 SSR 可在 4kV 的电压下提供较低的接通电阻（通常 10Ω ），而断开电阻可达 $(500M\Omega)$ 。

例子：开环直流电动机控制

在 MOSFET 驱动器介绍了一个用 MOSFET 控制直流电动机的例子（详见图 7.34）。

如果使用适当的 SSR，可以显著地简化这个电路。例如，电动机的正常工作需要 2A（连续的）12V 的驱动。这里可以使用 IOR PVN012 固态继电器。和大多数 SSR 不同，这个 SSR 可以开关交流或者直流负载，最大（20V，4.5A）。IOR PVN012 没有过零点检测。其最大控制电流为 10mA，和微控制器相兼容。图 7.35 展示了所需的电路。

注意：SSR 的一个主要优点是在控制器和电动机之间提供了很大程度的隔离。同时也注意，脉宽调制在某种情况下也可以用于控制电动机的转速，参见硬件脉宽调制和软件脉宽调制。

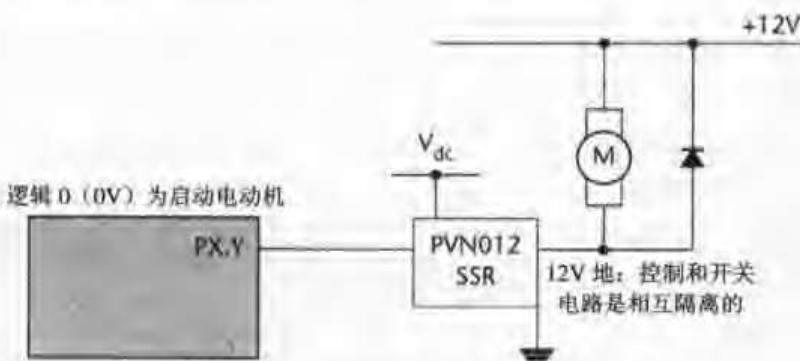


图 7.35 使用固态继电器的开环直流电动机控制

进阶阅读

Chapter 8

交流负载驱动

引言

考虑一下以下问题：

- 家庭或者工业供暖系统中的水泵必须在预定的时间打开和关闭。
- 围绕一座工厂的一系列高亮度照明灯，用于发生安全事故时的应急照明。
- 酿酒厂的电加热器必须维持精确的温度。
- 洗衣机中的电动机必须按照预先编好的程序运转。

在这些和许多其他系统中，必须安全可靠地控制电压相对较高的交流电源。本章将考虑解决这个问题的两种普遍使用的控制方法：

- 电磁继电器驱动
- 交流固态继电器驱动

电磁继电器驱动

适用场合

用 8051 系列的微控制器开发一种嵌入式系统。

需要为该系统设计相应的硬件。

问题

怎样用微控制器“接通”或者“切断”一个大功率的（交流电）设备？

背景知识

解决方案

在此模式中，将考虑怎样用电磁继电器控制交流电源。

电磁继电器本质上是由流过电磁线圈的电流控制的机械开关。电磁继电器多年来一直被用于接通和切断交流负载，通常负载的功率都很大。大多数情况下，大容量的电磁继电器不能直接从微控制器的端口驱动，需要用某种形式的三极管或者IC驱动电路驱动（例如，参见图8.1）。

近年来，一些电磁继电器可以用逻辑电平操作（5V，几个mA）。这种继电器的典型例子是图8.2所示的小型簧片继电器。

这类簧片继电器能开关10W功率和强电电压（最大交流250V）。注意，许多继电器的线圈两端都跨接了二极管，而且有许多的触点类型（常开、常闭、多刀）组合。

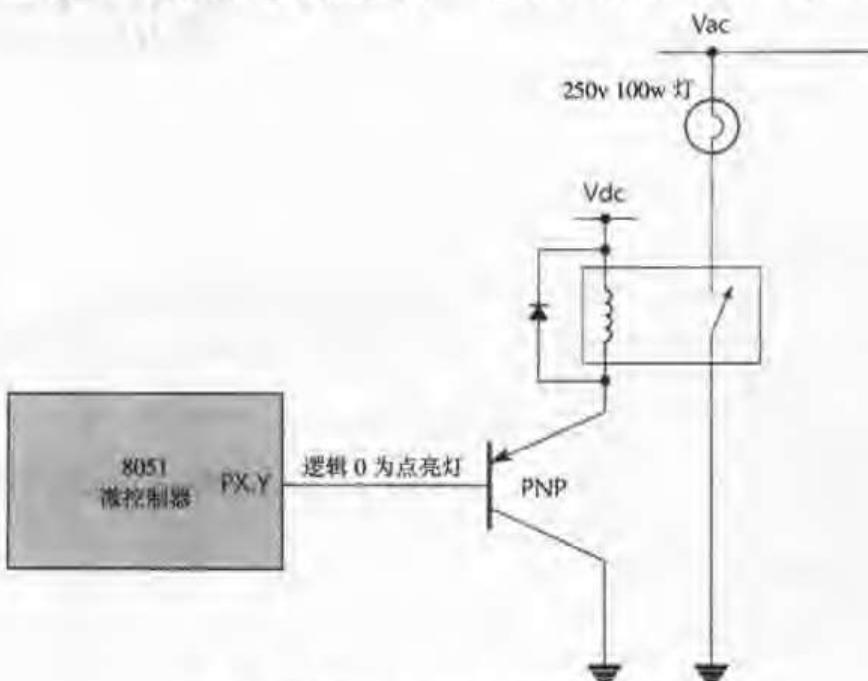


图8.1 使用电磁继电器控制交流电灯

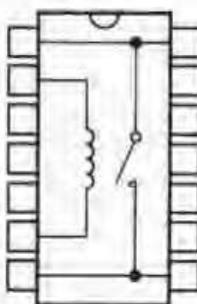


图8.2 典型的小型（电磁式）簧片继电器

上拉电阻

使用这个模式时，可能需要在硬件设计中加入上拉电阻，详情参见直接LED驱动。

可靠性和安全性

涉及强电的设计

始终要记住，人不能接触用于处理强电的电路。强电电路的设计必须使用户不会接触到危险的电压。

引脚复位后的初始电平

系统复位后，许多端口的特殊函数寄存器（SFR）被设为 0xFF。这对安全性和可靠性非常重要。

这个问题在 BJT 驱动器一节已经讲述过了，详情请参阅该模式。简单说来，因为输出引脚复位后为高，所以必须确保有安全性要求的设备连至微控制器时是低电平有效的，即相应端口引脚输出为 0 时启动设备。

过零点检测

将一个收音机放在住宅或者办公室的（机械式）照明开关的旁边。开关电灯时注意听收音机。收音机发出的噼啪声是电磁干扰（EMI）的典型特征。电磁干扰是机械开关闭合和断开时触点间的电弧形成的——这种 EMI 会干扰收音机的接收；但对于嵌入式系统，却可能是致命的。所以开关大功率负载时要小心，特别是对交流负载，因为交流负载一般都是高压的。

这个问题有解决的办法。为了弄清楚工作原理，请看图 8.3 所示的简单交流波形。如果在波形周期中合适的点开关负载，干扰可以大大减少；图 8.3 中在 A 点开关将引起最大的干扰，而在 B 点开关几乎不会引起干扰。所以，为了使干扰最小，需要检测出波形的过零点，并在这一点切换开关。

电磁继电器的一个重要缺点是不能和过零点检测电路配合使用。

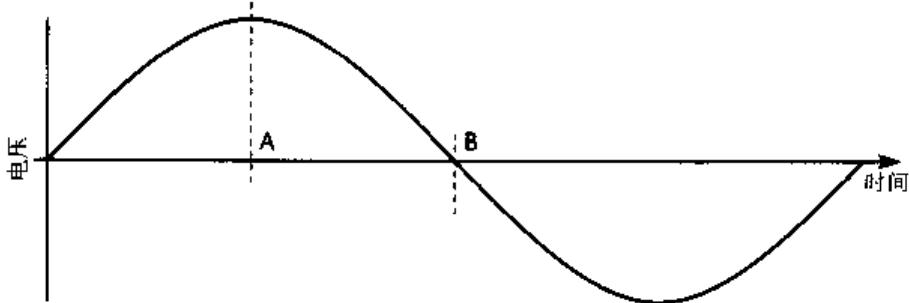


图 8.3 交流供电的最差开关点（A）和最优（B）开关点

接通灯和其他电感负载

正如第 7 章中所述，并不是所有的负载都是阻值固定的电阻性负载。例如，当控制灯或者直流电动机时，电路接通的初始电流非常大。这个冲击电流可能持续数百毫秒，才逐渐降低到

稳态值。驱动电路要能够承受接通瞬间的冲击电流。

解决冲击电流问题的一种方法是提高驱动电路的额定设计参数。这意味着，为了承受冲击电流，稳态电流为 1A 的灯或者电动机的驱动电路的驱动能力应达到 10A 甚至更高。注意，一般无法估测冲击电流的大小。请查对数据手册——一般其中都会提供相关信息。如果找不到精确的资料，至少假定冲击电流为稳态电流的 10 倍。

记住，正如 BJT 驱动器一节所述，另一个解决这个问题的方法是使用热敏电阻与负载串联。

并不是所有的驱动电路在过大的负载下都会立刻失效，在测试时驱动电路可能可以可靠地工作。然而，任何驱动电路超过最大额定值使用都将显著地缩短其使用年限，并且可能导致该电路在应用现场失效。如果对电路的抗冲击电流能力有怀疑，就应把电路的额定值至少提高 10 倍，并且在电路中增加热敏电阻。

切断电感性交流负载

和第 7 章对直流负载的论述类似，凡是包含线圈的都是电感负载，常见的例子是电磁继电器和电动机——切断这种负载必须谨慎从事，因为当电流被切断时，电感负载两端的反电动势可能会损坏开关电路。

为了保护控制感性直流负载的开关，如图 8.4 所示，可用二极管防止反电动势（感应冲击）。

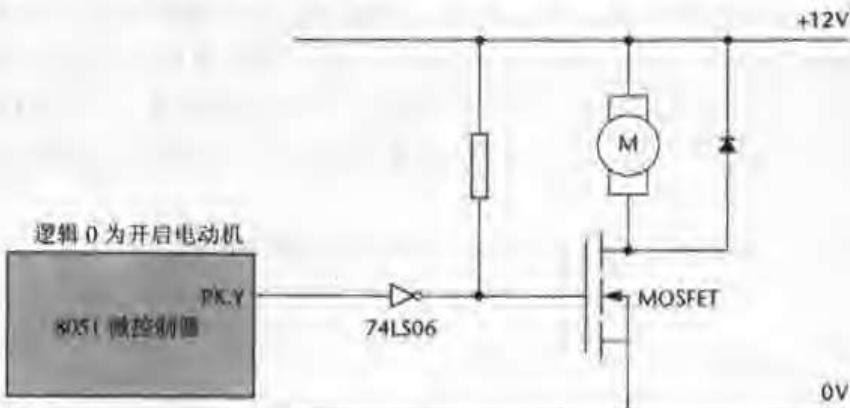


图 8.4 用二极管保护开关免受反电动势冲击 (感应冲击)

注意，这种方法只对直流负载有效。

这个方法对交流负载无效，因为二极管的整流会使驱动电压的半个周波为零。不过，在第 7 章也讲述了一种适用于直流负载的替代方法，通过使用一个电阻吸收反电动势 (图 8.5)。

基于电阻的保护方案有一种变体，被广泛地应用于交流负载切换电路，这种电路被称为阻容冲击吸收器 (图 8.6)。在大多数情况下，使用的电阻阻值 ($R_{snubber}$) 为 $10\sim10k\Omega$ ，使用的电容容量 ($C_{snubber}$) 为 $0.01\sim1\mu F$ (Lander, 1993)；在多数情况下可使用 100Ω 和 $0.05\mu F$ 的参数，但是对安全性至关重要的系统需要认真地核对参数。

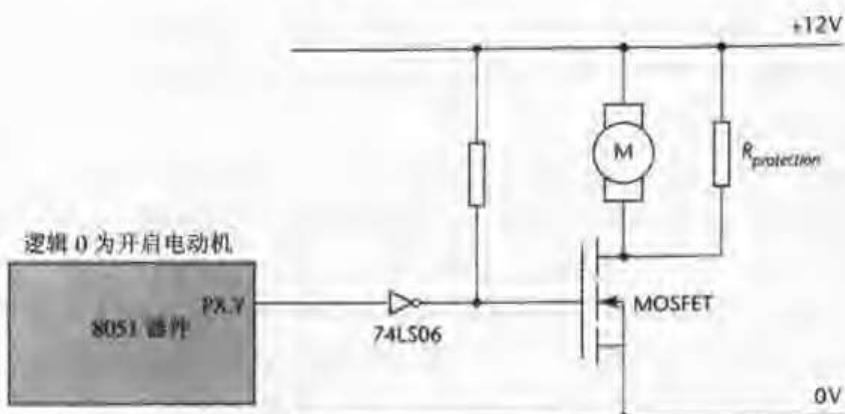


图 8.5 使用电阻吸收电感负载（此处为直流电动机）的反电动势

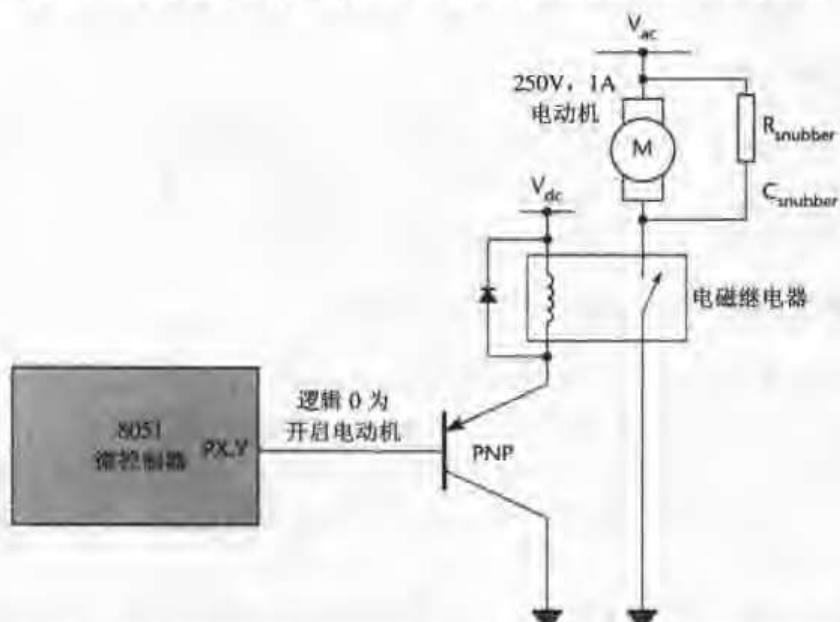


图 8.6 开关感性交流负载时使用 RC 冲击吸收电路保护电磁继电器

可移植性

电磁继电器可以用于任何微控制器、微处理器或者数字信号处理芯片。

优缺点小结

电磁继电器可以控制直流和交流负载，负载的电流可以从几个毫安到数千安培。

- ⑤ 当触点打开时，没有漏电流。也就是说，断开状态的电阻非常大（接近无穷大），此时触点两侧的电压可达 1500V。
- ⑤ 当触点合上时，接通电阻极低，所以继电器的功率损耗非常小。因此继电器不会发热，通常也不需要散热器。
- ⑤ 电磁继电器的购买成本通常比半导体驱动器件低（参见关于维护费用的注解）。

- ⑥ 开关时间一般是毫秒级的，相比之下，半导体开关的开关时间是微秒级的。
- ⑦ 和所有的机械开关一样，当继电器触点打开或合上时存在抖动的现象（第19章将详细研究抖动现象）。
- ⑧ 因为电磁继电器一般不能和过零点检测电路配合使用（参见可靠性和安全性问题），所以会在触点处产生电弧。因此，电磁继电器产生的电磁干扰比半导体继电器严重。
- ⑨ 和半导体开关不同，继电器包含活动部分，而活动部分会磨损。电磁继电器的机械寿命范围较大，但是典型值为1000万到3000万次通断循环。如果每天平均通断十次，可以认为1000万次的寿命相当于永远都不会损坏。
- ⑩ 然而，如果每毫秒通断一次，1000万次也只能工作3小时左右。
- ⑪ 如果电磁继电器处在振动环境中，开关触点可能会抖动——这是很危险的，可能引起系统的可靠性问题。

相关的模式和替代方案

虽然可以使用电磁继电器开关直流负载，但是一般这不是个好主意，控制直流负载的最佳解决方案参见BJT驱动器、IC驱动器和MOSFET驱动器。

交流负载控制的其他替代方案参见SSR驱动（交流）。

例子：用电磁继电器控制中央供暖系统中的泵

许多中央供暖系统需要控制小型的低功率泵。一个典型的泵的例子是Grundfos Selectric UPS 15-50：需要240V的交流电(50Hz)和0.17~0.42A的电流。

集中供暖控制是电磁继电器的一个很好的用途。每天只需接通不多的次数进行加热，所以电磁继电器的寿命会很长。万一继电器发生了故障，维护也不困难。偶尔的电磁脉冲对大多数的家庭环境也没什么问题。

在这个例子中，簧片继电器就能承受所需的电流了。图8.7展示了一个这样的电路。

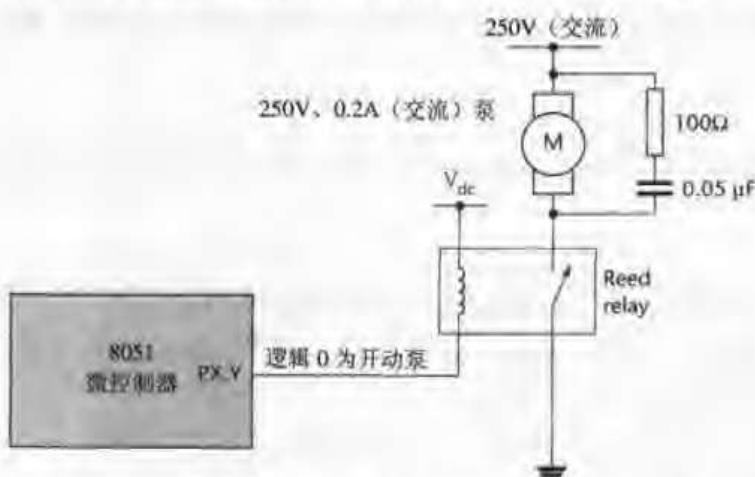


图8.7 使用簧片（电磁）继电器控制中央供暖系统的泵

进阶阅读

固态继电器驱动（交流）

适用场合

用 8051 系列的微控制器开发一种嵌入式系统。

需要为该系统设计相应的硬件。

问题

怎样用微控制器“接通”或者“切断”一个强电（交流电）设备？

背景知识

在固态继电器驱动（直流）一节中介绍了直流固态继电器，有关 SSR 的一般背景资料请参阅第 7 章的相关模式。

这两种器件的主要区别是，直流 SSR 的输出级一般由 MOSFET 组成，而交流 SSR 的输出级通常是双向晶闸管（TRIAC）。简单说来，双向晶闸管是一种允许电流双向流动的半导体开关。这正是开关交流负载所需要的。

解决方案

SSR 的使用通常很简单：SSR 的输入端和微控制器的端口兼容，并且因为内置的光电隔离，一般没有必要在微控制器和 SSR 间增加附加的电路。随后的例子将说明如何使用 SSR。

上拉电阻

使用这个模式时，可能需要在硬件设计中加入上拉电阻。详情参见直接 LED 驱动。

可靠性和安全性

一般的可靠性和安全性问题参见本章的开始部分。

基本的 SSR 使用原则参见固态继电器驱动（直流）。

交流和直流 SSR 的一个关键区别是 MOSFET 的损耗很低，而用于交流 SSR 的双向晶闸管输出级在导通时一般会有 1.5V 的压降。这一压降将导致每安培负载电流 1.5W 的损耗。所以如果使用大功率器件（大约大于 4A），则必须使用散热器。

安装散热器时，注意其和设备的机箱保持绝缘。不要用螺栓将 SSR 固定在机箱上，这可能会引起致命的后果。

可移植性

SSR 可被用于任何处理器类型。然而，仍然存在其他需要考虑的可移植性问题。

最重要的（已经简略地提到过）是交流 SSR 不能用于开关直流电。原因是交流 SSR 包含过零点检测电路。因为直流电源电压从不过零，所以 SSR 永远无法接通。

同样，因为大多数的直流 SSR 基于 MOSFET，而 MOSFET 不能用于开关交流电，所以 SSR 最多以整流器方式工作一会儿，然后就会过热。

优缺点小结

- ☺ SSR 没有开关跳跃。
- ☺ SSR 不会磨损（在正常使用下）。
- ☺ SSR 不会产生噪音。
- ☺ 许多（交流）SSR 内含过零点检测电路，因而能大大减少电磁干扰的发射。
- ☺ SSR 抗冲击和振动。
- ☺ SSR 具有很高的开关速度。
- ☺ 通过使用光电技术，SSR 的“控制”和“开关”电路间具备了很高的隔离水平。
- ☺ 电磁继电器通常可以开关更高的电压和电流。
- ☹ 交流 SSR 的导通电阻比电磁继电器大的多。这意味着会产生多余的热量，并需要散热器或者其他形式的冷却。
- ☹ 和电磁继电器不同，交流 SSR 的开关（通常是双向晶闸管）在断开状态有漏电流。
- ☹ 大多数的交流 SSR 不能开关直流电，部分原因是因为过零点检测电路。
- ☹ 过电压和/或过电流会在瞬间不可修复地烧坏 SSR，而电磁继电器的抗过压过流能力更强。
- ☹ SSR 的典型损坏方式是输出短路，这是非常危险的。
- ☹ SSR 开关侧有最低工作电压和工作电流的限制，而且可能相当的高，这使得小型系统的初期试验很难进行。

相关的模式和替代方案

替代方案参见本章和第 7 章的其他模式。

例子：用 SSR 控制中央供暖系统中的泵

在电磁继电器驱动一节，介绍了一个用小型簧片继电器控制中央供暖系统中的泵的例子。这里将考虑一个使用交流 SSR 的替代方案。

需要控制的泵仍是 Grundfos Selectric UPS 15-50：需要交流 240V (50 Hz) 和 0.17~0.42A 的电流工作。在本例中，将使用 Crydom MP240D3 固态继电器。Crydom MP240D3 适合于和微控制器接口，并且具备交流 280V, 3A 连续电流 (80A 浪涌电流) 的载流能力。图 8.8 展示

了一个合适的电路。

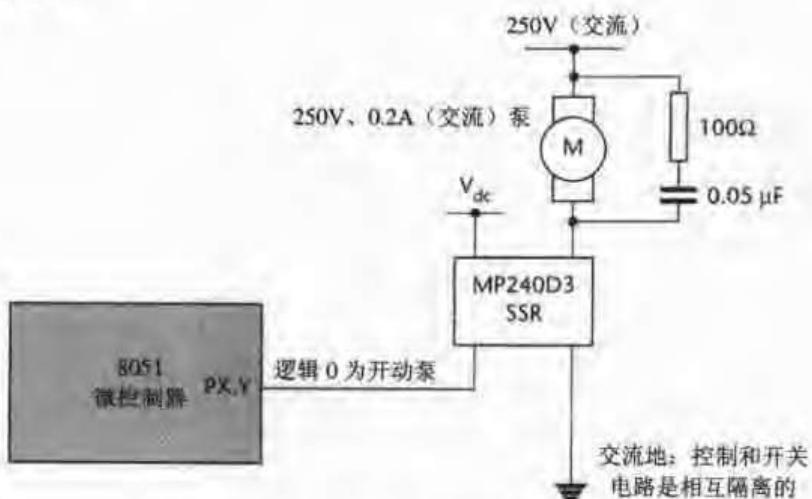
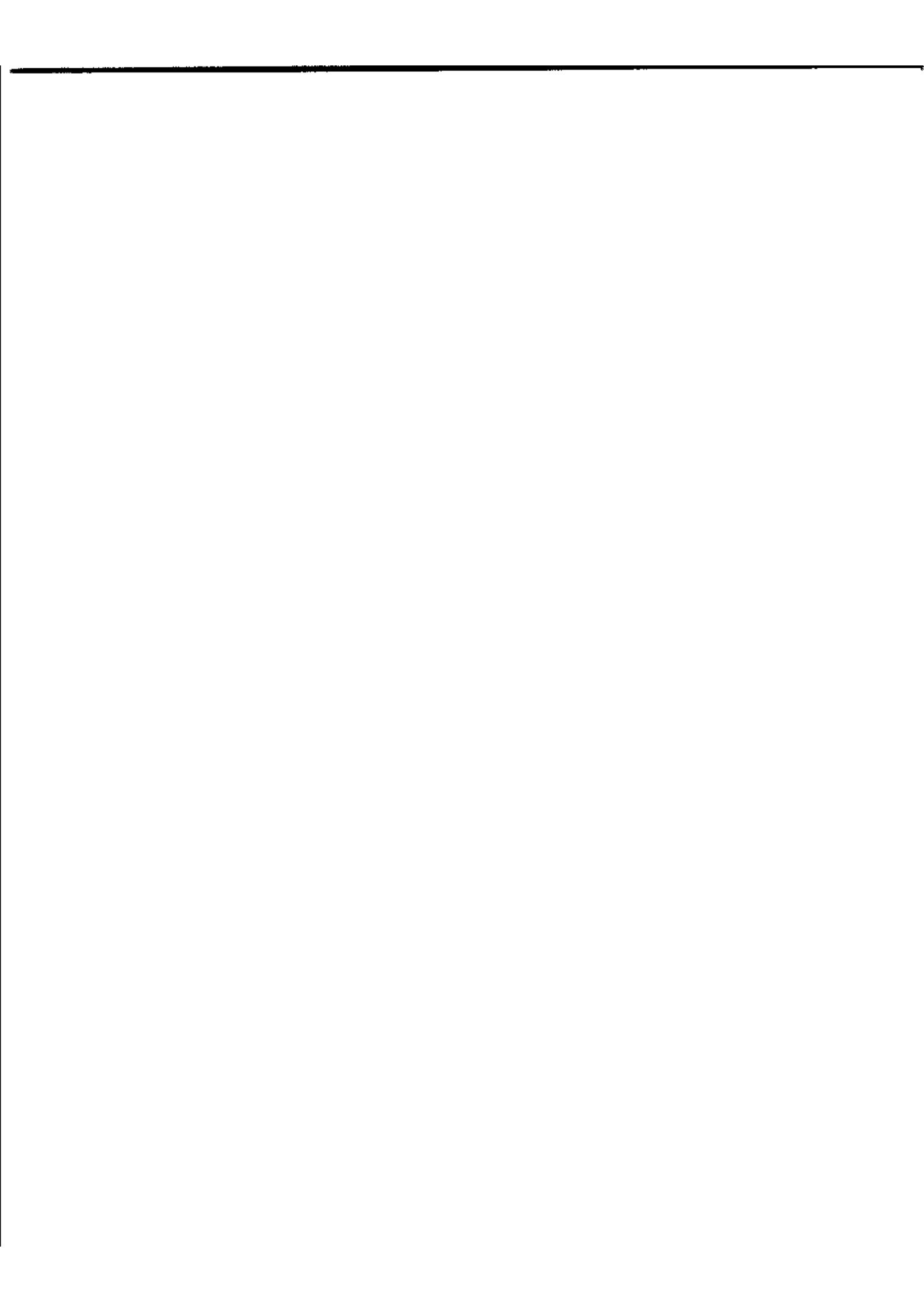


图 8.8 使用 Crydom 固态继电器控制中央供暖系统的泵

但是总的说来，此例中的集中供暖控制大概是电磁继电器最好的应用场合了。每天只需接通不多的次数进行加热，所以电磁继电器的寿命会很长。万一继电器发生了故障，维护也不困难。偶尔的电磁脉冲对大多数的家庭环境也没什么问题。

所以对这类系统，电磁继电器是更好的解决方案，成本也比 SSR 便宜大约 20%。

进阶阅读



Part 2

软件基础

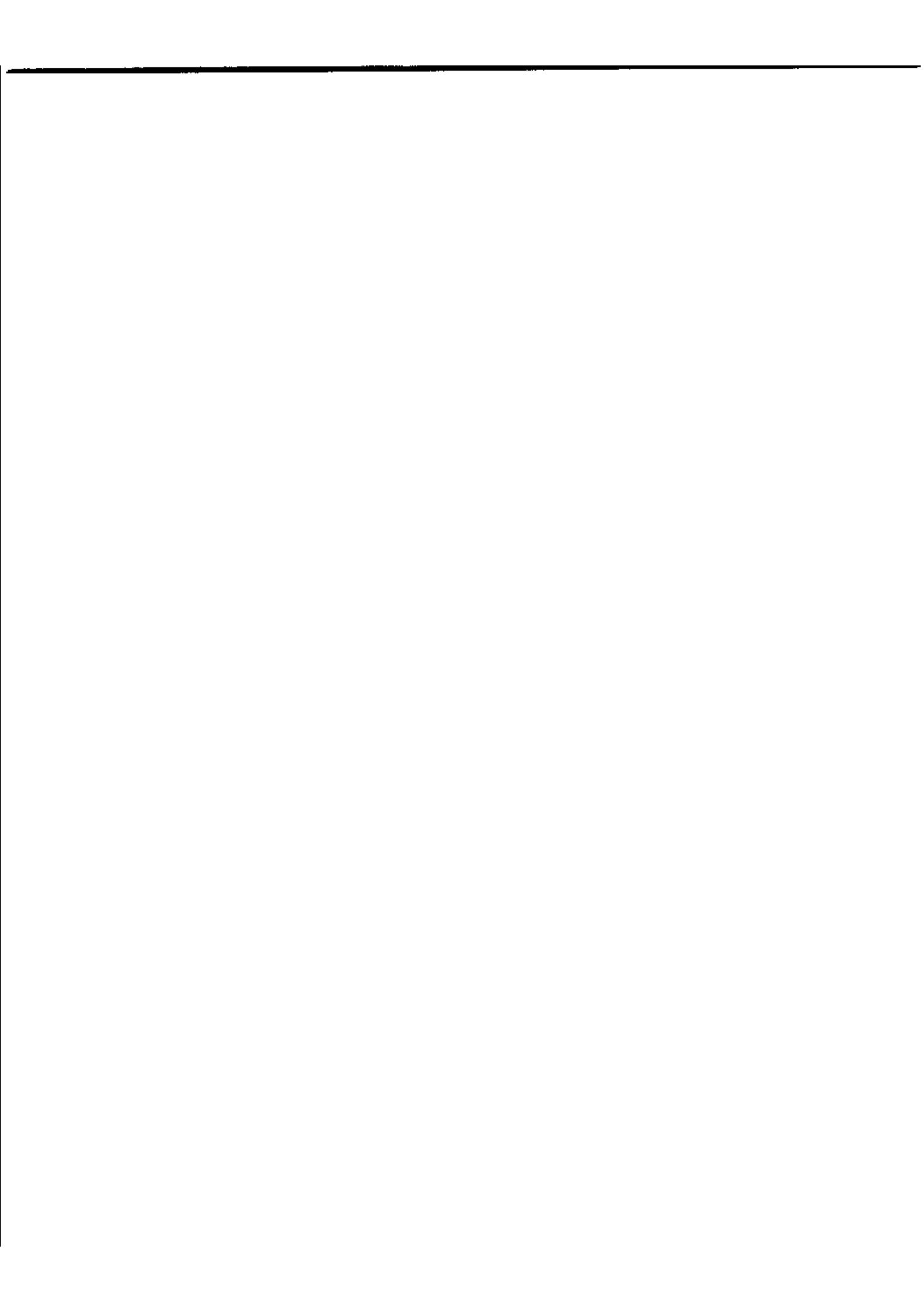
第 1 篇中的章节主要讲述的是为嵌入式系统开发满足要求的硬件平台。在第 2 篇中，将讨论相应的软件基础。

在第 9 章中，描述创建一个嵌入式系统所要求的最小软件平台。

在第 10 章中，讨论适用于第 7 章和第 8 章中介绍的驱动交流和直流硬件的软件技术。

在第 11 章中，探讨用于产生延迟的基于软件和硬件的技术。

在第 12 章中，考虑看门狗定时器的问题并讨论如何用它来提高系统的可靠性。



基本的软件体系结构

引言

本章第一个模式讨论创建一个典型的嵌入式系统所要求的最小软件平台，这个平台被称作超级循环。

请注意，本书使用超级循环来说明第 10、11 和 12 章中的一些介绍性的软件模式。在第 13 章中将说明对于大多数嵌入式系统，合作式调度器将提供一个比超级循环更合适的平台。

第二个模式（项目头文件）是一种实用的标准软件设计准则的具体实现。即，不要在多个文件中复制信息，将公用信息放在单个文件中并在需要的时候引用它。具体地说，项目头文件将系统中所使用的与微控制器相关的信息及项目的许多文件需要的其他关键信息放在一起。

超级循环

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的软件。

问题

为创建一个嵌入式 C 程序，需要什么样的最小软件平台？

背景知识

解决方案

嵌入式系统和台式机系统之间的一个主要区别是：大多数的嵌入式系统仅仅要求运行一个程序。这个程序将在微控制器上电的时候开始运行，并在断电的时候停止运行。

源程序清单 9.1~源程序清单 9.3 中介绍了一种实现该特性的常用软件结构。

```
/* -----
Main.C
-----
一个简单的超级循环应用的结构
[Compiles and runs but does nothing useful]
----- */
#include "X.h"
/* -----
void main(void)
{
    // Prepare for task X
    X_Init();
    while(1) // 'for ever' (Super Loop)
    {
        X(); // Perform the task
    }
}
/* -----
-----文件结束-----
----- */
----- */
```

源程序清单 9.1 简单超级循环的部分演示

```
/* -----
X.H
-----
- see X.C for details.
----- */
// 函数原型
void X_Init(void);
void X(void);
/* -----
-----文件结束-----
----- */
----- */
```

源程序清单 9.2 简单超级循环的部分演示

```
/* -----
X.C
-----
// 一个简单的超级循环应用的“任务”
[Compiles and runs but does nothing useful]
----- */
/* -----
void X_Init(void)
{
    // Prepare for task X
    // User code here...
}
```

```
}

/*
void X(void)
{
    // Perform task X
    // User code here ...
}

/*
-----文件结束-----
*/

```

源程序清单 9.3 简单超级循环的部分演示

源程序清单 9.1~源程序清单 9.3 给出了一种能够运行单个任务（函数 X）的简单的嵌入式结构。在执行了一些系统初始化（通过函数 Init_System()）之后，系统重复运行任务 X，直到系统掉电。

因为没有操作系统可供返回，所以需要“超级循环”或“死循环”，致使系统一直循环直到系统掉电。

硬件资源

超级循环没有重要的硬件资源问题。它不使用定时器、端口或其他设备。只需要少量字节的程序代码，用 C 不可能创建比它需要更少系统资源的工作平台。

可靠性和安全性

基于超级循环的系统既可靠又安全。因为总体结构很简单，并且易于理解，其硬件基础没有哪些方面是对早期的开发人员或后来的系统维护人员隐藏的。相比而言，如果在 Windows 或类似的复杂桌面平台下（包括 Linux 或 Unix）编程，系统并不是在开发者的完全控制之下。如果程序库里其他人写的代码质量低劣，可能会导致程序崩溃。而对于“超级循环”应用，将不需要归咎于任何其他人，这对于与安全相关的系统特别有吸引力。

然而，请注意，基于超级循环的系统并不一定是安全的。实际上，就一般而言，超级循环并不提供嵌入式系统所需要的功能，特别是它不提供在预定时间间隔调用函数的机制。正如在第 1 章讨论的，这种机制是大多数嵌入式系统的主要特征。如果需要这种功能，调度器（参见第 13 章）将是一种更可靠的平台。

可移植性

任何为嵌入式系统设计的“C”编译程序都能编译超级循环程序。由于循环完全建立在 ISO/ANSI “C” 上，所以代码本质上是可移植的。

优缺点小结

- ◎ 超级循环系统的主要优点是很简单，编写、调试、试验和维护相对容易。
- ◎ 超级循环效率很高，极少有硬件资源问题。
- ◎ 超级循环非常容易移植。
- ◎ 如果系统需要精确的定时（例如，需要精确的每 2ms 读取数据），这种结构不能提供所需要的精度和灵活性。
- ◎ 基本的超级循环一直运行在“满负荷”（正常运行方式）下。对于所有系统而言，这未必是必要的，并且它将严重地影响系统功耗。调度器同样能够解决这个问题。

相关的模式和替代解决方案

在大多数环境下，调度器将是一种更合适的选择，参见第 13 章。

例子：中央供暖控制器

假设希望开发一种基于微控制器的控制系统，作为建筑物中的中央供暖系统的一部分。这个系统的最简单版本由一个燃气锅炉（控制对象）、一个传感器（测量房间温度）、一个温度面板（用来设定要求的温度）和控制系统自身组成（图 9.1）。

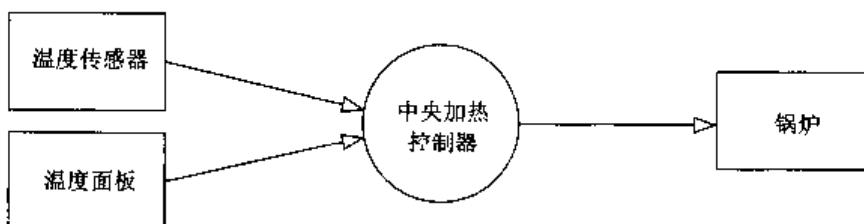


图 9.1 所要求的中央供暖系统控制器的简图

假定锅炉、温度传感器和温度面板通过合适的端口连接到系统上。更进一步假定控制系统将用“C”实现。

由于这里不需要精确定时，使用类似于源程序清单 9.4~源程序清单 9.6 显示的超级循环结构比较合适。

```

/*
Main.C (v1.00)

// 使用“超级循环”的中央供暖系统的结构
[Compiles and runs but does nothing useful]
*/
#include "Cen_Heat.h"
/*
void main(void)
{
}

```

```

// 初始化系统
C_HEAT_Init();
while(1) // “死循环”（超级循环）
{
    {
        // 了解用户需要多高的温度
        // (通过用户界面)
        C_HEAT_Get_Required_Temperature();
        // 了解当前房间的温度是多少
        // (通过温度传感器)
        C_HEAT_Get_Actual_Temperature();
        // 按照需要调节燃气锅炉
        C_HEAT_Control_Boiler();
    }
}
/*-----文件结束-----*/

```

源程序清单 9.4 简单的中央供暖系统的部分代码

```

/*-----*
 * Cen_Heat.H
 *
 * - see Cen_Heat.C for details.
 *-----*/
// 函数原型
void C_HEAT_Init(void);
void C_HEAT_Get_Required_Temperature(void);
void C_HEAT_Get_Actual_Temperature(void);
void C_HEAT_Control_Boiler(void);
/*-----文件结束-----*/

```

源程序清单 9.5 简单的中央供暖系统的部分代码

```

/*-----*
 * Cen_Heat.C
 *
 * 使用“超级循环”结构的集中供暖系统
 * [Compiles and runs but does nothing useful]
 *-----*/
/*-----*
 * void C_HEAT_Init(void)
 * {
 *     // 这里是用户代码...
 * }
 *-----*/
void C_HEAT_Get_Required_Temperature(void)
{

```

```

    // User code here...
}
/*-----*/
void C_HEAT_Get_Actual_temperature(void)
{
    // User code here...
}
/*-----*/
void C_HEAT_Control_Boiler(void)
{
    // User code here...
}
/*-----*
---文件结束---
-*-----*/

```

源程序清单 9.6 简单的中央供暖系统的部分代码

进阶阅读

项目头文件

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的软件。

问题

如何把项目使用的硬件平台的全部相关信息集中在一起？

背景知识

解决方案

正如在第 3 章看到的，8051 系列共用一组通用的内核功能。然而，它是一个系列而不是一组克隆，不同的芯片具有不同的特性和功能。例如，一些芯片每个指令需要 12 个振荡器周期，而其他的芯片执行相同指令却只要 6 个、4 个甚至 1 个振荡器周期（参见第 3 章和第 4 章）。

如果使用运行在某个振荡器频率的某种 8051 芯片来创建系统，当编译项目中有许多不同的源文件时，将需要这些信息。任何一个要使用这些代码的人也需要这些信息。

“项目头文件”只不过是一个将所有信息放在一起，且被所有项目文件包含的头文件。因而，它是标准软件设计准则的一种实际的实现。不用在许多文件中复制信息，将公用信息放在

单个文件中，并在需要的地方引用它。

在本书中的大多数例子中，都将使用项目头文件，它总是被称做 Main.H。源程序清单 9.7 中包括了一个典型的项目头文件的例子。请注意，这是一个真实例子，该文件的一些特性没有在本书中论及。

```
/*-----*
Main.H (v1.00)
-----
LCD_KEY 项目（参见第 22 章）的项目头文件
*-----*/
#ifndef _MAIN_H
#define _MAIN_H
//-----
// 每个项目都需要修改这一段
//-----
// 必须在这里包含相应的微控制器的头文件
#include <AT89X52.h>
// 如果使用延迟，必须在这里包含振荡器/芯片的详细资料
// -
// 振荡器/谐振器的频率 (Hz)，例如 (11059200u1)
#define OSC_FREQ (11059200UL)
// 每个指令的振荡周期数 (6 或 12)
#define OSC_PER_INST (12)
//-
// 以下部分不需要修改
//-
typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long Tlong;
// 杂项宏定义
#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif
#define RETURN_NORMAL (bit) 0
#define RETURN_ERROR (bit) 1
//-
// 中断
// 参见第 12 章
//-
// 通用的 8051 定时器中断（用于大多数调度器）
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5
// 其他的中断（用于共享时钟调度器）
#define INTERRUPT_UART_Rx_Tx 4
#define INTERRUPT_CAN_c515c 17
// -----
```

```

// 错误代码
// 参见第13章
//-----
#define ERROR_SCH_TOO_MANY_TASKS (1)
#define ERROR_SCH_CANNOT_DELETE_TASK (2)

#define ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK(0xAA)
#define ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER(0xAA)

#define ERROR_SCH_ONE_OR_MORE_SLAVES_DIT_NOT_STAR (0xA0)
#define ERROR_SCH_LOST_SLAVE (0x80)

#define ERROR_I2C_WRITE_BYTE_AT24C64 (11)
#define ERROR_I2C_READ_BYTE_AT24C64 (12)
#define ERROR_I2C_WRITE_BYTE (13)
#define ERROR_I2C_READ_BYTE (14)

#define ERROR_USART_TI (21)
#define ERROR_USART_WRITE_CHAR (22)
#endif

```

源程序清单9.7 一个典型的项目头文件的例子（Main.H）

硬件资源

没有硬件资源问题。

可靠性和安全性

使用项目头文件有助于提高可靠性，不仅因为代码更易读，更因为任何一个使用该项目的人都知道到哪里去查找关键信息，诸如微控制器的型号及振荡器频率。

正如在本章其余部分所讨论的，使用项目头文件有助于提高移植到不同微控制器上的系统的可靠性。

可移植性

使用项目头文件能够通过将与微控制器相关的主要数据放到一个地方，使代码更容易移植。

此外，该文件中的 `typedef` 语句创建了用于书中所有项目的三个主要的用户自定义类型：

```

typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;

```

因此，在项目中将看到下面这样的代码：

```
tWord Temperature;
```

如果代码被移植到其他地方（比方说一个 16 位平台），仅仅需要改变三个 `typedef` 语句就可以适应新编译程序的变量大小。如果不使用用户自定义类型，代码移植将变得非常复杂并且易于出错。

优缺点小结

- ☺ 项目头文件有助于使代码更加易读，而且使任何一个使用该项目的人知道到哪里去查找关键信息，诸如微控制器的型号和振荡器频率。
- ☺ 项目头文件有助于使代码更加容易移植。

相关的模式和替代解决方案

参见端口头文件。

例子

几乎 CD 上的每个项目的例子都包含项目头文件，可搜索 `Main.H` 文件。

进阶阅读

Chapter 10

使用端口

引言

本章中的第一个模式（端口输入/输出）和管理 8051 微控制器上的数字端口的软件技术有关。

第二个模式（端口头文件）包含了一种设计准则，帮助处理大型项目中不同模块对端口分别访问的要求。具体来说，端口头文件说明了如何将整个项目对端口的访问集中到单个文件中。使用这种技术能够使项目的开发、维护和移植变得更加简单。

同时应注意以下几点：

- 本章不讨论连接到端口上的硬件，相关硬件的细节参见第 7 章和第 8 章的内容。
- 本章不讨论模拟输入与输出，这方面的内容参见第 7 篇。

端口输入/输出

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的软件。

问题

如何编写软件来对 8051 微控制器的端口进行读写？

背景知识

标准 8051 有 4 个 8 位端口，且所有端口都是双向的。也就是说，既可以用来作输入也可以用来作输出。

为了限制芯片的尺寸，一些端口引脚具有复用功能。例如在第 6 章看到的，端口 0、端口 2（和端口 3 的一部分）一起提供地址和数据总线，用来支持对外部存储器的访问。类似地，

端口 3 上的其他两个引脚（引脚 0 和引脚 1）同时提供对片内 USART 的访问（参见第 18 章）。当用作复用功能时，这些引脚不能被用作普通的输入/输出。因此，在一些早期的 8051 系列芯片上，由于只能使用外部存储器，只有端口 1 可以用作通用的输入/输出操作。

这些讨论都针对标准 8051。8051 微控制器上的可用端口数量的变化非常大：精简 8051 有大约 2 个端口，而扩展的 8051 有最多十个端口（参见第 3 章）。尽管有这些区别，所有 8051 系列芯片上的端口都是以同样的方式控制。

解决方案

通过对 8051 端口使用所谓的特殊功能寄存器（SFR）进行软件控制。SFR 是 8 位的锁存器，这意味着写入端口的值被保持住，直到一个新值被写入或芯片复位。

标准 8051 系列上的四个基本的端口，以及其他端口分别由 SFR 代表。这些 SFR 分别被称为 P0、P1、P2、P3 等等。在物理位置上，SFR 是内部 RAM 的高位存储区域。P0 的地址为 0x80，P1 的地址为 0x90，P2 的地址为 0xA0，而 P3 的地址为 0xB0。

如果想从端口读数据，则必须读这些地址。假定使用 C 编译程序，向一个地址写的过程通常通过隐藏在头文件中的 SFR 变量声明来实现。这样，一个 8051 系列芯片的典型 SFR 头文件将包含以下代码：

```
sfr P0 = 0x80;
sfr P1 = 0x90;
sfr P2 = 0xA0;
sfr P3 = 0xB0;
```

在声明了 SFR 变量之后，能够以一种简单的方式向端口写数据。例如，能够以如下方式向端口 1 写数据：

```
unsigned char Port_data;
Port_data = 0x0F;
P1 = Port_data; // Write 00001111 to Port 1
```

类似的，能够以如下方式从端口 1 读数据：

```
unsigned char Port_data;
P1 = 0xFF;           // 将端口设置为'read mode'
Port_data = P1;      // 从端口读
```

注意，为了从一个引脚读数据，必须保证上次向该引脚写的是“1”。因为端口的复位默认值是 0xFF（参见端口复位默认值），看起来写这个值是不必要的，以下版本将侥幸运行正确：

```
unsigned char Port_data;
// 假设复位后没有向端口写入任何数据
// - 危险!!!
Port_data = P1;
```

这些代码的问题在于，在简单的测试程序中它能够工作，这可能会使开发人员有一种错误的安全感。如果今后有人修改了这个程序，增加了向同一个端口的所有或部分引脚写数据的程

序，这些代码通常将不能按照需要工作：

```
unsigned char Port_data;
P1 = 0x00;
...
// 假设复位后没有向端口写入任何数据
// - 不会一直工作
Port_data = P1;
```

一般说来，在程序开始的时候，使用初始化函数将端口设置为一个已知状态。当无法这么做的时候，在读任何的端口引脚之前总是写“1”是安全的，正如第一个例子说明的。

注意，在最后的例子中，假定希望读写整个端口。通常，也许希望控制连接到单个输出引脚上的某个 LED。例如，假定 LED 连到某个 8051 系列微控制器的端口 3 的引脚 0 上，通过控制整个端口可以使这个二极管闪烁，代码如下：

```
P3 = 0xFF;
... // 延迟
P3 = 0x00;
... // 延迟
P3 = 0xFF;
... // 等等
```

也可以使用 C51 编译程序提供的 sbit 变量进行更精确地控制。同时，取决于具体硬件（参见第 7 章），LED 可以用端口引脚输出逻辑 1 或逻辑 0 来点亮。软件可以编写得非常灵活，以很容易地适应今后的硬件变化：

```
#define LED_PORT P3
#define LED_ON 0           // 在这里很容易改变这个逻辑
#define LED_OFF 1
sbit Warning_led = LED_PORT^0; // LED 连接到 4.0
...
Warning_led = LED_ON;
... // 延迟
Warning_led = LED_OFF;
... // 延迟
Warning_led = LED_ON;
... // 等等
```

可靠性和安全性

端口复位默认值

在系统复位之后，各种端口特殊功能寄存器（SFR）的值均被设置为 0xFF。这些事实有非常重要的可靠性和安全性问题。

例如，考虑在端口上连接了一个机械化装置，这个装置在输出“逻辑 1”时被激活，当微控制器复位时，这个机械化装置将被激活。即使在程序一开始的时候将端口输出改变为 0，电

动机也会有短暂地“脉冲输出”。在一些系统中，这将可能导致系统用户或附近的人受到伤害甚至死亡。

因为输出引脚的复位状态为高，所以保证各个所有的有安全性问题的装置都以“低电平有效”的方式连接到微控制器是非常重要的。即，有关的端口引脚输出“0”将激活这些装置。

端口输入/输出以及存储器存取

缺乏经验的 8051 开发人员最常犯的错误是在使用外部存储器的同时（使用存储器的细节参见第 8 章），使用 P0、P2 或 P3 作为普通输入/输出口。

如果使用外部存储器，就不能将 P0 和 P2 用于其他功能，并且必须在写端口 3 时非常小心。

例如，如果正在使用外部存储器，则任何类似这样的语句

```
P3 = AD_data;
```

都将导致灾难。

正确的做法是使用 sbit 变量来保证只写“安全的”端口引脚（细节参见例子，读写位）。

硬件资源

端口输入/输出涉及端口引脚的使用。正如在前面讨论的，如果这些引脚用于输入/输出，那么通常不能将其用于他用。

注意：扩展 8051 提供更多的端口。例如，在 80C515C 上有 8 个 8 位端口，包括标准端口（0~3）和一个复用为模-数转换功能的 8 位端口，以及三个其他端口。

可移植性

一般说来，访问端口和 bit、sbit 关键字不是 ISO/ANSI C 语言的一部分。因此，根据定义，这些代码不全是可移植的。

也就是说，在标准 Keil 编译程序编译下，这些代码能在整个 8051 系列中通用。其他 8051 编译程序提供了类似的功能，只要做少量的程序修改即可编译使用。

优缺点小结

- ⑥ 这个模式提供了灵活而有效的 8051 端口访问，充分利用了内部 BDATA 存储区（在第 6 章中讨论）。
- ⑥ 如前所述，一般说来，访问端口和 bit、sbit 关键字不是 ISO/ANSI C 语言的一部分。因此，根据定义，这些代码不全是可移植的。注意，在非理想的情况下，这个问题无法避免。

相关的模式和替代解决方案

硬件问题

这个模式不涉及外部硬件，关于外部硬件问题的模式参见第 1 篇（特别是第 7 章和第 8

章的内容), 以及第3篇。

中断输入

因为在第1章中讨论的理由, 本书对中断输入的使用非常有限。

例子: 读写字节

源程序清单10.1说明了如何读取连接到8051系列微控制器端口上的一组8个开关, 并且将这些开关设置回送到一个输出端口。使用开关接口(软件)、直接LED驱动和IC缓冲器, 就能够使用这些代码在LED面板上显示开关设置。

注意, 不需要任何硬件来检验这些代码: Keil硬件仿真器(包含在CD上)能够合适的硬件。图10.1显示了一个这样的仿真输出。

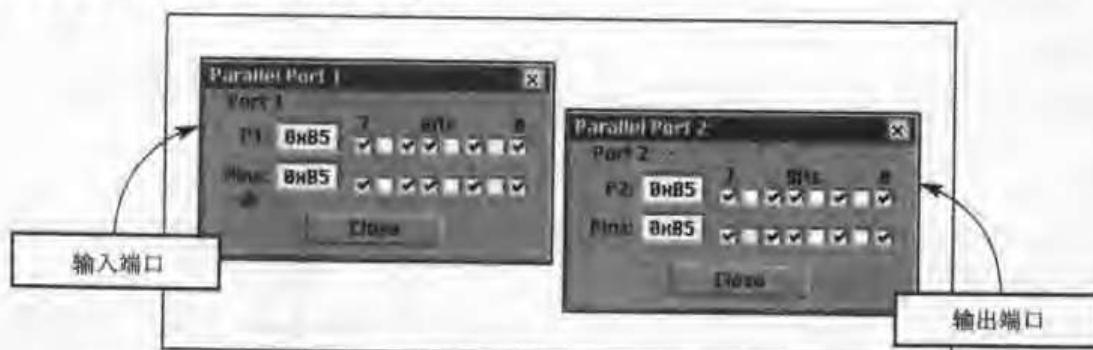


图10.1 使用CD上的Keil硬件仿真器生成的源程序清单10.1中的程序输出

```
/*
 *-----*
 * Main.C
 *-----*
 // 端口输入/输出模式的测试程序
 // 读端口1并将值复制到端口2
 *-----*/
// Main.H详见第9章
#include <Main.H>
/* ..... */
void main (void)
{
    unsigned char Port1_value;
    // 必须将P1设置为reading
    P1 = 0xFF;
    while(1)
    {
        // Read the value of P1
        Port1_value = P1;
        // Copy the value to P2
        P2 = Port1_value;
    }
}
```

```
/*
--文件结束--
*/
```

源程序清单 10.1 一个简单的将 P1 的值复制到 P2 的超级循环应用

例子：读写位

源程序清单 10.1 展示了如何读写整个端口。考虑另一个常见的问题：读写端口上的个别的引脚。这个问题的出现是因为往往端口的各个部分将用于不同的用途。

例如：假设有一个开关连接到端口 1（引脚 3），一个 LED 连接到端口 1（引脚 4），以及其他输入输出装置连接到端口的其他引脚。如何在不破坏其他操作的情况下读引脚 3 和写引脚 4？

可以通过使用按位与、或、取反运算符来实现。这些位操作运算符在桌面系统开发时不常用，但是为嵌入式系统提供了非常有用的数据处理功能。使用这些运算符的一些例子在源程序清单 10.2 中给出了。注意，这个文件是符合 ISO “C”（桌面 C）的，它不能在 Keil 编译程序上运行。

```
/*
-----*
Main.C
-----*
端口输入/输出模式的测试程序说明了位操作运算符的使用
-----*/
#include <stdio.h>
void Display_Byte(const unsigned char);
/* ..... */
int main()
{
    unsigned char x = 0xFE;
    unsigned int y = 0x0A0B;
    printf("%-35s", "x");
    Display_Byte(x);
    printf("%-35s", "1s complement [~x]");
    Display_Byte(~x);
    printf("%-35s", "Bitwise AND [x & 0x0f]");
    Display_Byte(x & 0x0f);
    printf("%-35s", "Bitwise OR [x | 0x0f]");
    Display_Byte(x | 0x0f);
    printf("%-35s", "Bitwise XOR [x ^ 0x0f]");
    Display_Byte(x ^ 0x0f);
    printf("%-35s", "Left shift, 1 place [x <<= 1] ");
    Display_Byte(x <<= 1);
    x = 0xFE; /* Return x to original value */
    printf("%-35s", "Right shift, 4 places [x >>= 4] ");
    Display_Byte(x >>= 4);
    printf("\n\n");
    printf("%-35s", "Display MS byte of unsigned int y");
    Display_Byte((unsigned char) (y >> 8));
```

```

printf("%-35s", "Display LS byte of unsigned int y");
Display_Byte((unsigned char) (y & 0xFF));
return 0;
}
/* -----
void Display_Byte(const unsigned char Ch)
{
    unsigned char i, c = Ch;
    unsigned char Mask = 1 << 7;
    for (i = 1; i <= 8; i++)
    {
        putchar(c & Mask ? '1' : '0');
        c <<= 1;
    }
    putchar('\n');
}
/* -----
---文件结束-----
*/

```

源程序清单 10.2 C 的位操作运算符的演示

源程序清单 10.2 中的程序的输出如下：

X	11111110
1s complement [~X]	00000001
Bitwise AND [x & 0x0f]	00001110
Bitwise OR [x 0x0f]	11111111
Bitwise XOR [x ^ 0x0f]	11110001
Left shift, 1 place [x <<= 1]	11111100
Right shift, 4 places [x >>=4]	00001111
Display MS byte of unsigned int y	00001010
Display LS byte of unsigned int y	00001011

源程序清单 10.3 中说明了嵌入式系统中的一些运算符的使用，将端口 1 上的引脚 X 的输入回送到引脚 Y。

```

/*-----*
Main.C
-----*
端口输入/输出模式的测试程序说明了位操作运算符的使用读写单独的位
注意：两个位在同一个端口上
-----*// Main.H 详见第 9 章
#include <Main.H>
void Write_Bit_P1(unsigned char, bit);
bit Read_Bit_P1(unsigned char);
/* ..... */
void main (void)

```

```
{  
bit x;  
for(;;) // Forever...  
{  
    x = Read_Bit_P1(0); // 读端口 1, 引脚 0  
    Write_Bit_P1(1,x); // 写入端口 1, 引脚 1  
}  
}  
/* ----- */  
void Write_Bit_P1(unsigned char Pin, bit Value)  
{  
    unsigned char p = 1;  
    p <<= Pin; // Left shift  
    // 如果希望这个引脚输出 1  
    if (Value)  
    {  
        P1 |= p; // 位逻辑 OR  
        return;  
    }  
    // 如果希望这个引脚输出 0  
    p = ~p; // 完成  
    P1 &= p; // 位逻辑 AND  
}  
/* ----- */  
bit Read_Bit_P1(unsigned char Pin)  
{  
    unsigned char p = 1;  
    p <<= Pin; // 左 shift  
  
    // 写入一个 1 到这个引脚(设置为 reading)  
    Write_Bit_P1(Pin, 1);  
    return (P1 & p); // 读这个引脚  
}  
/*-----*  
---文件结束-----*  
-----*/
```

源程序清单 10.3 读写位

例子：显示调度器中的错误代码

参见合作式调度器，错误代码通过连接到一个端口上的 LED 段显示。

例子：控制 LCD

参见 LCD 字符面板，通过控制单独的端口引脚向 LCD 显示面板发送数据。

进阶阅读

关于使用 C 的位操作运算符的详细资料能在任何有关 C 语言编程的标准教科书上找到。

端口头文件

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。

问题

如何管理大型项目中的端口访问配置？

背景知识

在典型的嵌入式项目中，可能使用 LCD、键盘，以及一个或多个 LED 来创建用户界面。系统可能有一个（RS-485）串行口连接到另一块电路板上的微控制器，可能要控制一个或多个大功率装置（比方说工业三相电动机）。

系统里的此类（软件）模块每一个独占一个或多个端口引脚。项目可能包括 10~20 个不同的源文件。如何保证修改某个模块内的端口访问时不会影响到其他模块？如何保证将系统移植到其他使用不同端口引脚的平台时简便易行？

简单的端口头文件设计模式解决了这个问题。

解决方案

端口头文件包含了一个简单但是有效的设计准则，帮助你处理一个大型项目中的许多不同模块需要分别访问端口的要求。具体地说，使用端口头文件，将把整个项目中的不同端口访问特性一起放到单个头文件中。使用这种技术能够使项目的开发、维护和移植变得简单。

端口头文件易于理解并且易于使用。

例如：考虑项目中有三个 C 文件（A、B、C），它们每个都需要访问一个或多个端口引脚或整个端口。

文件 A 包括以下部分：

```
// 文件 A  
sbit Pin_A = P3^2;  
...
```

文件 B 包括以下部分：

```
// 文件 B  
#define Port_B = P0;
```

...

文件 C 包括以下部分：

```
// 文件 C  
sbit Pin_C = P2^7  
...
```

在这个版本的代码中，所有对端口的访问都要求分布在多个文件中。作为比较，将所有对端口的访问集中在单个 Port.H 头文件有许多优点：

```
----- Port.H -----  
// 文件 B 用到的端口访问  
#define Port_B = P0;  
// 文件 A 用到的端口访问  
sbit Pin_A = P3^2;  
// 文件 C 用到的端口访问  
sbit Pin_C = P2^7;  
...
```

每个项目文件将包含#include Port.H。

源程序清单 10.4 显示了实际系统中的一个完整的 Port.H 文件的例子

```
-----  
Port.H(V1.00)  
-----  
LCD_KEY 项目（参见第 22 章）的端口头文件  
-----  
// -----Sch51.C-----  
// 如果不需要报告错误，将本行注释  
// #define SCH_REPORT_ERRORS  
#ifdef SCH_REPORT_ERRORS  
// 将显示错误代码的端口  
// 仅当报告错误时使用  
#define Error_port P1  
#endif  
// -----Keypad.C-----  
#define KEYPAD_PORT P0  
  
sbit C1 = KEYPAD_PORT^0;  
sbit C2 = KEYPAD_PORT^1;  
sbit C3 = KEYPAD_PORT^2;  
  
Sbit R1 = KEYPAD_PORT^6;  
Sbit R2 = KEYPAD_PORT^5;  
Sbit R3 = KEYPAD_PORT^4;  
Sbit R4 = KEYPAD_PORT^3;  
// -----LCD_A.C-----  
// 注意：可以使用任意组合的 6 个引脚（任意端口，任意顺序）  
// 注意：[]中的数字是许多 LCD 引脚号  
sbit LCD_D4 = P1^0; // DB4 [11]
```

```

sbit LCD_D5 = P1^1; // DB5 [12]
sbit LCD_D6 = P1^2; // DB6 [13]
sbit LCD_D7 = P1^3; // DB7 [14]

sbit LCD_RS = P1^4; // Display register select output [4]
sbit LCD_EN = P1^5; // Display enable output [6]

// 连接 LCD 上的 Vss[1] 到 GND
// 连接 LCD 上的 Vcc[2] 到 +5V
// 连接 LCD 上的 Vo[3] 到 GND
// 连接 LCD 上的 RW[4] 到 GND
// ----- LED_Flas.C -----
// LED 一端接 +5V，另一端通过串联合适的电阻到这个引脚
// 详见第 7 章
sbit LED_pin = P1^5;
/*-----文件结束-----*/

```

源程序清单 10.4 实际项目的端口头文件 (Port.H) 的例子，使用由键盘和液晶显示组成的用户界面

硬件资源

没有硬件资源问题。

可靠性和安全性

端口头文件虽然简单，但是能够提高可靠性和安全性，因为它避免了端口引脚之间潜在的冲突。特别是在项目的维护阶段，当要求开发人员（未必涉及到原始设计）改变代码时更是这样。

可移植性

端口头文件本身是可移植的，它能用于任何一种微控制器，而不是只与 8051 系列有联系。端口头文件通过在一个地点访问系统的所有端口访问要求，提高了可移植性。

优缺点小结

☺ 端口头文件既简单又有效——使用它！

相关的模式和替代解决方案

参见项目头文件。

例子：LED 柱形图显示

假设希望测试一个涉及模数转换器的系统。具体地说，想在连接到 8051 芯片端口 1 的一

组8个LED上显示读取的电压(图10.2)。

创建这个系统需要的主要软件文档如下(源程序清单10.5~源程序清单10.8)。CD中包含了这个项目的所有源文件。

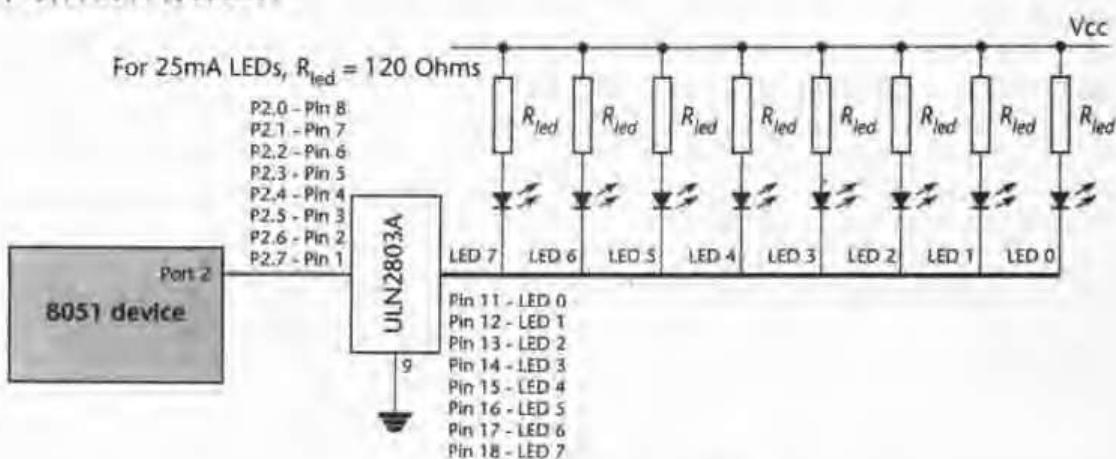


图10.2 LED柱形图显示的硬件

```
/*
Main.H (v1.00)

柱形图项目的项目头文件 (参见第9章)
*/
#ifndef _MAIN_H
#define _MAIN_H
// 每个项目都需要修改这一段
// 必须在这里包含相应的微控制器的头文件
#include <AT89x52.h>
// Must include oscillator / chip details here if delays are used
// 振荡器/谐振器的频率 (Hz), 例如 (11059200u1)
#define OSC_FREQ (12000000UL)
// 每个指令的振荡周期数 (6或12)
#define OSC_PER_INST (12)
// 以下部分不需要修改
typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;
// 杂项宏定义
#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif
```

```

#define RETURN_NORMAL (bit) 0
#define RETURN_ERROR (bit) 1
// -----
// 中断
// 参见第 12 章
// -----
// 通用的 8051 定时器中断 (用于大多数调度器)
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5
// 其他的中断 (用于共享时钟调度器)
#define INTERRUPT_UART_Rx_Tx 4
#define INTERRUPT_CAN_c515c 17
// -----
// 错误代码
// 参见第 13 章
// -----
#define ERROR_SCH_TOO_MANY_TASKS (1)
#define ERROR_SCH_CANNOT_DELETE_TASK (2)

#define ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK (0xAA)
#define ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER (0xAA)

#define ERROR_SCH_ONE_OR_MORE_SLVES_DID_NOT_START(0xA0)
#define ERROR_SCH_LOST_SLAVE (0x80)

#define ERROR_I2C_WRITE_BYTE_AT24C64 (11)
#define ERROR_I2C_READ_BYTE_AT24C64 (12)
#define ERROR_I2C_WRITE_BYTE (13)
#define ERROR_I2C_READ_BYTE (14)

#define ERROR_USART_TI (21)
#define ERROR_USART_WRITE_CHAR (22)
#endif
/ =====

```

源程序清单 10.5 创建 LED 柱形图显示所需的一个简单程序库的一部分

```

/*-----*
Port.H (v1.00)
-----*/
// 柱形图项目的项目头文件
/*-----*/
// ----- Bargraph.C -----
// LED 一端接+5V，另一端通过串联合适的电阻到这个引脚
// 参见第 7 章
// 如果需要，8 个端口引脚可以使用不同的端口
sbit Pin0 = P1^0;
sbit Pin1 = P1^1;
sbit Pin2 = P1^2;

```

```
sbit Pin3 = P1^3;
sbit Pin4 = P1^4;
sbit Pin5 = P1^5;
sbit Pin6 = P1^6;
sbit Pin7 = P1^7;
/*-----*
-----文件结束-----
*-----*/
```

源程序清单 10.6 创建 LED 柱形图显示所需的一个简单程序库的一部分

```
/*-----*
Main.c (v1.00)

柱形图显示的演示程序
*-----*/
#include "Main.h"
#include "Bargraph.h"
// -----声明公用变量-----
extern tBarGraph Data_G;
/* ..... */
/* ..... */
void main(void)
{
    tWord x;
    BARGRAPH_Init();
    while(1)
    {
        if (++x == 1000)
        {
            x = 0;
            Data_G++;
        }
        BARGRAPH_Update();
    }
}
/*-----*
-----文件结束-----
*-----*/
```

源程序清单 10.7 创建 LED 柱形图显示所需的一个简单程序库的一部分

```
/*-----*
Bargraph.c (v1.00)

// 简单的柱形图库。参见第 10 章
*-----*/
#include "Main.h"
#include "Port.h"
#include "Bargraph.h"
```

```
// ----- 声明公用变量-----
// The data to be displayed
tBargraph Data_G;
// -----私有常数-----
#define BARGRAPH_ON (1)
#define BARGRAPH_OFF (0)
// -----私有变量-----
// 这些变量存储门限值
// 用来刷新显示
static tBargraph M9_1_G;
static tBargraph M9_2_G;
static tBargraph M9_3_G;
static tBargraph M9_4_G;
static tBargraph M9_5_G;
static tBargraph M9_6_G;
static tBargraph M9_7_G;
static tBargraph M9_8_G;
/*-----*
BARGRAPH_Init()
准备柱形图显示
-----*/
void BARGRAPH_Init(void)
{
    Pin0 = BARGRAPH_OFF;
    Pin1 = BARGRAPH_OFF;
    Pin2 = BARGRAPH_OFF;
    Pin3 = BARGRAPH_OFF;
    Pin4 = BARGRAPH_OFF;
    Pin5 = BARGRAPH_OFF;
    Pin6 = BARGRAPH_OFF;
    Pin7 = BARGRAPH_OFF;
    // 使用线性刻度显示数据
    // 记住：9个可能的输出状态
    // 一次计算完毕
    M9_1_G = (BARGRAPH_MAX - BARGRAPH_MIN) / 9;
    M9_2_G = M9_1_G * 2;
    M9_3_G = M9_1_G * 3;
    M9_4_G = M9_1_G * 4;
    M9_5_G = M9_1_G * 5;
    M9_6_G = M9_1_G * 6;
    M9_7_G = M9_1_G * 7;
    M9_8_G = M9_1_G * 8;
}
/*-----*
BARGRAPH_Update()
刷新柱形图显示
-----*/
void BARGRAPH_Update(void)
{
```

```
tBargraph Data = Data_G - BARGRAPH_MIN;
Pin0 = ((Data >= M9_1_G) == BARGRAPH_ON);
Pin1 = ((Data >= M9_2_G) == BARGRAPH_ON);
Pin2 = ((Data >= M9_3_G) == BARGRAPH_ON);
Pin3 = ((Data >= M9_4_G) == BARGRAPH_ON);
Pin4 = ((Data >= M9_5_G) == BARGRAPH_ON);
Pin5 = ((Data >= M9_6_G) == BARGRAPH_ON);
Pin6 = ((Data >= M9_7_G) == BARGRAPH_ON);
Pin7 = ((Data >= M9_8_G) == BARGRAPH_ON);
}
/*-----*-
---文件结束 -----
-*-----*/
```

源程序清单 10.8 创建 LED 柱形图显示所需的一个简单程序库的一部分

进阶阅读

Chapter 11

延 迟

引言

产生精确的延迟是许多嵌入式系统的关键要求。

本章将讨论用来提供这样功能的两种不同的技术：

- 硬件延迟能够通过使用一个片内定时器来产生精确的延迟，特别适合于产生大约 0.1ms 或更长的延迟。
- 软件延迟是一种不需要硬件资源的简单技术，它是最灵活的延时机制，特别适合于产生较短的延迟（以微秒为单位）或没有定时器资源可用的场合。

硬件延迟

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的软件。

问题

必须在做出某种行为之前等待固定长度的时间（以毫秒为单位）。

背景知识

解决方案

所有的 8051 系列芯片都有至少两个 16 位定时/计数器：定时器 0 和定时器 1。这些定时器可用于产生精确的延迟。

首先介绍有关这些定时器的简要知识。

定时器0和定时器1

定时器0和定时器1有许多共同之处，我们将把它们放在一起讨论。

为了了解这些定时器是如何运行的，需要首先介绍TCON特殊功能寄存器（表11.1）。

表11.1 TCON特殊功能寄存器

Bit	7(msb)	6	5	4	3	2	1	0(lsb)
Name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

注意：灰色区与定时器无关。

各个位具有以下功能：

TF1 定时器1溢出标志

当定时器1溢出时由硬件设置。

（当处理器跳转到中断程序时由硬件清除。）

TR1 定时器1运行控制位

由软件设置/清除来打开或者关闭定时器1。

TF0 定时器0溢出标志

当定时器0溢出时由硬件设置。

（当处理器跳转到中断程序时由硬件清除。）

TR0 定时器0运行控制位

由软件设置/清除来打开或者关闭定时器0。

注意，定时器的溢出可用于产生中断。在硬件延迟的代码中没有使用这种功能，但是将在各种调度器模式中使用（细节参见第3篇和第6篇）。

为了禁止中断的产生，可以使用C语句：

ET0 = 0; //不使用中断(定时器0)

ET1 = 0; //不使用中断(定时器1)

此外，还必须介绍TMOD特殊功能寄存器（表11.2）。

表11.2 TMOD特殊功能寄存器

Bit	7(msb)	6	5	4	3	2	1	0(lsb)
Name	Gate	C/T	M1	M0	Gate	C/T	M1	M0

Timer 1

Timer 0

在TMOD中，首先需要注意的是使用M1和M0位可以（分别为每个定时器）设置3种主要的运行模式。本书将只涉及模式1和模式2。定时器0和定时器1中以同样的方式运行如下：

Mode 1 (M1 = 0; M0 = 1)

16位定时/计数器（手动重装）。

Mode 2 (M1 = 1; MO=0)

8位定时/计数器（8位自动重装）。^①

TMOD的其余位有以下用途：

GATE 门控制

当置为1时，只有在INTx引脚为高而且TRx控制位被置为1时，定时/计数器x才被使能。当置为0时，只要TRx控制位被置为1，定时器x就被使能。

C/T 计数器或定时器选择位

置为1时为计数器运行方式（从Tx引脚输入）。

置为0时为定时器运行方式（从内部系统时钟输入）。

最后，在讨论如何使用硬件来产生延迟之前，必须了解另外还有两个寄存器与每个定时器有关：这些寄存器被称为TLO和THO、TL1和TH1。正如下面介绍的，L和H分别指的是低字节和高字节。

使用定时器0和定时器1产生延迟

为了了解这些寄存器如何一起工作，下面将讨论一个具体的例子（源程序清单11.1）。

```
// 将定时器0配置为16位定时器
TMOD &= 0xF0;           // 清除所有有关T0的位(T1不变)
TMOD |= 0x01;            // 设置所需的T0的位(T1不变)
ET0 = 0;                 // 不使用中断
TH0 = 0;                 // 定时器0初始值(高位字节)
TL0 = 0;                 // 定时器0初始值(低位字节)
TF0 = 0;                 // 清除溢出标志
TR0 = 1;                 // 启动定时器0
while (TF0 == 0);        // 循环直到定时器0溢出(TF0 == 1)
TR0 = 0;                 // 停止定时器0
```

源程序清单11.1 使用定时器0产生简单的硬件延迟

在源程序清单11.1中，这些代码设置定时器0为模式1（16位定时器），不用门控：

```
TMOD &= 0xF0;           // 清除所有有关T0的位(T1不变)
TMOD |= 0x01;            // 设置所需的T0的位(T1不变)
```

正如前面所讨论的，禁止产生中断：

```
ET0 = 0;                 // 不使用中断
```

然后向定时器的寄存器加载初始时间值（将在本模式的其余部分进一步讨论）：

```
TH0 = 0;                 // 定时器0初始值(高位字节)
TL0 = 0;                 // 定时器0初始值(低位字节)
```

然后清除定时器标志，并且启动定时器计时：

^① 有关自动和手动定时器重装之间区别的讨论参见第13章。

```
TF0 = 0; // 清除溢出标志
TR0 = 0; // 启动定时器 0
```

早期 8051 的定时器每 12 个振荡周期加 1（将在“可移植性”一节介绍一些例外情况）。当（16 位）定时器溢出时，即从 65535 加 1，定时器标志（TF1）将被置 1。此外，正如所介绍的，这个溢出可用于产生中断。在这里不使用这个功能。

这是非常有用的特性，只通过改变定时器中保存的初始值，就可以指定溢出发生之前的振荡周期数，从而产生较短的延迟。

以背景知识中介绍的材料为基础，硬件延迟通常按以下形式计算：

- 计算定时器所需的初始值。
- 将这个值加载到定时器 0 或定时器 1 中。
- 启动定时器。
- 定时器将按照由振荡器频率确定的速度增加，而不需要软件的干预。等待定时器到达它的最大值后“翻转”。
- 定时器的翻转将改变标志变量的值，从而标志延迟的结束。
- 对于一个运行在 12MHz、12 个振荡周期/指令的 8051，使用 16 位定时器能够生产的最长延迟是 65ms。如果需要更长的延迟，可以重复这个过程。

总的说来，这是一种强大而可靠的技术，可用于产生精度相当高的可重复延迟。在本模式的后面将给出一个详细的代码例子。

为什么不使用定时器 2？

在很多情况下，正如在第 3 章看到的，现代的 8051 系列芯片基于稍晚一些的 8052 体系结构。这样的芯片包括另一个更强大的定时器（逻辑上称做定时器 2）。

定时器 2 可用于产生延迟（和本模式介绍的使用定时器 0 和定时器 1 的方式几乎相同），然而通常这种使用资源的方式是不合理的。这是因为定时器 2 是一个 16 位自动重装定时器。自动重装的特性对于产生延迟来说没有价值，但是对于贯穿于本书大部分内容所使用的调度器来说，它是一个理想的“时标”源。^②

硬件资源

硬件延迟需要非独占性地使用一个定时器。对这样的资源往往有竞争请求，因为它们对于驱动调度器而言是必不可少的，而且经常用定时器来产生超时（参见硬件超时）、脉宽调制（参见 3 级脉宽调制）、脉冲频率调制（参见硬件脉冲频率调制）和脉冲计数（参见硬件脉冲计数）。

可靠性和安全性

在“硬件延迟”模式中讨论的技术通常比基于软件的延迟更易于移植，而且更加精确。但

^② 这方面的详细信息参见第 13 章以及第 14 章的“合作式调度器”一节。

是调用延迟函数以及为定时器加载初始计数值需要时间，如果没有仔细地将这些因素考虑在内，就无法产生精确的时间延迟。

如果通过多次调用延迟函数来产生延迟，这些因素的影响将增加。使用这种技术产生的较长延迟的误差更大。

可移植性

这里讨论两个主要的可移植性问题。

定时器增加速度的差异

在早期的 8051（以及大多数现在的 8051）中，定时器 0 和定时器 1 每 12 个振荡器周期加 1。即使使用 12MHz 晶体振荡器时为 1MHz。一些更新的 8051 芯片使用 6、4 或 1 个振荡器周期。必须核对数据手册以保证计算初始重装值时考虑了这些差异。

注意：（在后面列出的）程序库代码本身的可移植性是很高的，因为它使用了项目头文件提供的信息。

在 8051 系列内移植

8051 系列中可用的定时器是有限的。小心地使用定时器有助于使代码更易于移植。

例如：在本书给出的许多系统中，（如果有的话）将使用定时器 2 来驱动调度器（参见第 13 章）。此外，在一些系统中，需要通过使用定时器 1 为串行网络产生波特率。因此，如果延迟代码基于定时器 0 则将很容易移植。

注意，当需要使用调度器和串行连接，而所选定的微控制器没有定时器 2 可用时，必须使用所有可用的定时器：T0 用于调度器，T1 用于产生波特率。于是将不得不使用软件延迟来产生所有需要的延迟。同时还应注意，在很多情况下，使用调度器能够消除大多数对延迟计算的需要。

最后需要注意的是：除定时器 2 之外，一些扩展 8051 具有一个额外的内部定时器，设计用作内部波特率发生器。这将使定时器 1 可以用作其他功能，诸如产生延迟（详细说明参见第 3 章）。

移植到 8051 系列以外的芯片

与 8051 系列相似，大多数微控制器都有片内定时器。只要有这样的定时器，使用这个模式就没有什么大问题。如果所选定的微控制器没有片内定时器，那么软件延迟模式提供了一个替代方案。

优缺点小结

- ⑤ 这些基本的时间延迟技术有很多的优点：它们非常简单，只用几行代码就可以实现。因此它们被广泛地使用，经常用于不特别强调定时精度的应用场合。
- ⑥ 不适于产生非常短的延迟。建议的替代方案参见“相关的模式和替代解决方案”一

节。

- ⑧ 由于需要人工重装初始定时值，所得到的延迟可能不如期望的那样精确。例如，当试图通过调用 20 次 50ms 的延迟来产生 1s 的延迟时，需要特别关注这种误差，因为将不可能做到精确。不要试图使用硬件延迟来实现实时时钟！
- ⑨ 需要访问一种重要的硬件资源（定时器）。
- ⑩ 计时的可移植性不太好，即使是 8051 系列内的不同芯片，晶振频率与指令周期频率的关系也不尽相同（然而需要注意的是，下列代码针对各种延迟解决了这个问题）。
- ⑪ 正如这里实现的，处理器停下来等待定时器溢出。当处理器的性能有限时，这将是不可接受的解决方案。使用调度器（参见第 13 章）往往能够降低这种 CPU 的时间浪费。

相关的模式和替代解决方案

本模式中给出的代码用来产生 N 毫秒时段的延迟，在许多系统中这是一个关键要求。如果这不能满足需要，那么下面有一些替代方案：

- 硬件超时能非常有效地用来产生从 10 微秒到大约十几毫秒的延迟。
- 对于小于 10 微秒的延迟，硬件延迟和硬件超时都不合适，两者都使用定时器 0/定时器 1。例如，对于早期的 12MHz 的 8051，理论上最小的时间增量是 1 微秒（即振荡器频率的 1/12）。然而，延迟往往小于调用延迟函数、设置，以及启动定时器的时间。一般说来，小于 10 微秒的延迟最好由软件来实现，参见“软件延迟”一节（以及第 15 章的“循环超时”一节）。
- 延迟将浪费 CPU 的时间。如果可能的话，最好避免使用它们。一种不需要延迟的替代方案可以用于许多环境中，参见第 14 章的“合作式调度器”一节。

例子：通用的延迟代码

闪烁的 LED 能够用来引起人们注意某个警告信息或错误状态。它还可以被用作一种节省功耗的方法。当然，实现这种功能有许多不同的方式。这里，说明硬件延迟模式的使用。

硬件

假定 LED（或类似装置，比如蜂鸣器）使用正逻辑连接到 8051 微控制器的端口 1（P1.2），即+5V 点亮 LED（图 11.1）。

软件

这里使用一些通用的延迟代码。通过项目头文件（Main.H）以及使用一些相关的 C 语言预处理指令，能够“自动”计算各种不同的硬件及振荡器组合的初始定时值。

项目需要的主要文件如下（源程序清单 11.2~源程序清单 11.4）。同样地，CD 上包括所有的文件。

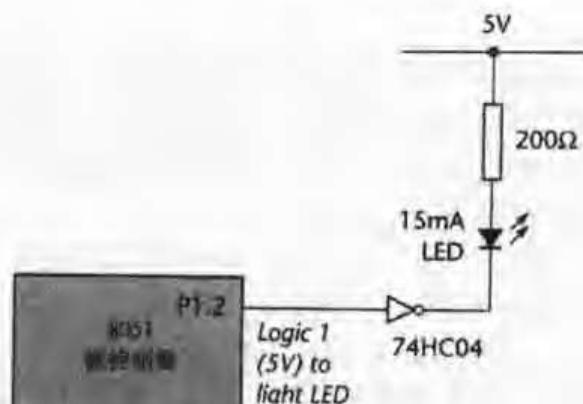


图 11.1 LED 硬件

注意，更详细的资料参见 IC 缓冲器一节。

```
/*
Main.H (v1.00)

// 项目DELAY_H (参见第11章) 的“项目头文件”(参见第9章)
*/
#ifndef _MAIN_H
#define _MAIN_H
//-
// 每个项目都需要修改这一段
//-
// 必须在这里包含相应的微控制器的头文件
#include <reg52.h>
// 这里包括振荡器/芯片的详细资料
// (使用通用的延迟/超时时必需)
// -
// 振荡器 / 谐振器的频率 (Hz), 例如 (11059200UL)
#define OSC_FREQ (12000000UL)
// 每个指令的振荡周期数 (4、6 或 12)
// 12-早期的 8051/8052 以及许多的新型芯片
// 6-各种 Infineon 以及 Philips 的芯片, 等等
// 4- Dallas 的芯片, 等等
// -
// 使用 Dallas 的芯片时小心
// -除非修改 CKCON, 否则定时器默认为 12 个振荡器周期/指令
// -如果在 Dallas 的芯片上使用通用的代码, 则在这里定义 12
#define OSC_PER_INST (12)
//-
// 以下部分不需要修改
//-
typedef unsigned char tByte;
```



```

typedef unsigned int tWord;
typedef unsigned long tLong;
// 杂项宏定义
#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif
#define RETURN_NORMAL (bit) 0
#define RETURN_ERROR (bit) 1
-----
// 中断
// 参见第13章的内容
-----
// 通用的8051定时器中断(用于大多数调度器)
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5
// 其他的中断(用于共享时钟调度器)
#define INTERRUPT_UART_Rx_Tx 4
#define INTERRUPT_CAN_c515c 17
-----
// 错误代码
// 参见第14章的内容
-----
#define ERROR_SCH_TOO_MANY_TASKS (1)
#define ERROR_SCH_CANNOT_DELETE_TASK (2)
#define ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK (3)
#define ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER (3)
#define ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START (4)
#define ERROR_SCH_LOST_SLAVE (5)
#define ERROR_SCH_CAN_BUS_ERROR (6)
#define ERROR_I2C_WRITE_BYTE_AT24C64 (11)
#define ERROR_I2C_READ_BYTE_AT24C64 (12)
#define ERROR_I2C_WRITE_BYTE (13)
#define ERROR_I2C_READ_BYTE (14)
#define ERROR_USART_TI (21)
#define ERROR_USART_WRITE_CHAR (22)
#endif
/*-----*
---文件结束---
*-----*/

```

源程序清单11.2 通用的延时代码(硬件延时)例子的一部分

```

/*-----*
Main.C (v1.00)
*-----*/

```

```
-----硬件延迟程序库的简单的测试程序-----*/
#include "Main.h"
#include "Delay.h"
#include "LED_Flas.h"
void main(void)
{
    LED_Flash_Init();
    while (1)
    {
        LED_Flash_Update();
        Hardware_Delay_T0(1000);
    }
}
-----文件结束-----*/
-----
```

源程序清单 11.3 通用的延迟代码（硬件延迟）例子的一部分

```
-----*----- Delay_T0.C (v1.00)
-----*----- 基于 T0 的简单的硬件延迟-----*/
#include "Main.H"
// ----- 私有常数 -----
// 用于简单的（硬件）延迟的定时器预置值
// 定时器是 16 位，手动重装（“单次有效”）
// 注意：这些值是可移植的，但计时是“近似的”
// 如果需要精确的计时则必须经过复核
//
// 定义用于 1ms 延迟的定时器 0/定时器 1 的重装值
// 注意：已考虑到函数调用等等的开销而做出调整
#define PRELOAD01 (65536 - (tWord)(OSC_FREQ / (OSC_PER_INST * 1063)))
#define PRELOAD01H (PRELOAD01 / 256)
#define PRELOAD01L (PRELOAD01 % 256)
/*
Hardware_Delay_T0()
产生（大约）N 毫秒延迟的函数
使用定时器 0（很容易移植到定时器 1）
*/
void Hardware_Delay_T0(const tWord N)
{
    tWord ms;
    // 将定时器 0 配置为 16 位定时器
    TMOD &= 0xF0; // 清除所有有关 T0 的位 (T1 不变)
    TMOD |= 0x01; // 设置所需的 T0 的位 (T1 不变)
    ET0 = 0; // 不使用中断
}
```

```
// 每个循环的延迟值“大约”是 1ms
for (ms = 0; ms < N; ms++)
{
    TH0 = PRELOAD01H;
    TL0 = PRELOAD01L;
    TF0 = 0;           // 清除溢出标志
    TR0 = 1;           // 启动定时器 0
    while (TF0 == 0); // 循环直到定时器 0 溢出 (TF0 == 1)
    TR0 = 0;           // 停止定时器 0
}
}

/*
-----文件结束-----
*/

```

源程序清单 11.4 通用的延迟代码（硬件延迟）例子的一部分

本项目在 Keil 硬件模拟器上运行的输出如图 11.2 所示。注意，得到的延迟值在要求值的 0.001% 范围之内。

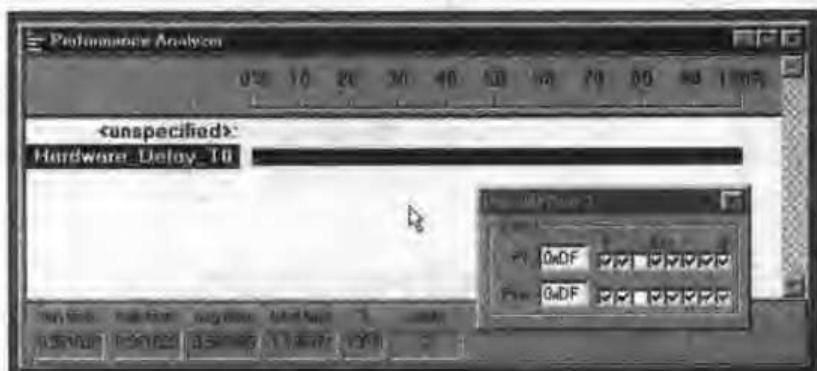


图 11.2 运行在 Keil 硬件模拟器上的硬件延迟例子的输出

进阶阅读

软件延迟

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的软件。

问题

如何不使用任何硬件（定时器）资源产生简单的延迟？

背景知识

背景资料参见硬件延迟。

解决方案

假设想要使连接到 8051 微控制器的端口 1 上的 LED 以 2s 的周期闪烁（1s 亮，1s 灭，无限循环）。需要的基本程序结构基于一个超级循环，如下：

```
...
while (1)
{
    P1 = 0xFF;
    //延迟一秒
    P1 = 0x00;
    //延迟一秒
}
```

如果希望实现这种延迟，然而没有定时器可用（参见硬件延迟），可以使用一种“软件延迟”技术，实现如下：

```
Loop_Delay()
{
    unsigned int x;
    for (x=0; x <= 65535; x++);
}
```

通过使用示波器或软件仿真程序能够测量得到的脉冲频率。如果发现延迟不够长，可以通过添加更多的层来延长，如 Longer_Loop_Delay() 所示：

```
Longer_Loop_Delay()
{
    unsigned int x;
    unsigned int y;
    for (x=0; x<=65535; x++)
    {
        for (y=0; y<=65535; y++);
    }
}
```

硬件资源

与硬件延迟不同，软件延迟不使用定时器资源。注意，然而 CPU 花费在计算延迟上的时间被浪费。在许多情况下，使用调度器（参见第 13 章）能够避免 CPU 时间的浪费。

可靠性和安全性

软件延迟不适用于需要精确定时的系统。

可移植性

软件延迟甚至能用于没有内部定时器的微控制器/微处理器。然而，得到的延迟的精度随硬件和软件的差别变化非常大。

优缺点小结

- ◎ 软件延迟可用于生成非常短的延迟。
- ◎ 软件延迟不需要硬件定时器。
- ◎ 软件延迟适用于所有的微控制器。
- ⊗ 很难生成精确的时延。
- ⊗ 如果决定使用不同的处理器，改变时钟频率，或者哪怕是改变编译程序的优化设置，也必须修改循环。

相关的模式和替代解决方案

在大多数情况下，最好避免使用延迟。一种不需要延迟的替代方案可以用于许多环境中，参见第 14 章的“合作式调度器”一节。

如果确实需要延迟，那么硬件延迟往往是最好的可选方案。

例子：在 I²C 程序库中产生 5 微秒的延迟

正如将在第 23 章讨论的，I²C 串行协议的软件实现需要短的延迟。在源程序清单 11.5 中说明了如何使用软件产生大约 5 微秒的延迟用于这样的程序库。

```
/*-----*
_I2C_Delay()
---一个短的软件延迟
调整为最少 5.425 微秒，以连接标准的 I2C 设备，比这更长的延迟仍可以正常工作。
现代的设备也可以使用较短的延迟
注意：使用硬件延迟无法做到！
-----*/
void _I2C_Delay(void)
{
    int x;
    x++;
    x++;
}
```

源程序清单 11.5 产生非常短的延迟

例子：闪烁 LED

这里重提前面“硬件延迟”一节提到的“闪烁 LED”的例子，进一步说明软件延迟模式。

为了控制 LED 的闪烁，使用一个包含软件延迟和“闪烁 LED”函数的死循环。

假定使用 12MHz 振荡器频率和 12 个振荡周期/指令的标准 8051 芯片（详细说明参见第 4 章中的内容）。

项目需要的主要文件如下（源程序清单 11.6~源程序清单 11.11）。同样地，CD 上包括所有的文件。

```
/*
Main.C (v1.00)

-----软件延迟模式的简单测试程序-----
*/
#include "Main.h"
#include "Loop_Del.h"
#include "LED_Flas.h"
void main(void)
{
    LED_Flash_Init();
    while (1)
    {
        LED_Flash_Update();
        Loop_Delay(1000);
    }
}
-----文件结束-----
*/
```

源程序清单 11.6 软件延迟（闪烁 LED）例子的一部分

```
/*
Loop_Del.H (v1.00)

-----详细资料参见 Loop_Del.C -----
*/
void Loop_Delay(const unsigned int);
-----文件结束-----
*/
```

源程序清单 11.7 软件延迟（闪烁 LED）例子的一部分

```
/*
Loop_Del.c (v1.00)

-----软件延迟模式的实现-----
*/
void Loop_Delay(const unsigned int);
-----Loop_Delay()-----
*/
```

延迟随参数而变，在12MHz（12个振荡周期）的8051上，参数大致是以毫秒为单位的延迟
需要根据应用的需要调整！

```
-----*/
void Loop_Delay(const unsigned int DELAY)
{
    {
        unsigned int x, y;
        for (x = 0; x <= DELAY; x++)
        {
            for (y = 0; y <= 120; y++);
        }
    }
/*
---文件结束---
*/

```

源程序清单11.8 软件延迟（闪烁LED）例子的一部分

```
-----*
LED_flas.H (v1.00)
-----
-详细资料参见 LED_flas.C
-----*/
// -----公用的函数原型-----
void LED_Flash_Init(void);
void LED_Flash_Update(void);
/*
---文件结束---
*/

```

源程序清单11.9 软件延迟（闪烁LED）例子的一部分

```
-----*
LED_flas.C (v1.00)
-----
简单的“闪烁LED”的测试函数
-----*/
#include "Main.h"
#include "LED_flas.h"
// ----- Port pins //端口引脚-----
// LED一端接+5V，通过串联合适的电阻到这个引脚
// [详细资料参见第7章]
sbit LED_Pin = P1^2;
/*-----公用变量定义-----*/
static bit LED_state_G;
/*
LED_Flash_Init()
- See below.
*/
void LED_Flash_Init(void)
```

```

{
LED_state_G = 0;
}
/*-----*/
LED_Flash_Update()
在指定端口引脚上闪烁 LED (或产生脉冲给蜂鸣器, 等等)。
必须按需要的闪烁频率的两倍计时。这样, 对于 1Hz 的闪烁 (0.5s 亮, 0.5s 灭)
必须以 2Hz 计时
/*-----*/
void LED_Flash_Update(void)
{
// 使 LED 从灭变亮 (反之亦然)
if (LED_state_G == 1)
{
LED_state_G = 0;
LED_pin = 0;
}
else
{
LED_state_G = 1;
LED_pin = 1;
}
}
/*-----文件结束-----*/
/*-----*/

```

源程序清单 11.10 软件延迟 (闪烁 LED) 例子的一部分

```

/*-----*
Main.H (v1.00)

// 项目 S_Delay 的“项目头文件”(参见第 9 章)
/*-----*/
#ifndef _MAIN_H
#define _MAIN_H
//-----
// 每个项目都需要修改这一段
//-----
// 必须在这里包含相应微控制器的头文件
#include <reg52.h>
// 这里包括振荡器/芯片的详细资料
//(在使用通用的延迟/超时时必需)
// -
// 振荡器/谐振器的频率 (Hz), 例如 (11059200UL)
#define OSC_FREQ (12000000UL)
// 振荡数/指令 (如:1、4、6 或 12)
// 12-早期的 8051/8652 以及许多新型芯片
/*-----*/

```

```
// 6-各种 Infineon 以及 Philips 的芯片，等等
// 4-Dallas 的芯片，等等
//
// 使用 Dallas 的芯片时小心
// 除非修改 CKCON，否则定时器默认为 12 个振荡器周期/指令
// 如果在 Dallas 的芯片上使用通用的代码，则在这里定义 12
#define OSC_PER_INST (12)
//-----
// 以下部分不需要修改
//-----
typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;
// 杂项宏定义
#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif
#define RETURN_NORMAL (bit) 0
#define RETURN_ERROR (bit) 1
//-----
// 中断
// 参见第 12 章的内容
//-----
// 通用的 8051 定时器中断（用于大多数调度器）
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5
// 其他的中断（用于共享时钟调度器）
#define INTERRUPT_UART_Rx_Tx 4
#define INTERRUPT_CAN_c515c 17
//-----
// 错误代码
// 参见第 13 章的内容
//-----
#define ERROR_SCH_TOO_MANY_TASKS (1)
#define ERROR_SCH_CANNOT_DELETE_TASK (2)
#define ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK (3)
#define ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER (3)
#define ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START (4)
#define ERROR_SCH_LOST_SLAVE (5)
#define ERROR_SCH_CAN_BUS_ERROR (6)
#define ERROR_I2C_WRITE_BYTE_AT24C64 (11)
#define ERROR_I2C_READ_BYTE_AT24C64 (12)
#define ERROR_I2C_WRITE_BYTE (13)
#define ERROR_I2C_READ_BYTE (14)
#define ERROR_USART_TI (21)
```

```
#define ERROR_USART_WRITE_CHAR (22)
#endif
/*-----文件结束-----*/
*/
```

源程序清单 11.11 软件延迟（闪烁 LED）例子的一部分

图 11.3 显示了这个系统在 Keil 硬件模拟器上的运行结果。正如从屏幕快照上看到的，在这些参数和编译程序设置下的延迟大约是 1s。

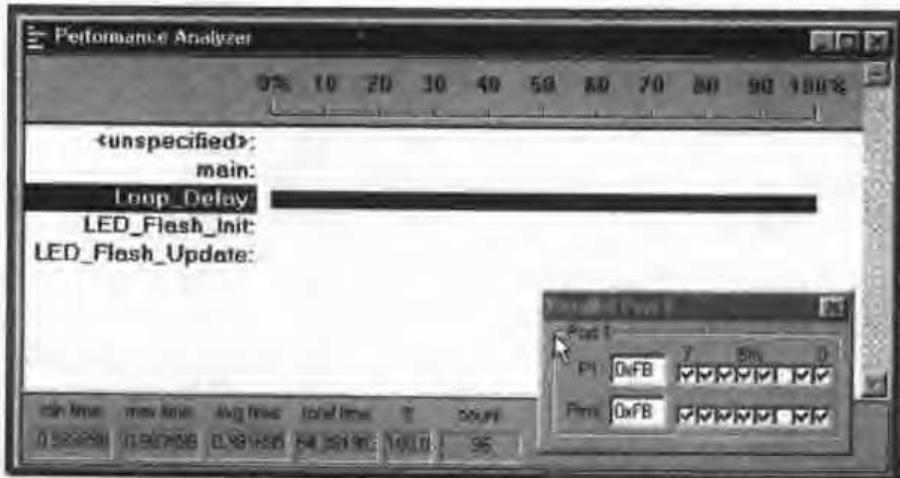


图 11.3 运行在 Keil 硬件模拟器上的软件延迟例子的输出

进阶阅读

Chapter 12

看门狗

引言

假设有一只饥饿的狗正在看守着一所房子（图 12.1），而有人要闯入。如果这个强盗的同谋以 2 分钟的间隔不停地向看门的狗扔肉块，那么这只狗将忙于吃食物而忽视保卫工作，因此将不会吠叫。然而，如果同谋扔完了肉或由于其他原因忘记喂狗，狗将开始吠叫，从而惊动邻居、房屋的主人或警察。



图 12.1 “看门狗”比喻的起源

计算机化的“看门狗定时器”遵循同样的基本方法。非常简单，如果每隔一定间隔不刷新定时器，它将溢出。在大多数情况下，定时器的溢出将复位系统。即使经过仔细规划和设计，嵌入式系统也有可能由于出乎意料的问题而“死机”，这种看门狗就是用来处理此类情况的。看门狗可用于在特定的情况下从这种状态恢复。

硬件看门狗模式讨论了如何在嵌入式系统中更加有效地应用这种技术。

硬件看门狗

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 需要为该系统设计相应的软件。

问题

如果系统由于出乎意料的软件或硬件错误而“死机”，如何保证系统将自动的复位？

背景知识

看门狗比喻的说明参见本章的引言。

解决方案

使用硬件看门狗意味着使用内部或者外部（硬件）定时器。

我们已经在前面的许多例子中看到，只要有可能，就应尽量使用片内模块，而不是相应的片外模块。具体地说，使用片内模块通常有以下好处：

- 降低硬件的复杂性，从而有助于提高系统的可靠性。
- 降低系统的成本。
- 降低系统的尺寸。

就看门狗定时器来说，情况更加复杂。因为外部看门狗芯片一般都能够提供大多数片内模块所不能提供的一些有用的功能。

例如：流行的“1232”看门狗（Dallas、Maxim、Linear Technology 和 AD 公司有各种版本可用）是一种低成本、低功耗的芯片。除了起到看门狗定时器的作用之外，它们还提供了对电源系统的监视功能（详细资料参见第 5 章的“可靠的复位”一节）。如果，正如在许多设计中遇到的那样，打算使用外部“可靠的复位”电路，那么 1232 芯片将允许扩展外部看门狗功能，只需添加很少的成本和增加非常少的硬件复杂性。

外部看门狗的另一个有益的特性是它们在本质上是可移植的。8051 系列的所有芯片通常都有能够使用相同的外部看门狗。相比而言，为内部看门狗编写的代码在用于不同硬件时通常必须重写。

使用（诸如 Infineon C515x 芯片中的）片内看门狗在某种情况下是有好处的。它可以确定系统是经历了一次普通复位还是由看门狗溢出所引起的复位。这样就可以根据情况来改变系统的行为。在使用外部看门狗时，通常必须通过复杂的编程才能得到这些信息。如果没有这些信息，系统将可能不断地由看门狗定时器溢出复位。

总之，如果需要看门狗功能，则必须仔细地比较内部和外部解决方案。没有惟一的“理想”

解决方案，必须综合考虑前面提到的问题，从而找到最适合需求的解决方案。

可靠性和安全性

在使用内部看门狗或者外部看门狗之前，必须确信使用这种定时器将提高（而不是减少）系统的可靠性。

首先要记住，看门狗动作应该用于灾难恢复。在设计得很好的系统中，看门狗复位的事件应该是一种很少发生且值得注意的事件。应该把看门狗的使用看作是“如果其他一切都已失效，那么必须让看门狗来复位系统”，这是一种从实际角度出发的对这种解决方法的功能的认识。

在使用看门狗时，如果在设计阶段考虑不周或者缺少适当的测试，可能会显著地降低系统的可靠性。在出现持续的硬件故障的情况下，错误的看门狗设计可能导致某些问题的出现。在这种情况下，实现错误的看门狗可能意味着系统将不断地复位，这将是极其危险的。

必须注意，因为它对错误作出反应所需的时间很长，所以看门狗对于许多系统都不适用。例如，假设在一个汽车制动系统中使用 500ms 的看门狗。当汽车以每小时 70 英里（每小时 110 千米）的速度行驶时遇到了问题。此种情况下，在开始复位制动系统之前，汽车和乘客已经向前方移动了大约 16 码（15 米）。简而言之，在需要快速恢复的场合，看门狗很少是最佳的解决方案。

可移植性

如上所述，内部看门狗的硬件不是 8051/8052 内核的一部分。因此，不同的 8051 芯片具有不同的看门狗，为片内看门狗编写的代码通常必须被改写以适用于不同的芯片。相比而言，为外部看门狗编写的软件具有更佳的可移植性。

优缺点小结

- ◎ 看门狗能够提供错误恢复的“最后手段”。应该把看门狗的使用看作是“如果其他一切都已失效，则必须让看门狗来复位系统”，这是一种从实际角度出发的对这种解决方法的功能上的认识。
- ◎ 在间歇故障的情形中（例如偶尔的电磁干扰脉冲）看门狗可能是非常有效的。
- ◎ 超时周期长的看门狗对于许多系统都不合适。
- ◎ 使用看门狗时，如果在设计阶段考虑不周或者缺少适当的测试，可能将显著地降低系统的可靠性。
- ◎ 在持续的硬件故障的情况下，错误的看门狗实现可能意味着系统将不断地复位。这将是非常危险的。

相关的模式和替代解决方案

在某些有限的环境中，也可以使用软件看门狗。

它可以由两个模块创建：

- 一个定时器中断服务程序

● 一个刷新函数

实际上，将定时器设置为在（比方说）60ms 后溢出。正常情况下，这个定时器将永远不会溢出，因为将定期调用刷新函数来重启定时器。但当程序被“阻塞”时，刷新函数就不会被调用。当定时器溢出时，中断服务程序将被调用。这样便实现一种“恰当的”错误恢复机制。

我们已经在几个系统中使用了软件看门狗。这种处理方法的主要问题是某些软件错误（例如那些由电磁干扰引起的）可能不仅破坏主要的程序代码，而且也将破坏看门狗定时器，虽然这在硬件看门狗上很少发生。相比之下，硬件看门狗更不易受到干扰。

软件看门狗的主要优点是可以具有多种不同的错误恢复方法（而不仅仅是整个芯片复位）。然而，使用片内硬件看门狗能够提供灵活的复位动作，在许多情况下，这是一种更可靠的解决方案。

例子：使用“1232”外部看门狗定时器

在这个例子中，假定将开发一个简单的集中供暖控制系统，通过使用一个外部“1232”看门狗芯片来提高系统的可靠性。

1232 的使用非常简单：

- 将看门狗连接到微控制器的复位管脚，如图 12.2 所示。

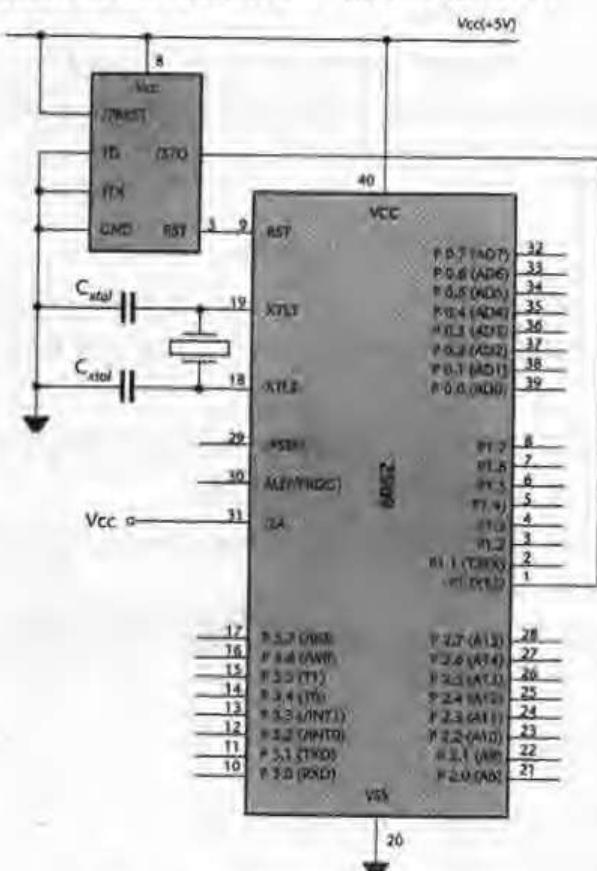


图 12.2 1232 看门狗的简单演示电路

- 选择三种（标称的）可选超时周期中的一个，连接 1232 上的 TD 管脚来选择相应的周期（参见表 12.1）。
- 定期向 1232 上的 ST 管脚发送脉冲，且脉冲周期小于超时周期。

表 12.1 1232 看门狗的定时

	最短超时	典型超时	最长超时
TD 接地	62.5ms	150ms	250ms
TD 浮空	250ms	600ms	1000ms
TD 接 Vcc	500ms	1200ms	2000ms

使用图 12.2 中的硬件的代码例子（150ms 超时），在源程序清单 12.1 和源程序清单 12.2 中给出。

```
/*
-----*
Main.C (v1.00)

-----*
使用“超级循环”的集中供暖系统的软件结构
***假定外部“1232”看门狗定时器在 P1^0 上...***
-----*/
#include "Cen_Heat.h"
#include "Dog_1232.h"
-----*/
void main(void)
{
    // 初始化系统
    C_HEAT_Init();
    // 看门狗自动启动
    while(1)
    {
        // 了解用户需要多高的温度
        // (通过用户界面)
        C_HEAT_Get_Required_Temperature();
        // 了解当前的房间温度是多少
        // (通过温度传感器).
        C_HEAT_Get_Actual_Temperature();
        // 按照需要调整燃气锅炉
        C_HEAT_Control_Boiler();
        // 喂看门狗
        WATCHDOG_Feed();
    }
}
-----文件结束-----
-----*/
```

源程序清单 12.1 使用超级循环和硬件看门狗所实现的集中供暖系统的部分演示

```

/*
-----*
Dog_1232.C (v1.00)
-----*
// 1232 外部看门狗定时器的程序库
-----*/
#include "Dog_1232.h"
#include "Main.h"
// ----- Port pins -----
// 连接 1232 (/ST管脚) 到 WATCHDOG_pin
sbit WATCHDOG_pin = P1^0;
/*-----*
WATCHDOG_Feed()
// “喂” 1232 类型的外部看门狗芯片
-----*/
void WATCHDOG_Feed(void)
{
    static bit WATCHDOG_state;
    // 改变看门狗管脚的状态
    if (WATCHDOG_state == 1)
    {
        WATCHDOG_state = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state = 1;
        WATCHDOG_pin = 1;
    }
}
/*-----*
----文件结束-----
-----*/

```

源程序清单 12.2 使用超级循环和硬件看门狗所实现的集中供暖系统的部分演示

例子：使用 Atmel 的 89S53 上的内部看门狗定时器

Atmel 的 89S53 是一种具有较好的片内看门狗定时器的标准 8051 微控制器的例子。

这个定时器的主要特点是由一个独立的振荡器控制。因此，系统能够对主晶体振荡器或谐振器的（间歇性）故障做出响应。

控制该看门狗定时器的主要寄存器是 WCON 寄存器，如表 12.2 所示。

表 12.2 用于控制 Atmel 的 AT89S53 上的片内看门狗定时器的 WCON 特殊功能寄存器

Bit	7	6	5	4	3	2	1	0
Name	PS2	PS1	PS0	NOT WD	NOT WD	NOT WD	WDTRST	WDTEN

WCON 中每个位的作用都在表 12.3 中进行了说明。

表 12.3 Atmel 的 AT89S53 的 WCON 特殊功能寄存器的更详细说明

位的符号	功 能
PS2 PS1 PS0	看门狗定时器的周期预置位。当所有三位置为“0”时，看门狗定时器标称周期为 16ms。当所有三位置为“1”时，标称周期为 2048ms。详细信息参见表 12.4。注意：实际的定时可能在标称值的±30%范围内变化
WDTRST	看门狗定时器复位。每当用户软件将这个位置为“1”时，将产生一个脉冲去复位看门狗定时器。WDTRST 位在下一个指令周期自动地复位为“0”。WDTRST 位是只写的
WDTEN	看门狗定时器使能位，WDTEN = 1 使能看门狗定时器，WDTEN = 0 禁止看门狗定时器

特殊功能寄存器 WCON 中的 PS0、PS1 和 PS2 周期预置位用来设置看门狗定时器的周期，从 16~2048ms。可选的定时周期如表 12.4 所示，实际的定时周期在标称值的±30%范围之内（Vcc = 5V 时）。

表 12.4 选择 AT89S53 中的看门狗定时器的周期

PS2	PS1	PS0	周期 (标称值)
0	0	0	16ms
0	0	1	32ms
0	1	0	64ms
0	1	1	128ms
1	0	0	256ms
1	0	1	512ms
1	1	0	1024ms
1	1	1	2048ms

看门狗定时器在上电复位时和省电模式下被禁止。通过设置特殊功能寄存器 WCON 中的 WDTEN 位而使能（地址为 96H），通过设置 WCON 中的 WDTRST 位复位看门狗定时器。当看门狗定时器没有被复位或禁止时，如果超时，则将产生一个内部的复位脉冲以复位 CPU。

源程序清单 12.3~源程序清单 12.7 说明了如何在前面所讨论的简单的集中供暖系统的例子中使用看门狗。

```
/*
Main.C (v1.00)

// 使用“超级循环”和“硬件看门狗”的集中供暖系统的演示
// [编译和运行]
*/
#include "Cen_Heat.h"
#include "Dog_AT.h"
```

```

/*-----*/
void main(void)
{
    // 初始化系统
    C_HEAT_Init();
    // 启动看门狗
    WATCHDOG_Init();
    while(1) // “死循环”（超级循环）
    {
        // 了解用户需要多高的温度
        // (通过用户界面)
        C_HEAT_Get_Required_Temperature();
        // 了解当前的房间温度是多少
        // (通过温度传感器)
        C_HEAT_Get_Actual_Temperature();
        // 按照需要调整燃气锅炉
        C_HEAT_Control_Boiler();
        // 喂看门狗
        WATCHDOG_Feed();
    }
}
/*-----文件结束-----*/
/*-----*/

```

源程序清单 12.3 使用超级循环和硬件看门狗所实现的集中供暖系统的部分演示

```

/*-----*/
Dog_AT.H (v1.00)

-详细资料参见 Dog_AT.C
/*-----*/
#include <At89S53.h>
// 函数原型
void WATCHDOG_Init(void); // 启动看门狗
// (为了提高速度) 使用宏来喂看门狗
#define WATCHDOG_Feed() WMCON |= 0x02
/*-----文件结束-----*/
/*-----*/

```

源程序清单 12.4 使用超级循环和硬件看门狗所实现的集中供暖系统的部分演示

```

/*-----*/
Dog_AT.C (v1.00)

Atmel 的 89S53 上的看门狗定时器功能的演示
[编译和运行]
/*-----*/
#include "Dog_AT.h"

```

```
-----*/
void WATCHDOG_Init(void)
{
    // 设置 512ms 的看门狗
    // PS2 = 1; PS1 = 0; PS0 = 1
    // Set WDTRST = 1
    // Set WDTEN = 1 -启动看门狗
    //
    // WMCN |= 10100011
    WMCN |= 0xA3;
}
-----文件结束-----
-----*/
```

源程序清单 12.5 使用超级循环和硬件看门狗所实现的集中供暖系统的部分演示

```
-----*
Cen_Heat.H (v1.00)
-----*
-详细资料参见 Cen_Heat.C
-----*/
// 函数原型
void C_HEAT_Init(void);
void C_HEAT_Get_Required_Temperature(void);
void C_HEAT_Get_Actual_Temperature(void);
void C_HEAT_Control_Boiler(void);
-----*
-----文件结束-----
-----*/
```

源程序清单 12.6 使用超级循环和硬件看门狗所实现的集中供暖系统的部分演示

```
-----*
Cen_Heat.C (v1.00)
-----*
使用“超级循环”的集中供暖系统的软件结构
[编译和运行]
-----*/
-----*/
void C_HEAT_Init(void)
{
    //
    // 这里是用户代码
    }
-----*/
void C_HEAT_Get_Required_Temperature(void)
{
    //
    // 这里是用户代码
    }
-----*/
```

```
void C_HEAT_Get_Actual_temperature(void)
{
    // 这里是用户代码
}

/*-----*/
void C_HEAT_Control_Boiler(void)
{
    // 这里是用户代码
}
/*-----*
 *-----文件结束-----*
 */
```

源程序清单 12.7 使用超级循环和硬件看门狗所实现的集中供暖系统的部分演示

进阶阅读

Part 3

单处理器系统的时间触发结构

本书主要涉及创建时间触发的嵌入式系统。在前面内容的基础之上，现在能够详细讨论为 8051 系列创建基于 C 的时间触发软件结构的方法。

在第 13 章中，将介绍调度器并说明如何用调度器来有效地建立时间触发系统。

在第 14 章中，将详细描述合作式调度器结构。这种简单而灵活的平台适用于非常广阔的嵌入式系统领域。

在系统中使用合作式调度器有许多好处，其中之一是简化了开发过程。然而，要从调度器得到最大的好处，必须以一种稍有不同（面向任务）的方法来进行设计。在第 15 章和第 16 章中将讨论如何具体实现。

最后，在第 17 章中，将给出混合式调度器。与本书中的所有调度器模式一样，混合式调度器也基于合作式调度器。然而，这种版本的混合式调度器也能支持单个抢占式任务。



调度器的介绍

本章将介绍对应于许多现代桌面系统中的“Windows”（或其他操作系统），调度器将在嵌入式系统中扮演类似的角色。

13.1 引言

在前面第 1 篇和第 2 篇的基础之上，我们现在能够详细讨论为 8051 系列微控制器创建时间触发系统的方法。

为了实现这种系统，将使用调度器。这是一种用于嵌入式系统的非常简单的运行环境。在这个介绍性的章节中，我们将说明什么是调度器以及合作式调度和抢占式调度之间的区别。同时将说明为什么使用合作式调度器将有助于使即便是最小的嵌入式系统也易于开发而且运行得更加可靠。

首先简要地回顾桌面系统使用操作系统的原因，并说明为什么这样的操作系统不适用于本书所讨论的嵌入式系统，并以此作为本章其余部分的讨论基础。

13.2 桌面操作系统

正如在序言里说明的，本书假定读者已经具有台式机应用软件开发的经验。在第 1 章中已讨论过，在许多信息系统中，桌面/工作站环境和诸如文字处理软件的通用桌面应用程序占主导地位。现代桌面环境的普遍特征是：用户通过高分辨率的图形屏幕加上键盘和鼠标与应用程序交互。操作系统以及相关的程序库为这种复杂的用户界面提供了支持。

在这样的一个平台上，通常用户所需要的程序（诸如文字处理软件）及全部数据（诸如文字处理软件处理的文件）都在需要时从磁盘实时读取。图 13.1 显示了这种文字处理软件的典型运行环境。在这里应用和底层硬件被隔离开来。例如，当用户想要将最新的小说保存在磁盘上时，文字处理软件将大多数必要的工作交给操作系统完成。随后，操作系统将许多硬件相关的命令交给 BIOS（基本输入/输出系统）。

操作系统（或 BIOS）对于台式 PC 机来说并不是必不可少的。然而，对于大多数用户来说，个人电脑的主要优点是它的灵活性。换句话说，同样的设备可以运行成千上万不同的程序。

如果 PC 没有操作系统，那么这些程序中的每一个都将需要能够独立实现所有的底层功能。这样，其效率将非常低，而且将使系统更加昂贵。同时也很可能将导致错误，因为即使是最小的程序也必须复制许多功能。

想像一下没有 Windows（或 UNIX）的世界会是个什么样子，可以加深对这个问题的认识。考虑一个曾经在 PC 上广泛使用的早期操作系统——DOS。用过 DOS 应用程序的读者可能会记得，每个程序都必须提供合适的打印机驱动程序。如果打印机后来有所变化，通常意味着必须升级 PC 上的每个应用程序来使用新的硬件。有了 Windows 后，就不会出现这样的问题。当购买了新打印机时，只需要一个驱动程序。安装了驱动程序之后，计算机上的每个程序都可以直接使用这个新的硬件。

对桌面操作系统的一种看法是：它用来运行多个程序，并且提供这些程序需要的“公共代码”（用于打印、文件存储、图形显示等等）。这减少了相同程序模块的复制，降低了出错的机会，从而使整个系统更加可靠并易于维护。

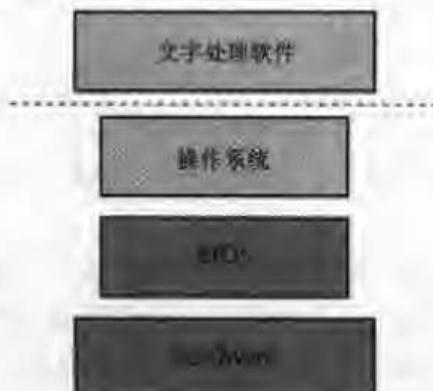


图 13.1 运行文字处理软件的桌面计算机系统的 BIOS/OS 多层结构的原理示意图

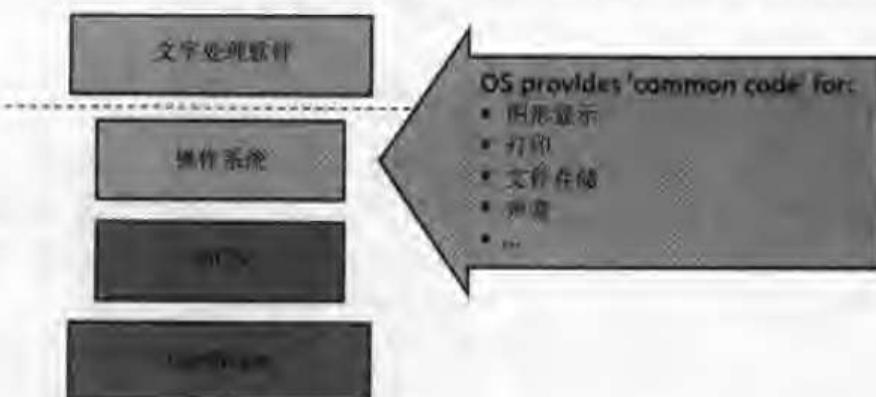


图 13.2 桌面系统中操作系统的作用的原理示意图

13.3 对超级循环结构的评价

现代桌面操作系统的许多特性（诸如图形显示、打印以及磁盘存取）对于嵌入式系统价值

不大。嵌入式系统中没有很复杂的图形屏幕、打印机和磁盘。

因此，正如在第 9 章看到的，许多简单的嵌入式系统所使用的软件结构都是一种超级循环（源程序清单 13.1）的形式。

```
/*-----*
Main.C
-----*
简单的超级循环应用的结构
-----*/
#include "x.h"
-----*/
void main(void)
{
    //
    // 准备任务 X
    X_Init();
    while(1) // “死循环”（超级循环）
    {
        X(); // 执行任务
    }
}
-----文件结束-----
-----*/
```

源程序清单 13.1 简单超级循环的部分演示

源程序清单 13.1 中展示的超级循环结构的主要优点是：

- (1) 简单，因此易于理解；
- (2) 几乎不占用系统存储器或 CPU 资源。

然而，“没有投入也就没有回报”。超级循环占用很少的存储器和处理器资源是因为它们几乎不为开发人员提供什么功能。尤其是这种结构，它很难在精确的时间间隔执行任务 X。正如我们将看到的，这种限制是一个非常大的缺点。

例如：考虑从一系列不同的嵌入式项目汇集的许多要求（没有特别的前后顺序）：

- 必须以 0.5s 间隔测量汽车的当前速度。
- 每秒必须刷新显示 40 次。
- 计算出来的新的油门位置必须每隔 0.5s 输出。
- 必须每秒执行 20 次时间-频率变换。
- 如果已经发出警报，则必须在 20 分钟之后关掉（法律上的要求）。
- 当前门被打开时，如果在 30s 内没有输入正确的口令，则必须发出警报。
- 必须每秒采样 1 000 次发动机振动数据。
- 必须每秒执行 20 次频域数据分类。
- 必须每 200ms 扫描一次键盘。
- 主机（控制）节点必须每秒与所有其他节点（传感器节点和发出警报节点）通信一次。

- 必须每 0.5s 计算一次新的油门位置。
- 传感器必须每秒采样一次。

总结这个列表可以发现，许多嵌入式系统必须在某些时刻执行任务。更具体地说，需要执行的任务分为两种：

- 周期性任务，（比方说）每 100ms 执行一次
- 单次任务，（比方说）在 50ms 的延迟之后执行一次

利用源程序清单 13.1 所示的基本结构很难实现上述任务。例如，假设必须每隔 200ms 启动任务 X，而完成该任务需要 10ms。源程序清单 13.2 说明了修改源程序清单 13.1 中的代码以实现这个要求的一种方法。

```
/*-----*/
void main(void)
{
    Init_System()
    while(1) // “死循环”（超级循环）
    {
        X(); // 执行任务（耗时 10ms）
        Delay_190ms(); // 延迟 190ms
    }
}
```

源程序清单 13.2 使用超级循环结构来每隔一段时间执行一次任务

源程序清单 13.2 中说明的方法通常是难以实际运用的，因为只有当满足以下条件时它才能工作：

1. 知道任务 X 的精确的运行时间
2. 这个运行时间永不变化

在实际系统中，很难确定任务的精确运行时间。假设有一个非常简单的任务，不与外界相互作用，而仅仅执行某些内部计算。即使在这种相当有限的情况下，改变编译程序优化设置，或者即使是改变一些表面上不相关的程序部分，也可能改变任务运行的速度。这使得调节定时的过程非常乏味并且易于出错。

第二种条件更是问题多多。在嵌入式系统中，任务往往需要以复杂的方式与外界相互作用。在这种情况下，任务的运行时间将随着外界行为的变化而变化，而程序员极难控制这种变化。

13.4 更好的解决方案

解决这个问题的更好方案是使用基于定时器的中断，在一定的时刻调用函数。

基于定时器的中断和中断服务程序

正如在第 1 章看到的，中断是一种用来当发生“事件”时通知处理器的硬件机制。这种事件可能是内部事件或者外部事件。8051/8052 内核结构共支持 7 个中断源：

- 三个定时/计数器中断 [分别与定时器 0、定时器 1 和定时器 2（如果有的话）相关]

- 两个有关 UART 的中断（注意：它们共用同一个中断向量，可以看作是一个中断源）
- 两个外部中断

此外，另有一个程序员极少控制的中断源：

- “上电复位”（POR）中断

当中断产生时，处理器“跳转”到程序存储器底部的某个地址。这些地址必须包含微控制器对中断做出响应的相应代码。通常，这里将包含另一个“跳转”指令，跳到位于程序存储器其他地方相应的“中断服务程序”地址。

虽然处理中断的这个过程看起来有点复杂，然而使用高级语言来创建中断服务程序（ISR）的过程是简单的，如源程序清单 13.3 所示。

```
/*-----*
Main.c
-----*
简单的定时器中断服务程序的演示程序
*-----*/
#include <AT89x52.h>
#define INTERRUPT_Timer_2_Overflow 5
// 函数原型
// 注意：中断服务程序不被直接调用，因此不需要原型
void Timer_2_Init(void);
/* ----- */
void main(void)
{
    Timer_2_Init(); // 设置定时器 2
    EA = 1; // 允许所有中断
    while(1); // 一个空的超级循环
}
/* ----- */
void Timer_2_Init(void)
{
    // 定时器 2 配置为 16 位定时器
    // 当溢出时自动重装
    //
    // 本代码（通用 8051/52）假定用在振荡周期为 12MHz 的系统
    // 因此定时器 2 的精度是 1.000 微秒
    // （详细资料参见第 11 章）
    //
    // 重装值为 FC18（十六进制）=64536（十进制）
    // 当定时器（16 位）的值达到 65536（十进制）时溢出
    // 这样，在这种设置下，定时器将每隔 1ms 溢出
    T2CON = 0x04; // 加载定时器 2 的控制寄存器
    T2MOD = 0x00; // 加载定时器 2 的模式寄存器
    TH2 = 0xFC; // 加载定时器 2 的高位字节
    RCAP2H = 0xFC; // 加载定时器 2 的重装捕捉寄存器的高位字节
    TL2 = 0x18; // 加载定时器 2 的低位字节
    RCAP2L = 0x18; // 加载定时器 2 的重装捕捉寄存器的低位字节
```

```

// 使能定时器 2 中断，并且将调用中断服务程序
// 每当定时器溢出时，如下
ET2 = 1;
// 起动定时器 2 运行
TR2 = 1;
}
/* ----- */
void X(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // 每隔 1ms 调用这个中断服务程序
    // 所需的代码放在这里...
}
/* ----- 文件结束 ----- */
/* ----- */

```

源程序清单 13.3 使用定时器中断服务程序定期调用任务的系统结构

在 Keil 硬件模拟器中运行源程序清单 13.3 所示程序的结果如图 13.3 所示。



图 13.3 在 Keil 硬件模拟器中运行源程序清单 13.3 所示的程序的结果

对我们来说，源程序清单 13.3 中的大部分代码应该是比较熟悉的。Timer_2_Init()函数中用于设置定时器 2 的代码和在第 11 章中讨论的延迟代码相同。在这个例子中有两个主要区别：

1. 当定时器溢出时将产生中断
 2. 定时器将自动重装，然后立即再次开始计数
- 在下面的小节中将进一步讨论这两个区别。

中断服务程序 (ISR)

定时器 2 溢出产生的中断激活中断服务程序，这里将调用 X()。

```

/* ----- */
void X(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // 每隔 1ms 调用这个中断服务程序一次
    // 所需的代码放在这里...
}
/* ----- */

```

这个函数和定时器溢出之间的联系是使用 Keil 关键字 interrupt(包含在函数定义的函数头之后):

```
void X(void) interrupt INTERRUPT_Timer_2_Overflow
```

加上以下的#define 伪指令来实现的:

```
#define INTERRUPT_Timer_2_Overflow 5
```

为了理解“5”的来源,需要注意,ISR 使用的中断号直接与 8051 的 IE 特殊功能寄存器中的中断源使能位的位置相对应。即 IE 寄存器的位 0 与使用“中断 0”的函数有关。表 13.1 显示了中断源和早期 8051/8052 定义的中断号之间的联系。

总的说来,使用定时器溢出中断是一种安全而强大的技术,其应用将贯穿于本书。

自动定时器重装

正如前面指出的,当定时器 2 溢出时,将自动重装并立即再次开始计数。在这种情况下,定时器将使用“捕捉”寄存器(注意,这些寄存器的名称随芯片厂家的不同而稍有变化)中的数值重装:

```
RCAP2H = 0xFC; // 加载定时器 2 的重装捕捉寄存器的高位字节  
RCAP2L = 0x18; // 加载定时器 2 的重装捕捉寄存器的低位字节
```

表 13.1 8051 中断源

中断源	地址	IE 位
上电复位	0x00	-
外部中断 0	0x03	0
定时器 0 溢出	0x0B	1
外部中断 1	0x13	2
定时器 1 溢出	0x1B	3
UART 收发	0x23	4
定时器 2 溢出	0x2B	5

注意,许多 8051 有更多的中断源,相应中断号的详细资料参见厂家的文档。

自动重装功能保证了定时器始终以精确的 1ms 间隔产生所需的时标,而不需要任何软件开销,以及任何用户程序的干预。

定时器 2 所具有的自动重装能力简化了将这个定时器用作定期时标源的应用。注意,定时器 0 和定时器 1 也具有自动重装功能,但只有作为 8 位定时器运行时才可用。在大多数系统中,8 位定时器只能用来产生间隔大约 0.25ms(或更短)的中断,通常这是不够用的。

13.5 例子：闪烁LED

上面给出的例子是抽象的，这里给出定时器驱动的中断服务程序的另一个例子。在这个例子中，使用定时器以有规律的时间间隔闪烁 LED（源程序清单 13.4）。

注意，在这个系统中使用定时器 1 溢出来激活中断服务程序。正如在第 11 章中讨论的，定时器 1 没有 16 位“自动重装”模式。因此，在每次定时器溢出时都必须手工重装。函数 Timer_1_Manual_Reload() 将执行这个操作。

```
/* -----
Main.c
-----
简单的定时器中断服务程序的演示程序
闪烁 LED
----- */
// 标准“8052”芯片
#include <AT89x52.h>
#define INTERRUPT_Timer_1_Overflow 3
bit LED_state_G;
// 通过这个管脚闪烁 LED
sbit LED_pin = p1^7;
// 函数原型
// 注意：中断服务程序不能被直接调用，因此不需要原型
void Timer_1_Init(void);
void Timer_1_Manual_Reload(void);
void LED_Flash_Init(void);
/* ----- */
void main(void)
{
    Timer_1_Init(); // 设置定时器 1
    LED_Flash_Init(); // 准备闪烁 LED
    EA = 1; // 允许所有中断
    while(1)
    {
        PCON |= 0x01; // 进入睡眠（空闲模式）
    }
}
/* ----- */
void Timer_1_Init(void)
{
    // 定时器 1 配置为 16 位定时器
    // 当溢出时手工重装
    TMOD &= 0x0F; // 清除所有有关 T1 的位 (T0 不变)
    TMOD |= 0x10; // 设置所需的 T1 的位 (T0 不变)
    // 设置定时器重装值
    Timer_1_Manual_Reload();
    // 使能定时器 1 中断
```

```
ET1 = 1;
}

/* -----
void Timer_1_Manual_Reload(void)
{
// 停止定时器 1
TR1 = 0;
// 本代码（通用 8051/8052）假定用在振荡周期为 500kHz 的系统中,
// 那么定时器 1 的精度就是 0.000024s
//（详细资料参见第 11 章）
//
// 想要每秒产生一个中断:
// 需要 1.0/0.000024 定时器增量
// 即（大约）41667 个定时器增量
//
// 重装值（十进制）是 65536 - 41667 -> 23869 = 0x5D3D
TL1 = 0x3D;
TH1 = 0x5D;
// 启动定时器 1
TR1 = 1;
}

/* -----
void LED_Flash_Init(void)
{
// 准备闪烁 LED
LED_state_G = 0;
}

/* -----
void LED_Flash_Update(void) interrupt INTERRUPT_Timer_1_Overflow
{
// 在指定端口管脚上闪烁 LED（或产生脉冲给蜂鸣器）
// 频率由定时器设置决定
// 本版本必须手工重装定时器
//（定时器 0 或定时器 1 无法执行 16 位自动重装）
Timer_1_Manual_Reload();
// 使 LED 从灭变亮（反之亦然）
//（每秒重复）
if (LED_state_G == 1)
{
LED_state_G = 0;
LED_pin = 0;
}
else
{
LED_state_G = 1;
LED_pin = 1;
}
```

```

    }
/*-----文件结束-----*/

```

源程序清单 13.4 使用定时器驱动的中断服务程序来按照有规律的时间间隔闪烁 LED

图 13.4 显示了源程序清单 13.4 的程序在 Keil 硬件模拟器下的运行结果。

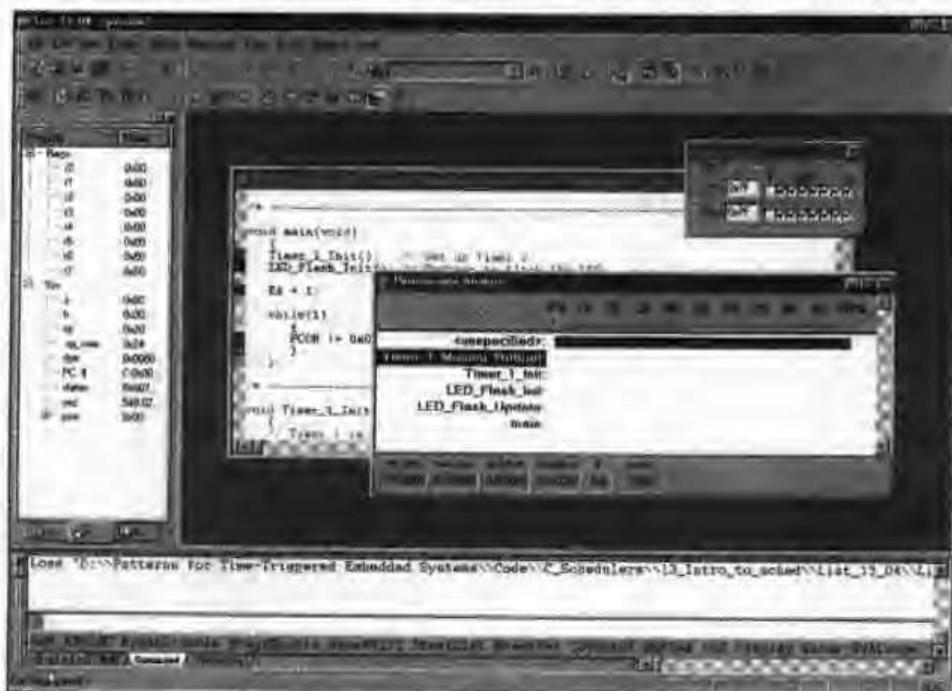


图 13.4 源程序清单 13.4 的程序在 Keil 硬件模拟器下的运行结果

13.6 在不同的时间间隔执行多个任务

虽然绝大多数嵌入式系统只要求运行一个程序，但是确实有必要支持多个任务（在本书中通过“C”函数的形式实现）的运行。如上所述，这些任务必须以周期性或单次的方式运行，一般具有不同的运行时间并以不同的时间间隔运行。例如，可能需要每隔 1ms 从模数转换器读取输入，每隔 200ms 读取一个或多个开关状态，以及每隔 3ms 刷新一次 LCD 显示。

通过扩展第 13.5 节中讨论的技术，就可以运行多个任务。例如，假设微控制器有（比如说）三个定时器可用。通过使用独立的中断服务程序来执行每个任务，可以使用这些定时器来控制三个任务的运行（源程序清单 13.5）。

```

/*-----*
 Main.c
-----*/

```

多个定时器中断服务程序结构

```

-----*/

```

```
#include <AT89x52.h>
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5
// 函数原型
// 注意，中断服务程序不被直接调用，因此不需要原型
void Timer_0_Init(void);
void Timer_1_Init(void);
void Timer_2_Init(void);
/* -----
void main(void)
{
    Timer_0_Init(); // 设置定时器 0
    Timer_1_Init(); // 设置定时器 1
    Timer_2_Init(); // 设置定时器 2
    EA = 1; // 允许所有中断
    while(1);
}
/* -----
void Timer_0_Init(void)
{
    // 详略
}
/* -----
void Timer_1_Init(void)
{
    // 详略
}
/* -----
void Timer_2_Init(void)
{
    // 详略
}
/* -----
void X(void) interrupt INTERRUPT_Timer_0_Overflow
{
    // 每隔 1ms 调用这个中断服务程序一次
    // 详细代码在此省略
}
/* -----
void Y(void) interrupt INTERRUPT_Timer_1_Overflow
{
    // 每隔 2ms 调用这个中断服务程序一次
    // 详细代码在此省略
}
/* -----
void Z(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // 每隔 5ms 调用这个中断服务程序一次
}
```

```

    // 所需的代码放在这里...
}

/*
-----文件结束-----
*/

```

源程序清单 13.5 使用三个独立的定时器来执行三个任务的系统结构

通常只要有足够的定时器可用，这种方法就将有效。然而，这种方法违反了基本的软件设计准则。

例如在源程序清单 13.5 中，有三个不同的定时器需要管理（而如果有 100 个任务，将需要 100 个定时器）。这就使系统的维护变得非常困难，例如，如果改变振荡器频率，将需要在 100 处做相应改动。而且也很难扩展，例如，如果没有更多的硬件定时器可用，如何再添加一个任务？

除了违反最基本的软件设计准则外，源程序清单 13.5 的程序有一个更具体的问题。这个问题出现在当多个中断同时产生的情况下。正如在第 1 章看到的，系统有多于一个有效的中断将可能导致不可预知的运行结果，由此将造成性能上的不可靠。

再回头看源程序清单 13.5，同时产生多个中断的情况是不可避免的。处理这种情况是有可能的，但是将极大地增加系统的复杂性，

总的说来，正如将在下一节看到的，使用调度器将提供一个非常完美的解决方案。

13.7 什么是调度器

可以从两种角度来看调度器：

- 一方面，调度器可以看作是一个简单的操作系统，允许以周期性或（更少见）单次方式来调用任务。
- 从底层的角度来看，调度器可以看作是一个由许多不同任务共享的定时器中断服务程序。因此，只需要初始化一个定时器，而且改变定时的时候通常只需要改变一个函数。此外，无论需要运行 1 个、10 个还是 100 个不同的任务，通常都可以使用同一个调度器完成。注意，这种“共用中断服务程序”与桌面操作系统提供的共用打印功能非常类似。

例如：源程序清单 13.6 展示了如何使用调度器来调度源程序清单 13.5 中的三个任务。

```

/*
void main(void)
{
    // 设置调度器一次
    SCH_Init();
    // 增加任务 (1ms 时标间隔)
    // Function_A 将每隔 2ms 运行一次
    SCH_Add_Task(Function_A, 0, 2); SC
    // Function_B 将每隔 10ms 运行一次
    SCH_Add_Task(Function_B, 1, 10);
}

```

```

// Function_C 将每隔 15ms 运行一次
SCH_Add_Task(Function_C, 3, 15);
SCH_Start();
while(10
{
    SCH_Dispatch_Tasks();
}
}

/*
---文件结束---
*/

```

源程序清单 13.6 使用一个调度器来（以不同的时间间隔）运行三个周期性任务

13.8 合作式调度和抢占式调度

前面非常简单地讨论了使用调度器在特定的时刻运行函数。在下一章详细讨论调度器的创建和使用之前，必须了解调度器分为两大类：

- 合作式调度器
- 抢占式调度器

图 13.5 和图 13.6 中比较了这两种调度器的特性。

合作式调度器

- 合作式调度器提供了一种单任务的系统结构

操作：

- 任务在特定的时刻被调度运行（以周期性或单次方式）
- 当任务需要运行时，被添加到等待队列
- 当 CPU 空闲时，运行等待任务中的下一个（如果有的话）
- 任务运行直到完成，然后由调度器来控制

实现：

- 这种调度器很简单，用少量代码即可实现
- 该调度器必须一次只为一个任务分配存储器
- 该调度器通常完全由高级语言（比如“C”）实现
- 该调度器不是一种独立的系统，它是开发人员的代码的一部分

性能：

- 设计阶段需要小心以快速响应外部事件

可靠性和安全性：

- 合作式调度简单、可预测、可靠并且安全

图 13.5 合作式调度器的特性

本书中主要使用合作式调度器，并且有限制地使用混合式的调度器（图 13.7）。这两种调度器一起提供了所需要的功能（在多个任务之间共享一个定时器，既能够运行“周期性的”任务，也能够运行“单次”任务），同时避免了（完全的）抢占式平台的固有复杂性。

合作式调度器不但可靠而且可预测的主要原因是在任一时刻只有一个任务是活动的。这个任务运行直到完成，然后由调度器来控制。与此相对，在完全的抢占式系统的情况下，有多个活动任务。在这样的系统中，假设有一个任务正在从端口读取时，调度器执行了“上下文切换”，使另一个任务访问同一个端口。在这种情况下，如果不采取措施阻止这种操作，数据将可能丢失或被破坏。

抢占式调度器

- 抢占式调度器提供了一种多任务的系统结构

操作:

- 任务在特定的时刻被调度运行（以周期性或单次方式）
- 当任务需要运行时，被添加到等待队列
- 等待的任务（如果有的话）运行一段固定的时间，如果没有完成，将被暂停并放回到等待队列。然后下一个等待任务将运行一段固定的时间，诸如此类等等

实现:

- 这种调度器相对复杂，因为必须实现诸如信号灯这样的特性，用来在“并行处理的”任务试图访问共用的资源时避免冲突
- 该调度器必须为抢占任务的所有中间状态分配存储器
- 该调度器通常将（至少是部分的）由汇编语言编写
- 该调度器通常作为一个独立的系统被创建

性能:

- 对外部事件的响应速度快

可靠性和安全性:

- 与合作式调度相比，通常认为其更不可预测，并且可靠性较低

图 13.6 抢占式调度器的特性

混合式调度器

- 混合式调度器提供了有限的多任务功能

操作:

- 支持多个合作式调度的任务
- 支持一个抢占式任务（可以中断合作式任务）

实现:

- 这种调度器很简单，用少量代码即可实现
- 该调度器必须一次为两个任务分配存储器
- 该调度器通常完全由高级语言（比如“C”）实现
- 该调度器不是一种独立的系统，它是开发人员的代码的一部分

性能:

- 对外部事件的响应速度快

可靠性和安全性:

- 只要小心设计，可以和单纯的合作式调度器一样可靠

图 13.7 混合式调度器的特性

这个问题往往出现在多任务处理平台中所谓的代码“关键段”。这样的关键段是那些一旦开始就必须不中断地运行直至完成的代码区。关键段的例子包括：

- 修改或读取变量（尤其是用于任务间通信的全局变量）的代码。一般说来，这是最常见的关键段，因为任务间的通信往往是一种关键要求。
- 与诸如端口、模数转换器（ADC）等等硬件接口的代码。例如，如果同一个模数转换器同时由多个任务使用，将发生什么？
- 调用公共函数的代码。例如，如果同一个函数同时由多个任务调用，将会发生什么？在合作式系统中，这些问题不会出现，因为任何时刻都只有一个任务是活动的。在抢占式系统中处理这样的关键代码段主要有两种可行的方法：
- 在开始关键段之前通过禁止调度器来“暂停”调度；当离开关键段时重新允许调度器中断。
- 使用“锁”（或其他形式的“信号灯机制”）来实现类似的结果。

第一个解决方案是当开始访问共享资源（比方说端口 X）时，禁止调度器。这直接解决了该问题，因为（比方说）任务 A 在完成对端口 X 的操作之前不会中断运行。然而，这种“解决方案”是不完美的。首先，如果禁止调度器，将不再能够跟踪消逝的时间，所有的定时函数都将开始漂移在这个例子中，每当访问端口 X 时，定时将漂移相当于任务 A 的运行时间，这是难以接受的。

初看起来，使用加锁的方法易于实现，是一种较好的解决方案。在进入关键代码段之前，将相关的资源“锁住”；而当完成对资源的访问后，再“解锁”。当被锁定时，其他程序都无法进入该关键段。^①

以下是一种实现方式：

1. 任务 A 检查想要访问的端口 X 的“锁”。
2. 如果被锁定，任务 A 将等待。
3. 当端口被解锁时，任务 A 先设置锁，然后使用该端口。
4. 当任务 A 完成端口访问时，离开该关键段，并解锁该端口。

这种算法的代码实现似乎也很简单，如源程序清单 13.7 所示。

```
#define UNLOCKED      0
#define LOCKED       1

bit Lock:      // 全局锁标志

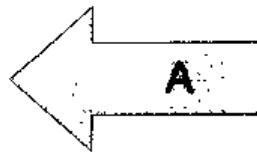
// ...
// 准备进入关键段
// 等待锁被解开
// (为简单起见，没有显示超时功能)
```

^①当然，这只是用来部分地解决由多任务导致的问题。如果任务 A 的目的是从 ADC 读数，当任务 A 运行时任务 B 已经锁定了该 ADC，那么任务 A 将无法实现所需的功能。使用锁或任何其他机制无法解决这个问题。然而，它们可以防止系统崩溃。当然，如果使用合作式调度器，这个问题将不会出现。

```

while(Lock == LOCKED);
// 锁被解开
// 进入关键段
// 设置锁
Lock = LOCKED;
// 这里是关键代码
// 准备离开关键段
// 释放锁
Lock = UNLOCKED;
// ...

```



源程序清单 13.7 在抢占式调度器中实现一种简单的加锁机制

然而，上述代码并不是在所有情况下都能正确运行。

考虑源程序清单 13.7 中标记为“**A**”的部分代码，如果系统是完全的抢占式，那么我们的任务可能在调度器执行上下文切换并且允许任务 B 访问 CPU 的同时运行到这个地方。如果任务 Y 也要访问端口 X，那么将可能出现以下情况：

- 任务 A 检查端口 X 的锁并发现端口没有锁定；然而，任务 A 还没有改变锁标志。
- 然后任务 B “被切入”。任务 B 检查锁标志，它仍然是开放的。任务 B 设置该锁标志并开始使用端口 X。
- 然后任务 A 再次 “被切入”。就任务 A 而言，该端口没有锁定。因此这个任务将设置标志并开始使用端口，而没发现任务 B 已经这样做了。
-

正如所看到的，这些简单的锁定代码违反了互斥的原则。即允许多个任务访问关键代码段。出现这个问题的原因是：在任务检查锁标志之后但是改变锁标志之前，有可能发生上下文切换。换句话说，“检查并设置锁的代码”（用来控制对关键代码段的访问）本身就是一个关键段。

这个问题可以得到解决。例如，因为“检查并设置”锁定码需要花费的时间很短，所以可以在段时间内禁止中断。然而，这本身并不是一种完整的解决方案，因为即使在“检查并设置”的短时间内也可能会产生中断，所以必须检查相关的中断标志，如果必要的话，还要调用有关的中断服务程序。这可以做到，但是将增加运行环境的复杂性。

13.9 抢占式调度器详解

与本章前面几节相比，本节中的讨论更具专业性，在第一次阅读本书时可以跳过。

各种研究表明，与抢占式调度器相比，合作式调度器有许多合乎需要的特性。例如，Nissanke (1997, 第 237 页) 写道：

[抢占式]调度需要更多的运行开销，因为在上下文切换时对部分计算结果进行存储与检索。[合作式]算法不需要这样的开销。[合作式]算法的其他优点包括：更易理解、更具可预测性、易于测试及其固有的保证对任何共享资源或数据独占访问的性能。

与之类似, Allworth (1981, 第 53~54 页) 写道:

使用这种[合作式]技术将得到显著的优点。因为程序是不可中断的, 所以同步上的问题不会造成共享数据方面的问题。可以实现子程序的共享而不必编写可重入代码或实现加锁以及解锁机制。

同样, 在最近的一次讲座中, Bates (2000) 指出了与抢占式的替代方案相比, 合作式调度有下四个优点:

1. 这种调度器更简单
2. 减少了开销
3. 容易测试
4. 权威认证机构更容易接受这种形式的调度

虽然有这些观点, 但上面这些作者和这一领域的绝大多数开发人员都集中于使用抢占式调度器。抢占式解决方案被广泛讨论的部分原因是在可用选择上的混淆。例如, Bennett (1994, 第 205 页) 写道:

如果考虑调度 CPU 的时间分配, 那么有两个基本的可选方案: [1]循环调度, [2]抢占式调度。

实际上, 与 Bennett 的论述相反, 他所说的循环调度本质上是超级循环的一种形式。正如在第 9 章看到的, 这类结构仅适用于有限范围内的一些非常简单的系统, 尤其是在那些对精确定时没有特别要求并且可用的存储器和 CPU 资源比较有限的场合。超级循环并不足以代表适用于广阔领域的合作式调度结构。

然而, Bennett 并不是个别的例子, 其他研究人员也有类似的假定 (参见 Barnett, 1995)。例如, Locke (1992, 第 37 页) 在一本广泛被引用的出版物中建议:

传统上, 有两种基本方法来实现具有硬实时要求的应用系统的总体设计: 循环执行……和固定优先级[抢占式]结构。

类似的, Cooling (1991, 第 292-3 页) 比较了合作式调度和抢占式调度解决方案。然而, 他对合作式调度器的讨论同样也限制在只考虑循环调度的特例。因此, 得出抢占式解决方案更有效的结论不足为奇。

当合理地比较抢占式调度和合作式调度的不同性能时, 往往关注的是长的任务将影响合作式调度器的响应。这种关注由 Allworth (1981) 简要概括:

合作式方法的主要缺点是当前的程序正在运行的时候, 系统对外界的变化不敏感。因此, 如果要求系统的实时响应能力不被削弱, 系统的进程就必须极其简短。

这种关注通常是没有必要的, 因为有四个主要的技术原因:

- 在许多嵌入式系统中, 任务的运行时间极其简短。例如, 考虑书中讨论的一个更复杂的算法——比例积分微分 (PID) 控制。大多数基本的 8051 微控制器都可以在 0.4ms 内执行一次 PID 计算, 在广泛应用 PID 算法的飞行控制系统中, 10ms 左右的采样速

度十分常见，0.4ms 的计算时间并没有显著占用处理器的开销。

- 在那些有长任务的系统中，往往是因为开发人员不知道运用一些简单的技术来将这些长任务以合适的方式分解，从而将偶尔调用的长任务转换成为经常调用的短任务。贯穿于本书将运用这种技术，将在第 16 章对其进行介绍和分析。
- 在很多情况下，微控制器性能的提高远远超过了许多嵌入式系统对性能上的要求。例如，前面给出的在早期的具有 1 MIPS 性能指标的 8051 微控制器上运行 PID 的性能数据。正如在第 3 章看到的，这个系列许多低成本的芯片现在具有 5~50MIPS 的性能指标。解决性能问题的简单而高性价比的方式往往并不是使用更复杂的软件结构，而是更新硬件。
- 如果设计任务或微控制器的升级不能提供足够的性能上的改善，那么可以使用多个微控制器，现在这是很普通的。例如，一个典型的汽车平台包含超过 40 个嵌入式处理器 (Leen 等, 1999)。有了更多可用的处理器，如果必要的话，长任务可以很容易地“移植”到另一个处理器上，使主 CPU 能够对其他事件迅速响应（这种方法的许多例子参见本书第 6 篇）。

最后值得注意的是，抢占式调度器被更广泛地讨论和使用的原因可能并不完全是技术上的问题。实际上，对于一些公司来说，使用抢占式平台有明显的商业上的好处。例如在第 9 章展示的，全部使用高级程序设计语言，只需大约 300 行 C 代码就可以很容易地构造一个合作式调度器。这些代码的可移植性非常好，易于理解和使用，并且实际上是免费使用的。相比而言，抢占式运行环境增加了复杂性，从而导致更长的代码结构（即使在简单的实现中也需要增大十倍，Labrosse, 1998）。在大多数情况下，这些代码的长度和复杂性使其不适用于普通开发人员自己构建，因此提供了商业“实时操作系统”产品的销售基础，抢占式调度器通常价格昂贵，而且往往还需要占用很大的实时开销。这些平台持续不断的宣传和销售反过来又促进了对这个领域的更进一步的学术研究。例如，根据 Liu 和 Ha (1995) 所述：

再设计的目标是采用现成的商业化标准操作系统。因为这些操作系统不支持循环调度，所以我们必须放弃这种传统的调度方法。

13.10 小结

在本章中，我们说明了什么是调度器以及合作式调度和抢占式调度之间的区别。并说明了合作式调度器提供了一种简单而可靠的运行环境，这正好满足了大多数嵌入式系统的需要。

近年来，我们已经在许多实际系统中使用了本书介绍的合作式调度器版本，并帮助许多学生在他们的第一个嵌入式系统中使用这种结构。我们确信，正确使用这种调度器不仅将使设计简单、清晰和可靠，而且将使桌面开发人员在面对嵌入式系统开发的挑战时能更从容地面对。

13.11 进阶阅读

Allworth, S.T. (1981)《实时软件设计入门》，Macmillan, 伦敦。

Bates, I. (2000)《调度及定时分析入门》, 在实时系统中使用 Ada 语言 (4月6日, 2000), IEE 会议出版物 00/034。

Burns, A. 和 Wellings, A. (1997)《实时系统和编程语言》, Addison-Wesley, 伦敦。

Cooling, J.E. (1991)《实时系统软件设计》, Chapman and Hall, 伦敦。

Kopetz, H. (1997)《实时系统: 分布式嵌入式系统的设计准则》, Kluwer Academic, 纽约。

Leen, G., Heffernan, D. 和 Dunne, A. (1999)《汽车中的数字网络》。Computing and Control, 10 (6): 257~266。

Leveson, N.G. (1995)《安全件: 系统安全和计算机》, Addison-Wesley, Reading, MA.

Liu, J.W.S. 和 Ha, R. (1995)《验证实时限制的方法》系统和软件期刊, 30 (1-2), 85~98。

Locke, C.D. (1992)《硬实时应用的软件结构: 周期性执行和固定优先级执行的比较》。实时系统期刊, 4: 37~53。

Nissanke, N. (1997)《实时系统》, Prentice Hall, 伦敦。

Shaw, A.C. (2001)《实时系统和软件》, Wiley, 纽约。

Storey, N. (1996)《安全至上的计算机系统》, Addison-Wesley, 伦敦。

Ward, N.J. (1991)《安全至上的航空控制系统的静态分析》, 摘自 D.E.Corbyn 和 N.P., Bray (eds)《航空运输安全: 安全和可靠性学会春季会议论文集》。1991, SaRS。

Chapter 14

合作式调度器

引言

在本章中，我们将讨论建立适用于单处理器平台的合作式调度器的技术。这种技术为各种嵌入式系统提供了一种灵活而又可预测的软件平台，这些嵌入式系统从简单的消费类小玩意到飞机控制系统，无所不包。

本章将介绍以下模式：

- 合作式调度器

合作式调度器提供了一种简单而可预测性非常高的平台。该调度器全部用“C”编写而且成为系统的一部分。这将使整个系统的运行更加清晰且易于开发、维护，以及向不同的平台上移植。存储器的开销为每个任务 7 个字节，对 CPU 的要求（随时标间隔而变）很低。

合作式调度器

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一个时间触发结构。

问题

怎样建立并使用合作式调度器？

背景知识

本节中给出一些相关背景材料的链接。

什么是合作式调度器？

合作式调度器的入门介绍参见第 13 章。

函数指针

许多C程序员不熟悉函数指针。相对来说，函数指针很少用于桌面程序，然而它却是创建调度器的关键。因此将在这里提供一个介绍性的简短例子。

需注意的要点是：例如，正如能够确定一组数据在存储器中的起始地址，也可以在存储器中找到特定函数的可执行程序代码的起始地址。这个地址用于“指向”该函数，最重要的是，它可用于调用该函数。

只要小心使用，函数指针能够使复杂的程序更易于设计和实现。例如，假设正在开发一个大规模的、安全至上的系统来控制一个工厂。一旦检测到紧急的情况，将希望尽可能快速地关闭系统。然而，关闭系统的合理方式随系统的状态而变化。因此，将建立多种恢复函数和一个函数指针。每当系统状态改变时，就改变函数指针使它总是指向最合理的恢复函数。这样就可以保证一旦出现紧急情况，便能够通过函数指针快速地调用最合理的函数。

源程序清单14.1和源程序清单14.2中的例子说明了函数指针的一些基本特征。

```
-----  
Main.C (v1.00)  
-----  
  
函数指针的演示  
-----  
*-----  
#include "Main.h"  
#include "Printf51.h"  
#include <stdio.h>  
// -----私有函数原型-----  
void Square_Number(int, int*);  
/* ..... */  
/* ..... */  
int main(void)  
{  
    int a = 2, b = 3;  
    void (* pFn)(int, int*); /* 声明 pFn 作为 fn 的指针，fn 带有 int 参数和 int  
                           指针参数（返回 void） */  
    //  
    int Result_a, Result_b;  
  
    // 准备[在 Keil 硬件模拟器上]使用 printf()  
    Printf51_Init();  
    pFn = Square_Number; // pFn 存放 Square_Number 的地址  
    printf("Function code starts at address: %u\n", (tWord) pFn);  
    printf("Data item a starts at address: %u\n\n", (tWord) &a);  
  
    // 以传统的方式调用"Square_Number"  
    Square_Number(a, &Result_a);  
  
    // 使用函数指针调用"Square_Number"  
    (*pFn)(b, &Result_b);  
}
```

```

printf("%d squared is %d (using normal fn call)\n", a, Result_a);
printf("%d squared is %d (using fn pointer)\n", b, Result_b);
while(1);
return 0;
}
/*-----*/
void Square_Number(int a, int* b)
{
// 演示——计算 a 的平方
*b = a * a;
}
/*-----文件结束-----*/
/*-----*/

```

源程序清单 14.1 介绍函数指针的使用的部分例子

```

/*-----*/
Printf51.C (v1.00)

简单的串行初始化程序，用来允许使用 Keil 硬件模拟器来运行桌面 C 程序例子
[串行接口程序库的详细资料将在第 18 章给出，这里的代码仅用作演示！！！]
/*-----*/
#include "Main.h"
#include "Printf51.h"
/*-----*/
Printf51_Init()
简单的串行初始化程序，用来允许使用 Keil 硬件模拟器来运行桌面 C 程序例子
/*-----*/
void Printf51_Init(void)
{
const tWord BAUD_RATE = 9600;
PCON &= 0x7F; // 将 SMOD 位置为 0 (波特率不加倍)
// 接收
// 8 位数据、1 个起始位、1 个停止位
// 可变波特率 (异步)
// 只有当收到一个有效的停止位时接收标志才置为 1
// T1 位置为 1 (发送缓冲器空)
SCON = 0x72;
TMOD |= 0x20; // T1 为模式 2、8 位自动重装
// OSC_FREQ 和 OSC_PER_INST 的详细资料参见 Main.H
TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
    / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));
TL1 = TH1;
TR1 = 1; // 启动定时器
TI = 1; // 发送填充字节
// 中断“不”使能
ES = 0;
}

```

```
/*-----文件结束-----*/
*/
```

源程序清单 14.2 介绍函数指针的使用的部分例子

注意，源程序清单中的程序设置了 80x52 微控制器上的 UART，以便使用 Keil 的 printf() 库函数。这个例子仅用作演示，将在第 18 章给出完整的串行程序库的例子。

源程序清单 14.1 和源程序清单 14.2 中的程序在 Keil 硬件模拟器上产生如图 14.1 所示的输出。

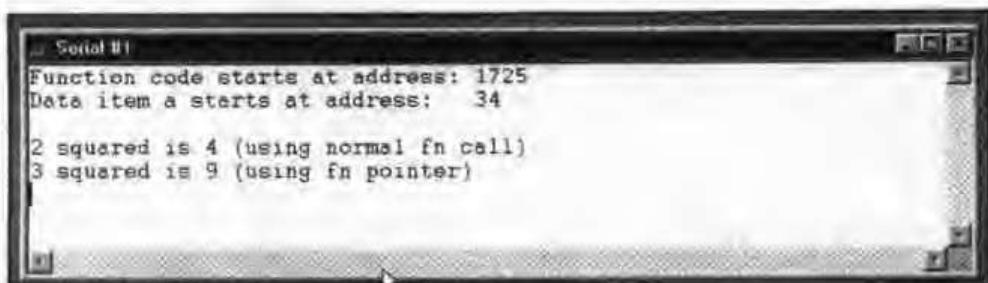


图 14.1 使用函数指针

解决方案

调度器有以下主要组成部分：

- 调度器数据结构。
- 初始化函数。
- 中断服务程序（ISR），用来以一定的间隔刷新调度器。
- 向调度器增加任务的函数。
- 使任务在应当运行的时候被执行的调度函数。
- 从调度器删除任务的函数（并不是所有系统都需要）。

在本节中将讨论这些所需的模块。

概述

在讨论调度器的模块之前，先讨论一下在用户看来什么是调度器。用一个简单的例子来说明：一个用来重复闪烁 LED 的调度器，一秒亮，一秒灭，如此循环。

源程序清单 14.3 显示了如何实现它：

```
/*-----*/
void main(void)
{
    // 设置调度器
    SCH_Init_T2();
    // 为"Flash_LED"任务作准备
    LED_Flash_Init();
```

```

// 增加"Flash LED"任务 (1000ms 亮, 1000ms 灭)
// 定时单位为时标 (1ms 时标间隔)
// (最大的间隔/延迟是 65535 个时标)
SCH_Add_Task(LED_Flash_Update, 0, 1000);
// 开始调度器
SCH_Start(); //刷新任务队列
while(1)
{
    SCH_Dispatch_Tasks();
}
/*
----- */
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // 刷新任务队列
    ...
}

```

源程序清单 14.3 一个简单调度应用的主要组成部分

源程序清单 14.3 如下运行：

1. 假定 LED 将通过 LED_Flash_Update()任务被点亮和熄灭。这样，如果 LED 最初是熄灭的，则调用 LED_Flash_Update()两次，LED 将被点亮然后再次熄灭。

因此，为了获得需要的闪烁频率，要求调度器每秒调用 LED_Flash_Update()一次，且无限循环。

2. 使用函数 SCH_Init_T2()来准备调度器。

3. 调度器准备好后，使用函数 SCH_Add_Task()将函数 LED_Flash_Update()添加到调度任务队列中。同时，以如下方式指定 LED 以需要的频率闪烁：

```

// 增加 "Flash LED" 任务 (1000ms 亮, 1000ms 灭)
// 定时单位为时标 (1ms 时标间隔)
// (最大的间隔/延迟是 65535 个时标)
SCH_Add_Task(LED_Flash_Update, 0, 1000);

```

(随后将讨论 SCH_Add_Task()的所有参数，并研究它的内部结构)。

4. 函数 LED_Flash_Update()的定时将由函数 SCH_Update()控制，SCH_Update()是一个由定时器 2 溢出触发的中断服务程序：

```

void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // 刷新任务队列
    ...
}

```

5.“刷新”中断服务程序不运行任务，而是计算任务应该在什么时候运行并设置标志。运行 LED_Flash_Update()的任务由调度函数 (SCH_Dispatch_Tasks()) 完成，这个函数在主 (超级) 循环中运行。

```

while(1)
{
    SCH_Dispatch_Tasks();
}

```

在详细讨论这些模块之前，应该承认对于闪烁 LED 来说，这是一种复杂的实现方式。如果目标是开发一个需要最少存储器以及最短代码长度的闪烁 LED 应用，那么这并不是一种好的解决方案。然而，我们的主要目的是在后面所有的例子中都将使用同样的调度器结构。这些系统将包括许多有实际价值的、复杂的系统。为理解这种平台的运行方式所付出的努力将很快得到回报。

还需要强调的是调度器是一种“低成本的”方案，它占用很小比例的 CPU 资源（将随后介绍精确的百分比）。此外，就调度器本身而言，每个任务只要求不超过 7 个字节的存储器。因为在一个典型的系统中不会超过 4~6 个任务，即使运行在 8 位微控制器上，所需的任务预算（大约 40 个字节）也是不多的。

调度器数据结构以及任务队列

调度器的核心是调度器数据结构。这是一种用户自定义的数据类型，集中了每个任务所需的信息。

这里（从文件 Sch51.H）复制了数据结构：

```

// 可能的话，存储在 DATA 区，以供快速存取
// 每个任务的存储器总和是 7 个字节
typedef data struct
{
    // 指向任务的指针（必须是一个"void(void)"函数）
    void (code * pTask)(void);
    // 延迟（时标）直到函数将（下一次）运行
    // 详细说明参见 SCH_Add_Task()
    tWord Delay;
    // 连续运行之间的间隔（时标）
    // ——详细说明参见 SCH_Add_Task()
    tWord Period;
    // 当任务需要运行时（由调度器）加 1
    tByte RunMe;
} sTask;

```

文件 Sch51.H 也包括常数 SCH_MAX_TASKS：

```

// 在程序运行期间的任一时刻允许的任务最大数目
//
// 每个新建项目都必须调整
#define SCH_TASKS (1)

```

在文件 Sch51.C 中，数据类型 sTask 和常数 SCH_MAX_TASKS 一起用来创建任务队列，并一直被调度器所引用：

```
// 任务队列
```

```
sTask SCH_tasks_G[SCH_MAX_TASKS];
```

任务队列的大小

必须通过调整 `SCH_MAX_TASKS` 的值来保证足够长的任务队列，以保存系统所需的任务。例如，如果需要调度如下的三个任务：

```
SCH_Add_Task(Function_A, 0, 2);
SCH_Add_Task(Function_B, 1, 10);
SCH_Add_Task(Function_C, 3, 15);
```

那么 `SCH_MAX_TASKS` 必须为 3（或更大），以保证调度器的正常运行。

同时注意，如果不满足这个条件，调度器将产生一个错误代码（参见图 14.3）。

初始化函数

如同大多数需要被调度的任务一样，调度器本身也需要一个初始化函数。虽然该函数执行各种重要的操作，诸如准备调度器队列（在前面讨论的）以及准备错误代码变量（将在后面讨论），然而这个函数的主要用途是设置定时器，用来产生驱动调度器的定期“时标”。

正如在第 11 章看到的，大多数 8051 芯片都有三个定时器（定时器 0、定时器 1，以及定时器 2），它们中的任何一个都能用来驱动调度器。然而，只有定时器 2 可以用作自动重装的 16 位精度定时器。因此，如果可能的话，使用该定时器是合理的。

注意：本书配套 CD 上包含了许多不同的 8051 调度器。这些不同的调度器说明了所有三种定时器的使用。

使用定时器 2 的一个初始化函数的例子在源程序清单 14.4 中给出：

```
/*
 *-----k-----
 * SCH_Init_T2()
 * 调度器初始化函数。准备调度器数据结构并且设置定时器以所需的频率中断。
 * 必须在使用调度器之前调用这个函数
 *-----k-----*/
void SCH_Init_T2(void)
{
    tByte i;
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // - SCH_Delete_Task() 将产生一个错误代码
    // (因为任务队列是空的)
    Error_code_G = 0;
    // 现在设置定时器 2
    // 自动重装、16 位定时器功能
    // 晶振假定为 12MHz
```

```

// 定时器2的精度是0.000001s(1微秒)
// 要求的定时器2溢出为0.001s(1ms)
// -需要1000个定时器时标
// 重装值为65536 - 1000 = 64536(十进制) = 0xFC18
T2CON = 0x04; // 加载定时器2的控制寄存器
T2MOD = 0x00; // 加载定时器2的模式寄存器
TH2 = 0xFC; // 加载定时器2的高位字节
RCAP2H = 0xFC; // 加载定时器2的重装捕捉寄存器的高位字节
TL2 = 0x18; // 加载定时器2的低位字节
RCAP2L = 0x18; // 加载定时器2的重装捕捉寄存器的低位字节
ET2 = 1; // 使能定时器2中断
TR2 = 1; // 启动定时器2
}

```

源程序清单 14.4 调度器初始化函数的例子

当使用本书中的任何一个调度器时，通常必须修改初始化代码来满足需要。尤其必须保证：

1. 初始化函数中假定的振荡器/谐振器频率与硬件相符。在源程序清单 14.4 中，假定频率为 12MHz。
 2. 调度器的时标间隔满足需要。在源程序清单 14.4 中，时标间隔为 1ms。
- 下面的“可靠性和安全性”中将提供有关选择时标间隔的指导。

“每个微控制器一个中断”的原则

调度器的初始化函数将使能和微控制器某个定时器溢出有关的中断。

因为在第 1 章中讨论的理由，本书始终假定只有“时标”中断源是活动的。具体地说，假定没有别的中断被使能。

如果在允许有其他的中断时试图使用调度器代码，那么系统根本不能保证运行正常。通常，充其量也只不过是得到完全不可预知的而且很不可靠的系统行为。

“刷新”函数

“刷新”函数是调度器的中断服务程序。它由定时器的溢出激活（正如在前面讨论的，使用“初始化”函数来设置），和大多数调度器类似，刷新函数并不复杂（参见源程序清单 14.5）。

当刷新函数确定某个任务需要运行时，将这个任务的 RunMe 标志加 1，然后该任务将由调度程序执行，正如在后面讨论的。

```

/*
 *-----*
 * SCH_Update()
 * 这是调度器的中断服务程序。初始化函数中的定时器设置决定了它的调用频率
 * 这个版本由定时器2中断触发，定时器自动重装
 *-----*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    TF2 = 0; // 必须手工清除
}

```

```

注意：计算单位为“时标”（不是毫秒）
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
    // 检测这里是否有任务
    if (SCH_tasks_G[Index].pTask)
    {
        if (SCH_tasks_G[Index].Delay == 0)
        {
            // 任务需要运行
            SCH_tasks_G[Index].RunMe += 1; // “RunMe” 标志加 1
            if (SCH_tasks_G[Index].Period)
            {
                // 调度周期性的任务再次运行
                SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
            }
        }
        else
        {
            // 还没有准备好运行，延迟减 1
            SCH_tasks_G[Index].Delay -= 1;
        }
    }
}
}

```

源程序清单 14.5 一个调度器“刷新”函数

“添加任务”函数

正如其名称所暗示的，“添加任务”函数用来添加任务到任务队列上，以保证它们在需要的时候被调用。

“添加任务”函数的参数在图 14.2 中进行了说明。

这里是一些例子。

这组参数使函数 Do_X() 在 1000 个调度器时标后运行一次：

```
SCH_Add_Task(Do_X, 1000, 0);
```

这组参数的作用相同，但是将任务标识符（在任务队列中的位置）保存以便以后在必要时删除该任务（关于从任务队列删除任务的更详尽的资料参见 SCH_Delete_Task()）：

```
Task_ID = SCH_Add_Task(Do_X, 1000, 0);
```

这组参数使函数 Do_X() 每隔 1000 个调度器时标周期性地运行一次。一旦调度开始，该任务就开始运行：

```
SCH_Add_Task(Do_X, 0, 1000);
```

这组参数使函数 Do_X() 每隔 1000 个调度器时标周期性地运行一次。任务将首先在 T = 300 个时标时执行，然后在 1300 个时标、2300 个时标等等执行：

```
SCH_Add_Task(Do_X, 300, 1000);
```

该函数的内容在源程序清单 14.6 中进行了介绍。

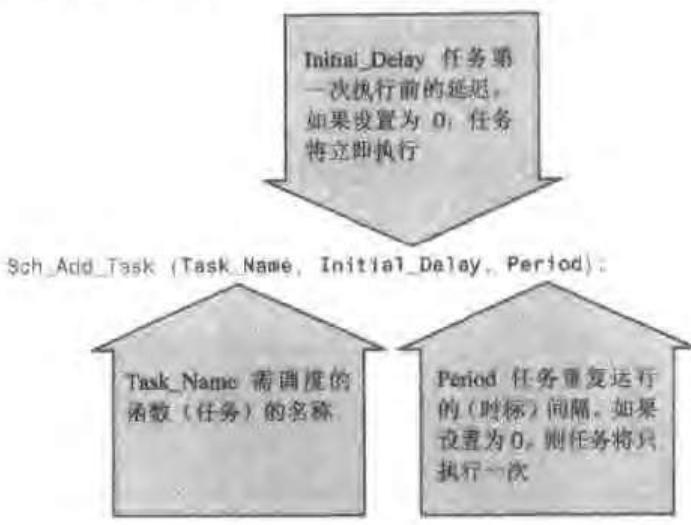


图 14.2 SCH_Add_Task()函数的参数

```
/*
SCH_Add_Task()
使任务(函数)每隔一定间隔或在用户定义的延迟之后执行
*/
tByte SCH_Add_Task(void * pFunction(),
                   const tWord Delay,
                   const tWord PERIOD)
{
    tByte Index = 0;
    // 首先在队列中找到一个空隙(如果有的话)
    while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
    {
        Index++;
    }
    // 是否已经到达队列的结尾?
    if (Index == SCH_MAX_TASKS)
    {
        // 任务队列已满
        //
        // 设置全局错误变量
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
        // 同时返回错误代码
        return SCH_MAX_TASKS;
    }
    // 如果能运行到这里, 则说明任务队列中有空间
    SCH_tasks_G[Index].pTask = pFunction;
    SCH_tasks_G[Index].Delay = DDELAY;
    SCH_tasks_G[Index].Period = PERIOD;
}
```

```
SCH_tasks_G[Index].RunMe = 0;
return Index; // 返回任务的位置（以便以后删除）
}
```

源程序清单 14.6 调度器“添加任务”函数的实现

“调度”程序函数

正如已经看到的，“刷新”函数不执行任何函数任务，需要运行的任务由“调度程序”函数激活。

源程序清单 14.7 中给出了调度程序的具体内容。

```
/*
SCH_Dispatch_Tasks()
这是“调度程序”。当任务（函数）需要运行时，SCH_Dispatch_Tasks()将运行它。
这个函数必须被主循环（重复）调用
*/
void SCH_Dispatch_Tasks(void)
{
    tByte Index;
    // 调度（运行）下一个任务（如果有任务就绪）
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        if (SCH_tasks_G[Index].RunMe > 0)
        {
            (*SCH_tasks_G[Index].pTask()); // 执行任务
            SCH_tasks_G[Index].RunMe -= 1; // 复位/减小 RunMe 标志
            // 周期性的任务将自动地再次执行
            // -如果这是个“单次”任务，将它从队列中删除
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }
    // 报告系统状况
    SCH_Report_Status();
    // 这里调度器进入空闲模式
    SCH_Go_To_Sleep();
}
```

源程序清单 14.7 调度器“调度任务”函数的实现

调度程序是超级循环中的惟一模块：

```
/*
void main(void)
{
    ...
}
```

```

while(1)
{
    SCH_Dispatch_Tasks();
}

```

是否需要一个调度函数？

乍看起来，即使用“刷新”函数又使用“调度”函数似乎是一种相当复杂的任务执行方式。具体地说，看起来调度函数也许是不必要的，而刷新函数能够直接激活任务。然而，为了在长任务的情况下使调度器的可靠性最大化，分离刷新和调度操作是必要的。

假设有一个 1ms 时标间隔的调度器，无论出于什么原因，有时待调度的任务具有 3ms 的运行时间。

如果刷新函数直接运行函数，长任务将一直运行，时标中断将被禁止。具体地说，将漏掉两个“时标”。这意味着所有的系统定时都将受到严重影响，并且可能有两个（或更多个）任务不能被调度执行。

如果将刷新和调度函数分开，则当长任务运行的时候系统时标仍然能够被处理。这意味着虽然发生任务“抖动”（漏掉的任务不能在正确的时间运行），但这些任务最终将运行。

函数指针和 Keil 链接选项

当如下编写程序时：

```
SCH_Add_Task(Do_X, 1000, 0);
```

“添加任务”函数的第一个参数是指向函数 Do_X() 的指针，该函数指针被传递给调度函数，通过这个函数执行任务：

```

if (SCH_tasks_G[Index].RunMe > 0)
{
    (*SCH_tasks_G[Index].pTask)(); // 执行任务
}

```

我们在“背景知识”一节中曾经讨论过函数指针的使用，在小型的微控制器上使用 C 函数指针是一种挑战，尤其是当函数指针被用作函数参数时。

在桌面系统上，函数参数通常使用压栈和出栈汇编指令由堆栈来传递。因为 8051 的堆栈大小有限（最多只有 128 字节，在一些芯片上只有 64 字节），所以函数参数必须使用不同的技术来传递：就 Keil C51 而言，这些参数保存在固定的存储单元内。当调用链接程序时，它将建立一个程序调用树，判断哪些函数参数是互斥的（即哪些函数不能同时被调用），并且将复用这些参数。

当函数指针被用作函数参数时，链接程序将难以确定正确的调用树，“添加任务”函数就是这样的例子。有两种现实的选择来处理这种情况：

1. 通过设置项目的链接选项为禁止复用，来阻止编译程序使用 OVERLAY 伪指令。

注意，与使用复用的应用程序相比，通常需要更多的 RAM 来运行程序。

2. 通过在链接程序的“其他选项”对话框中的明确说明，来通知链接程序如何为系统创建正确的调用树。

这种解决方案通常使用更少的存储器，但是编译程序往往无法反馈是否在提供的信息中存在错误。如果在这个方案中出错，程序将产生不可预测的结果。

链接选项并不难以理解，假设已经运行了在本章开始处所给出的简单的闪烁 LED 例子，并且正在调度一个任务，如下所示：

```
void main(void)
{
//...
// 添加“Flash LED”任务（1000ms 亮，1000ms 灭）
// 定时单位为时标（1ms 时标间隔）
// （最大的间隔/延迟是 65535 个时标）
SCH_Add_Task(LED_Flash_Update, 0, 1000);
//...
```

因为指针 LED_Flash_Update 在 main() 中出现，所以在链接程序看来该函数似乎是从 main() 中调用的，然而实际上该函数是从 SCH_Dispatch_Tasks 调用的。

显式地使用链接选项来做出这种改变，这样：

```
OVERLAY
(main ~ (LED_Flash_Update),
SCH_Dispatch_Tasks ! (LED_Flash_Update))
```

阅读 CD 上的各个项目文件，可以看到已经实现了所需的链接选项。

有关函数指针的更加详尽的资料请参考 Keil 编译程序文档以及 Keil Application Note 129（包含在 CD 上）。

函数指针是否安全？

使用基于函数指针的调度器是否有危险？使用函数指针是否将使程序更不可靠？这两个问题的答案都是“不”，然而有一些警告。

在断定使用函数指针是不安全的之前，需要讨论替代方案，可以创建不使用函数指针的调度器（我们已经创建了几个这样的调度器）。然而，与基于指针的版本相比较，最后所得到的代码比较长、更不灵活、更不易修改和阅读。以我们的经验，这些因素加在一起对可靠性造成的不良影响，将远远超过通过避免使用函数指针而得到的好处。

如果选择不使用 Keil OVERLAY 伪指令，就不需要考虑函数指针的复杂性。如果不希望使用这种指针或者代码以后可能被经验较少的开发人员维护，那么对于当前版本的 Keil 编译程序，这是最合理的方案。

“开始”函数

“开始”函数非常简单（参见源程序清单 14.8）。在所有任务都被添加之后，将调用这个函数来开始调度过程。该函数通过允许全局中断来实现其功能。

```
/*-----*/
void SCH_Start(void)
{
```

```
EA = 1;
}
```

源程序清单 14.8 调度器“开始”函数的实现

“删除任务”函数

当任务被添加到任务队列时，SCH_Add_Task()返回该任务在任务队列中的位置：

```
Task_ID = SCH_Add_Task(Do_X, 1000, 0);
```

有时需要从队列中删除任务，可以如下使用 SCH_Delete_Task()来实现：

```
SCH_Delete_Task(TAsk_ID);
```

SCH_Delete_Task()的详细资料在源程序清单 14.9 中给出。

```
/*-----*/
bit SCH_Delete_Task(const tByte TASK_INDEX)
{
    bit Return_code;
    if (SCH_tasks_G[TASK_INDEX].pTask == 0)
    {
        // 这里没有任务...
        //
        // 设置全局错误变量
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;
        // 同时返回错误代码
        Return_code = RETURN_ERROR;
    }
    else
    {
        Return_code = RETURN_NORMAL;
    }
    SCH_tasks_G[TASK_INDEX].pTask = 0x0000;
    SCH_tasks_G[TASK_INDEX].Delay = 0;
    SCH_tasks_G[TASK_INDEX].Period = 0;
    SCH_tasks_G[TASK_INDEX].RunMe = 0;
    return Return_code; // 返回状态
}
```

源程序清单 14.9 调度器“删除任务”函数的实现

降低功耗

调度应用程序的一个重要特性是能够支持低功耗运行。这是有可能的，因为当前所有的 8051 系列芯片都提供“空闲”模式来暂停 CPU 的活动，同时保持处理器的状态。在这种模式下，运行处理器所要求的功率一般减少大约 50%。

在调度应用程序中，这种空闲模式尤其有用。因为可以由软件控制进入空闲模式（如源程序清单 14.10 所示），而当微控制器收到任何中断时返回正常运行方式。因为调度器会产生定

期的时钟中断，所以可以在每次调用调度程序的结尾将系统置为“睡眠”，并将在下一个定时器时标产生时醒来。

```
/*-----*/
void SCH_Go_To_Sleep()
{
    PCON |= 0x01;      // 进入空闲模式(通用8051版本)
    // 在80c515/80c505上为了避免意外的触发，进入空闲模式需要两个连续的指令
    // PCON |= 0x01;    // 进入空闲模式(#1)
    // PCON |= 0x20;    // 进入空闲模式(#2)
```

源程序清单 14.10 调度器“睡眠”函数的实现

例程14.10

报告错误

硬件会产生故障，而软件永远是不完美的，因此错误是难免的。

为了在调度应用程序的任何部分都能报告错误，使用一个(8位)错误代码变量 Error_code_G。该变量在 Sch51.C 中定义如下：

```
// 用来显示错误代码
tByte Error_code_G = 0;
```

为了记录错误，需要包含以下代码：

```
Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
Error_code_G = ERROR_SCH_LOST_SLAVE;
Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
```

这些错误代码在文件 Main.H 中给出，该文件是项目头文件模式的一个例子。

调度器有一个函数 SCH_Report_Status()，由刷新函数调用，用来报告这些错误代码。

一种实现的例子在源程序清单 14.11 中给出。

```
/*-----*/
void SCH_Report_Status(void)
{
#ifdef SCH_REPORT_ERRORS
// 只在需要报告错误时适用
// 检查新的错误代码
if (Error_code_G != Last_error_code_G)
{
    // 假定 LED 采用负逻辑
    Error_port = 255 - Error_code_G;
    Last_error_code_G = Error_code_G;
    if (Error_code_G != 0)
    {

```

```

        Error_tick_count_G = 60000;
    }
else
{
    Error_tick_count_G = 0;
}
}
else
{
    if (Error_tick_count_G != 0)
    {
        if (--Error_count_G == 0)
        {
            Error_code_G = 0; // 复位错误代码
        }
    }
}
#endif
}

```

源程序清单 14.11 调度器“报告状态”函数的实现

注意：通过修改 Port.H 头文件可以禁止错误报告：

```
// 如果不需要错误报告，就将这一行注释掉
// #define SCH_REPORT_ERRORS
```

当需要错误报告时，在哪个端口显示错误代码同样可以通过 Port.H 设置来决定：

```
#ifdef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1

#endif
```

注意：在这个实现中，错误代码每隔 60 000 个时标（当时标频率为 1ms 时，即为 1 分钟）报告。

显示错误代码的最简单的方式是在端口上连接 8 个 LED(以及相应的缓冲器)，正如在“IC 驱动器”模式中讨论的。图 14.3 说明了一种解决方法。

错误代码的含义

这里讨论的错误报告在本质上是底层的形式，主要用来帮助应用程序的开发人员或合格的维护工程师完成系统维护。系统中可能还需要另一个用户界面，以一种对用户更加友好的方式来通知用户发现错误。

增加一个看门狗

这里给出的基本调度程序不提供对看门狗定时器的支持。

这样的支持将很有用，并且很容易添加，如下所示：

- 在调度器开始函数中启动看门狗。
- 在调度器刷新函数中刷新看门狗。

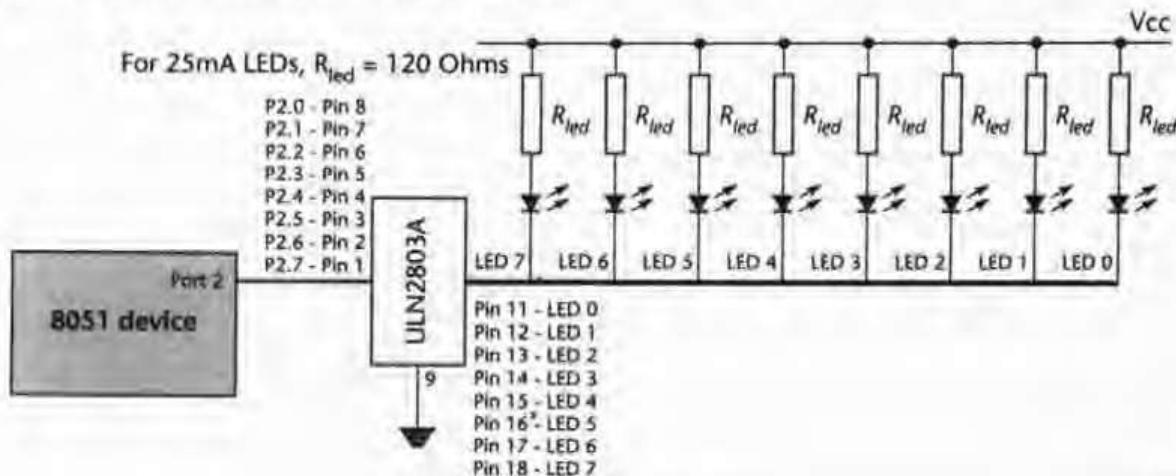


图 14.3 报告错误代码的硬件

硬件资源

主要从三个方面来讨论有关硬件资源的问题：定时器、存储器和 CPU 开销。

定时器

本模式需要一个硬件定时器。如果可能的话，应该使用具有自动重装功能的 16 位定时器，比如定时器 2（详细资料参见第 13 章）。

存储器

主调度程序需要的存储器为每个任务 7 字节。大多数应用程序要求有大约六个或更少的任务。即使在标准的有 256 字节内部存储器的 8051/8052 中，所有存储器的开销也是较小的。

CPU 开销

一些开发人员仍然不使用调度器的主要原因可能是担心这种结构带来的 CPU 开销，这种关注通常是没必要的。因为在典型的系统中，调度器平均消耗大约 5% 的可用 CPU 时间。

为了说明这个论点，首先考虑最坏情况下的例子：一个运行在每个振荡器周期 12 个机器周期下的标准 8051，用来调度一个简单的任务（闪烁 LED）。进一步假定：要求该调度器提供 1ms 时标间隔，而振荡器频率是 12MHz。

可以使用（包含在 CD 上的）Keil 硬件模拟器来模拟这个系统。结果显示 CPU 开销的大部分被调度器的“刷新”中断服务程序占用。所有的调度器处理过程合计占用大约 14% 的 CPU 可用时间（图 14.4）。

如果再添加一个任务，调度器仍然是 80% “空闲”的（图 14.5）。



图 14.4 使用 Keil 硬件模拟器来估计运行在一个 12MHz (每个指令 12 振荡周期) 的 8051 上的 1ms 时标调度器占用的 CPU 开销, 86%空闲

注意, 正在运行一个任务。试验显示 CPU 是 86%空闲的, 因此可允许的任务最大运行时间为大约 0.86ms.



图 14.5 使用 Keil 硬件模拟器来估计运行在一个 12MHz (每个指令 12 振荡周期) 的 8051 上的 1ms 时标调度器占用的 CPU 开销, 80%空闲

注意, 正在运行两个任务。试验显示 CPU 是 80%空闲的, 因此可允许的任务最大运行时间为大约 0.80ms.

在大多数情况下, 在这样一个系统中能够调度最多大约 12 个任务。图 14.6 显示了这种情况。



图 14.6 使用 Keil 硬件模拟器来估计运行在一个 12MHz (每个指令 12 振荡周期) 的 8051 上的 1ms 时标调度器占用的 CPU 开销, 11%空闲

注意, 正在运行 12 个任务。试验显示 CPU 是 11%空闲的, 因此可允许的任务最大运行时间为大约 0.11ms.

当然, 以现在的标准, 这种 12MHz/12 个振荡周期的 8051 是一种慢速芯片。图 14.7 显示了相同的系统 (运行一个任务), 改为运行在 24MHz 的芯片上的调度器。这个系统是 93%空闲的。

类似的, 相同系统改为运行在 Dallas 的 320/520 (32MHz) 的芯片上时, 为 98%空闲, 如图 14.8 (左边) 所示。在这种情况下, 同样的系统能够轻松地运行 12 个任务[图 14.8 (右边)] 并且仍然有 85%的空闲。

最后, 图 14.9 显示了在 12MHz/12 个时钟的 8051 上运行 10ms 时标的调度器。在左边, 系统运行 1 个任务并且是 98%空闲的。在右边, 系统运行 12 个任务并且是 91%空闲的。



图 14.7 使用 Keil 硬件模拟器来估计运行在一个 24MHz (每个指令 12 个振荡周期) 的 8051 上的 1ms 时标调度器占用的 CPU 开销

注意，正在运行一个任务。试验显示 CPU 是 93% 空闲的，因此可允许的任务最大运行时间为大约 0.93ms。



图 14.8 使用 Keil 硬件模拟器来估计运行在一个 32MHz (每个指令 4 个振荡周期) 的 8051 上的 1ms 时标调度器占用的 CPU 开销

注意，(左边) 正在运行一个任务。试验显示 CPU 是 97% 空闲的，因此可允许的任务的最大运行时间为大约 0.97ms。(右边) 正在运行 12 个任务。试验显示 CPU 是 85% 空闲的，因此可允许的任务的最大运行时间为大约 0.85ms。



图 14.9 使用 Keil 硬件模拟器来估计运行在一个 12MHz (每个指令 12 个振荡周期) 的 8051 上的 10ms 时标调度器占用的 CPU 开销

注意，(左边) 正在运行一个任务。试验显示 CPU 是 98% 空闲的，因此可允许的任务的最大运行时间为大约 9.8ms。(右边) 正在运行 12 个任务。试验显示 CPU 是 91% 空闲的，因此可允许的任务的最大运行时间为大约 9.1ms。

可靠性和安全性

在本节中，讨论一些关键的可靠性和安全性问题。

确信任务队列足够大

详细资料参见“解决方案”一节。

小心使用函数指针

详细资料参见“背景知识”和“解决方案”两节。

处理任务重叠

假设系统中有两个任务（任务 A 和任务 B）。更进一步假定任务 A 将每隔 1s 运行而任务 B 将每隔 3s 运行，同时假定每个任务的运行时间为大约 0.5ms。

假设以如下方式（1ms 时标间隔）调度这些任务：

```
SCH_Add_Task(TaskA, 0, 1000);  
SCH_Add_Task(TaskB, 0, 3000);
```

在这个例子中，这两个任务有时会同时运行。当两个任务都将运行时，任务 B 总是在任务 A 之后运行（详细资料参见源程序清单 14.5 和源程序清单 14.6）。这意味着如果任务 A 的运行时间有变化，那么任务 B 将出现“抖动”。当两个任务重叠时，则不能在正确时间调用。

或者，假设以如下方式调度这些任务：

```
SCH_Add_Task(TaskA, 0, 1000);  
SCH_Add_Task(TaskB, 5, 3000);
```

现在，两个任务仍然按要求（分别）每隔 1000ms 和 3000ms 运行。然而，任务 A 的调度总是在任务 B 之前 5ms 开始。因此，任务 B 将总能准时运行。

在很多情况下，通过明智地使用任务初始延迟，能够避免所有（或大多数）的任务重叠。

确定所需的时标间隔

贯穿于本书，主要讨论按照 ms 尺度运行的应用程序。这样，添加到调度器中的各种任务一般将间隔（比方说）12ms、3ms 和 1000ms。

大多数情况下，满足各种任务间隔需要的最简单的方式是分配 1ms 的调度器时标间隔，这很容易做到。详细资料参见第 11 章的“硬件延迟模式”和第 13 章的内容。

记住，调度器本身将占用微控制器的 CPU 负荷。在较短的时标间隔下，这种开销将显著增加（参见“硬件资源”一节）。为了使调度器开销尽可能的低（以及减少功耗，参见以下内容），应该使用较长的时标间隔。

如果想把开销和功耗降低到最小值，调度器的时标间隔应该设置为所有任务的运行间隔（以及间隔偏移量）的“最大公因数”。只需要使用一些简单的高中数学知识就可以很容易地计算出来。

假设有三个任务（X、Y 和 Z），任务 X 每隔 10ms 运行，任务 Y 每隔 30ms 运行，而任务 Z 每隔 25ms 运行。设置调度器的时标间隔需要计算有关的因子，如下所示：

- 任务 X 的间隔（10ms）因子^①为 1ms、2ms、5ms 和 10ms。
- 类似地，任务 Y 的间隔（30ms）因子如下：1ms、2ms、3ms、5ms、6ms、10ms、15ms

^① 因子是在 1 到 X 之间的整数，通过因子可以整除 X。

和 30ms。

- 最后，任务 Z 的间隔（25ms）因子如下：1ms、5ms 和 25ms。

因此，在这个例子中，最大公因数是 5ms，这就是所需的时标间隔。

注意：如果任务间隔为（比方说）5ms、25ms 和 1000ms，计算最大公因数的过程看起来就将极其乏味，因为 1000 有许多因子。然而，实际上只需考虑小于或等于最小的任务间隔的因子。因此，在这个例子中，只对 5、25 和 1000 在 1~5 之间的因子感兴趣。这里的最大公因数为 5ms。

如果考虑任务初始延迟，情况将稍微复杂些。

回顾前面的例子，假设决定使用 5ms 的调度器，添加三个任务到调度器，如下所示：

```
SCH_Add_Task(X, 0, 2);
SCH_Add_Task(Y, 0, 6);
SCH_Add_Task(Z, 0, 5);
```

显而易见，这些任务将经常重叠。例如，每当任务 Y 被调度运行时，任务 X 也将被调度运行。在某些时刻，三个任务将需要同时运行。为了避免这种情况的发生，可以添加一些任务初始延迟，如下所示：

- 任务 X 将每隔 10ms 运行，立即开始这个任务。
- 任务 Z 将每隔 25ms 运行，在 2ms 之后开始这个任务。
- 任务 Y 将每隔 30ms 运行，在 1ms 之后开始这个任务。

当计算所需的调度器间隔时，必须既要考虑任务间隔又要考虑初始的延迟。在这个例子中，现在需要找到 10、25、30、1 和 2 的最大公因数，这意味着现在需要 1ms 的调度器间隔。

建立可预测的和可靠的调度的准则

1. 为了精确调度，调度器的时标间隔应该设置为所有任务间隔的“最大公因数”（参见前面的内容）。

2. 所有任务的运行时间都应小于调度时标间隔，以保证调度程序总是能够在任何任务需要运行的时候调用它。往往用软件模拟来测量任务的运行时间。

3. 为了满足规则 2，在任何情况下，所有任务都必须设置“超时”，以使它们不会阻塞调度器。注意，在被调度任务中，这个规则往往能够在所需的地方结合循环超时或硬件超时模式。

请记住，这个规则同样适用于任何从被调度任务中调用的函数，包括编译程序厂家提供的所有程序库代码。在很多情况下，标准函数（例如 `printf()`）不包括超时特性。因此它们不能用于需要可预测性的场合。

4. 所有要求被调度的任务的运行总时间必须小于可用的处理器时间。当然，总的处理器时间必须既包括“任务运行时间”，又包括运行调度器的刷新和调度操作所需的“调度器运行时间”。

5. 应该使被调度的任务永远不会要求同时运行。即，应该使任务重叠减到最少。注意，

当所有任务的运行时间比调度器的时标间隔短得多，以及可以容忍任务抖动时，这个问题就无足轻重了。

可移植性

本模式说明的合作式调度器全部由 C 编写，而且只需要使用 8051 内核的硬件功能。因此，为 8051 系列中的某个芯片创建的调度器可以很容易地移植到所有其他 8051 芯片上使用。

这里说明的方法也可以毫不费力地运用到其他微控制器上。

优缺点小结

合作式调度器的总体优缺点概括如下：

- ◎ 这种调度器很简单，用少量代码即可实现。
- ◎ 基于调度器的应用程序本质上可预测、安全而且可靠。
- ◎ 调度器全部由 C 语言编写。它不是独立的系统，而是开发人员所编写代码的一部分。
- ◎ 调度器支持团队开发，因为往往能够基本上独立地开发单个任务，然后再将这些单个任务组合到最终的系统中。
- ◎ 设计阶段需要十分小心以保证快速地响应外部事件。
- ◎ 任务中使用多个中断是不安全的。系统中惟一的活动中断应该是用于驱动调度器本身的、有关定时器的中断。

相关的模式和替代解决方案

替代方案参见：

- 混合式调度器
- 单任务调度器
- 一年调度器
- 稳定的调度器
- 抢占式调度器，详细资料见第 13 章

例子：（通用的）调度器内核程序库

本书中的调度器全部使用这里给出的内核文件库构造。详细描述了（为不同的硬件、时钟频率及时标间隔而设计的）调度器的要求，然后在这个内核之上添加少量其他文件。

该内核库的大多数内容已经在本章的前面做了说明。源程序清单 14.12 和源程序清单 14.13 给出了完整的库，并说明了如何组合各种模块。

```
/*-----*
 * SCH51.h (v1.00)
 *-----*
 * 详细资料参见 SCH51.c
 *-----*/
```

```

#ifndef _SCH51_H
#define _SCH51_H
#include "Main.h"
// -----公用数据类型声明-----
// 可能的话，存储在 DATA 区，以供快速存取
// 每个任务的存储器总和是 7 个字节
typedef data struct
{
    // 指向任务的指针（必须是一个"void(void)"函数）
    void (code * pTask)(void);
    // 延迟（时标）直到函数将（下一次）运行
    // -详细说明参见 SCH_Add_Task()
    tWord Delay;
    // 在连续的运行之间的间隔（时标）
    // -详细说明参见 SCH_Add_Task()
    tWord Period;
    // 当任务需要运行时（由调度器）加 1
    tByte RunMe;
} sTask;
// -----公用的函数原型-----
// 调度器内核函数
void SCH_Dispatch_Tasks(void);
tByte SCH_Add_Task(void (code*) (void), const tWord, const tWord);
bit SCH_Delete_Task(const tByte);
void SCH_Report_Status(void);
// -----公用的常数-----
// 在任一时刻要求的任务最大数目
// 在程序的运行期间
//
// 每个新建项目都必须调整
#define SCH_MAX_TASKS (1)
#endif

/*
-----文件结束-----
*/

```

源程序清单 14.12 调度器内核库的一部分

```

/*
SCH51.C (v1.00)

***这里是调度器内核函数***
---这个函数可以用于所有 8051 芯片---
*** SCH_MAX_TASKS 必须由用户设置 ***
---参见 Sch51.h ---
***包括省电模式 ***
---必须确认省电模式被修改以适用于所选定的芯片---
--- (通常只有在扩展 8051 上，诸如 C515C、c509 等等才需要) ---
//
```

```

-----*/
#include "Main.h"
#include "Port.h"
#include "Sch51.h"
// -----公用变量定义-----
// 任务队列
sTask SCH_tasks_G[SCH_MAX_TASKS];
// 用来显示错误代码
// 错误代码的详细资料参见 Main.H
// 错误显示端口的详细资料参见 Port.H
tByte Error_code_G = 0;
// -----私有函数原型-----
static void SCH_Go_To_Sleep(void);
// -----私有变量-----
// 跟踪自从上一次记录错误以来的时间(见下文)
static tWord Error_tick_count_G;
// 上次的错误代码(在1分钟之后复位)
static tByte Last_error_code_G;
/*-
这里是“调度”函数。当一个任务(函数)需要运行时, SCH_Dispatch_Tasks()将运行它
这个函数必须被主循环(重复)调用
-----*/
void SCH_Dispatch_Tasks(void)
{
    tByte Index;
    // 调度(运行)下一个任务(如果有任务就绪)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        if (SCH_tasks_G[Index].RunMe > 0)
        {
            (*SCH_tasks_G[Index].pTask)();
            SCH_tasks_G[Index].RunMe -= 1; // 复位/降低 RunMe 标志
            // 周期性的任务将自动地再次运行
            // -如果这是个“单次”任务, 将它从队列中删除
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }
    // 报告系统状况
    SCH_Report_Status();
    // 这里调度器进入空闲模式
    SCH_Go_To_Sleep();
}
/*-
使任务(函数)每隔一定间隔或在用户定义的延迟之后运行
Fn_P -被调度的函数的名称[注意: 所有被调度的函数必须是“void, void”。
即函数必须没有参数, 并且返回类型为 void
-----*/

```

DELAY - 在任务第一次被运行之前的间隔（时标）
 PERIOD - 如果“PERIOD”为0，该函数将在由“DELAY”确定的时间被调用一次，
 如果 PERIOD 非 0，那么该函数将按 PERIOD 的值在确定的间隔被重复调用
 （下面的例子将有助于理解这些）

```

// PERIOD
返回值:
返回被添加任务在任务队列中的位置。如果返回值是 SCH_MAX_TASKS，  

那么该任务不能被加到队列中（空间不够）。如果返回值<SCH_MAX_TASKS，  

那么该任务被成功添加
注意：如果以后要删除任务，将需要这个返回值，参见 SCH_Delete_Task()
例子：
Task_ID = SCH_Add_Task(Do_X, 1000, 0);
使函数 Do_X() 在 1000 个调度器时标之后运行一次
Task_ID = SCH_Add_Task(Do_X, 0, 1000);
使函数 Do_X() 每隔 1000 个调度器时标运行一次
Task_ID = SCH_Add_Task(Do_X, 300, 1000);
使函数 Do_X() 每隔 1000 个时标定时运行一次。任务在 T=300 个时标时第一次运行，然后
1300 时标、2300 时标，等等
-----*/
tByte SCH_Add_Task(void * pFunction(),
                   const tWord DELAY,
                   const tWord PERIOD)
{
  tByte Index = 0;

  // 首先在队列中找到一个空隙（如果有的话）-
  while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
  {
    Index++;
  }
  // 是否已经到达队列的结尾？
  if (Index == SCH_MAX_TASKS)
  {
    // 任务对列已满
    //
    // 设置全局错误变量
    Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
    // 同时返回错误代码
    return SCH_MAX_TASKS;
  }

  // 如果能运行到这里，说明任务队列中有空间
  SCH_tasks_G[Index].pTask = pFunction;
  SCH_tasks_G[Index].Delay = DELAY;
  SCH_tasks_G[Index].Period = PERIOD;
  SCH_tasks_G[Index].RunMe = 0;
  return Index; // 返回任务的位置（以便以后删除）
}
/*-----*/

```

```

SCH_Delete_Task()
从调度器删除任务。注意，并不从存储器中删除相关的函数，仅仅是不再由调度器调用这个任务。
TASK_INDEX -任务索引。由 SCH_Add_Task() 提供
返回值：RETURN_ERROR 或 RETURN_NORMAL
-----*/
bit SCH_Delete_Task(const tByte TASK_INDEX)
{
    bit Return_code;
    if (SCH_tasks_G[TASK_INDEX].pTask == 0)
    {
        // 这里没有任务
        //
        // 设置全局错误变量
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;
        // ...同时返回错误代码
        Return_code = RETURN_ERROR;
    }
    else
    {
        Return_code = RETURN_NORMAL;
    }
    SCH_tasks_G[TASK_INDEX].pTask = 0x0000;
    SCH_tasks_G[TASK_INDEX].Delay = 0;
    SCH_tasks_G[TASK_INDEX].Period = 0;
    SCH_tasks_G[TASK_INDEX].RunMe = 0;
    return Return_code;      // 返回状态
}
/*-----*/
SCH_Report_Status()
用来显示错误代码的简单的函数
这个版本将在连接到端口的 LED 上显示错误代码
如果需要的话，可以修改为通过串行连接等方式来报告错误
Errors are only displayed for a limited period
(60000 ticks = 1 minute at 1ms tick interval).
错误只在有限的时间内显示（1ms 时标间隔时，60000 时标=1 分钟），此后错误代码被复位为 0。
这些代码可以很容易地修改为“永远”显示最近的错误，这在系统中可能更为合理。
更加详尽的资料请参见第 10 章
-----*/
void SCH_Report_Status(void)
{
#ifdef SCH_REPORT_ERRORS
    // 只在需要报告错误时适用
    // 检查新的错误代码
    if (Error_code_G != Last_error_code_G)
    {
        // 假定 LED 采用负逻辑
        Error_port = 255 - Error_code_G;

        Last_error_code_G = Error_code_G;

```

```

if (Error_code_G != 0)
{
    Error_tick_count_G = 60000;
}
· else
{
    Error_tick_count_G = 0;
}
}
else
{
    if (Error_tick_count_G != 0)
    {
        if (--Error_tick_count_G == 0)
        {
            Error_code_G = 0; // 复位错误代码
        }
    }
}
#endif
}
/*-----*/
SCH_Go_To_Sleep()
本调度器在时钟时标之间将进入“空闲模式”以节省功耗，下一个时钟时标将使处理器返回到正常操作状态
注意，如果这个函数由宏来实现，或简单地将这里的代码粘贴到“调度”函数中，那么可以带来少量的性能改善
然而，通过采用函数调用的方式来实现，可以在开发期间更容易地使用 Keil 硬件模拟器中的“性能分析器”来估计调度器的性能。这方面的例子参见第 14 章
***如果使用看门狗的话，可能需要禁止这个功能***
***根据硬件的需要修改***
/*-----*/
void SCH_Go_To_Sleep()
{
    PCON |= 0x01; // 进入空闲模式（通用 8051 版本）
    // 在 80c515 / 80c505 上，为了避免意外的触发，进入空闲模式需要两个连续的指令
    // PCON |= 0x01; // 进入空闲模式 (#1)
    // PCON |= 0x20; // 进入空闲模式 (#2)
}
/*-----*/
---- 文件结束 -----
/*-----*/

```

源程序清单 14.13 调度器内核库的一部分

例子：用于 8051/8052 的一个通用的 16 位定时的调度器

这个例子展示了如何使用（前面例子给出的）调度器内核文件作为一个完整的调度器的基

础。这里，使用定时器2来产生调度器“时标”。

注意，因为使用了定时器2，所以这些代码不能用于那些没有这个定时器的最基本的8051芯片。然而，它将在大多数当前的8051上运行。同时还要注意，CD上包括其他使用T0和T1来产生时标的调度器。

下面这个特例使用调度器来闪烁LED（1秒亮，1秒灭）无限循环。

除源程序清单14.12和源程序清单14.13给出的调度器内核文件之外，这里给出了所有的文件（源程序清单14.14~源程序清单14.18）。包括内核文件在内的所有文件都包含在CD上。

```
/*
-----*
Main.c (v1.00)

-----*
演示程序用于：
通用的16位自动重装调度器（使用T2）
假定采用12MHz的振荡器（-> 1ms时标间隔）
***所有定时单位为时标（而不是毫秒）***
所需链接程序选项：
OVERLAY (main ~ (LED_Flash_Update),
SCH_dispatch_tasks ! (LED_Flash_Update))
-----*/
#include "Main.h"
#include "2_01_12g.h"
#include "LED_flas.h"
/* ..... */
/* ..... */
void main(void)
{
    // 设置调度器
    SCH_Init_T2();
    // 为“Flash_LED”任务作准备
    LED_Flash_Init();
    // 添加“Flash LED”任务（1000ms亮，1000ms灭）
    // -定时单位为时标（1ms间隔）
    // （最大的间隔 / 延迟是65535个时标）
    SCH_Add_Task(LED_Flash_Update, 0, 1000);
    // 启动调度器
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
-----*
-----文件结束-----*
-----*/
```

源程序清单14.14 用于8051/8052的通用的16位定时的调度器的部分库

http://www.ustb.edu.cn/~cjl

```
/*
LED_flas.H (v1.00)
-----[资料]-----
-详细资料参见 LED_flas.C。
-----[函数]-----
// -----公用的函数原型-----
void LED_Flash_Init(void);
void LED_Flash_Update(void);
-----[文件结束]-----
*/
```

源程序清单 14.15 用于 8051/8052 的通用的 16 位定时的调度器的部分库

```
/*
LED_flas.C (v1.00)
-----[测试]-----
用于调度器的简单的“闪烁 LED”的测试函数
-----[变量]-----
#include "Main.h"
#include "Port.h"
#include "LED_flas.h"
// -----私有变量定义-----
static bit LED_state_G;
-----[函数]-----
LED_Flash_Init()
- See below.
void LED_Flash_Init(void)
{
    LED_state_G = 0;
}
-----[说明]-----
LED_Flash_Update()
在指定端口引脚上闪烁 LED(或产生脉冲给蜂鸣器, 等等)
必须按需要的闪烁频率的两倍计时。这样对于 1Hz 的
闪烁(0.5s 亮, 0.5s 灭)必须以 2Hz 计时
void LED_Flash_Update(void)
{
    // 使 LED 从灭变亮(反之亦然)
    if (LED_state_G == 1)
    {
        LED_state_G = 0;
        LED_pin = 0;
    }
    else
    {
        LED_state_G = 1;
        LED_pin = 1;
    }
}
```

```

        )
}

/*
-----文件结束-----
*/

```

源程序清单 14.16 用于 8051/8052 的通用的 16 位定时的调度器的部分库

```

/*
2_01_12g.C (v1.00)
-----
***这里是用于标准 8051/8052 的一种调度器***
***使用 T2 定时，16 位自动重装，12MHz 的振荡器->1ms (精确) 时标间隔***
*/
#include "2_01_12g.h"
// -----公用变量声明-----
// 任务队列 (参见 Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量
//
// 用来显示错误代码的端口以及有关错误代码的详细资料参见 Port.H
extern tByte Error_code_G;
/*
SCH_Init_T2()
调度器初始化函数。准备调度器数据结构并且设置定时器以所需的频率中断
必须在使用调度器之前调用这个函数
*/
void SCH_Init_T2(void)
{
    tByte i;
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // - SCH_Delete_Task() 将产生一个错误代码
    // (因为任务队列是空的)
    Error_code_G = 0;
    // 现在设置定时器 2
    // 自动重装的 16 位定时器函数
    // 晶振假定为 12MHz
    // 定时器 2 的精度是 0.000001s (1μs)
    // 要求的定时器 2 溢出为 0.001s (1ms)
    // -需要 1000 个定时器时标
    // 重装值为 65536 - 1000 = 64536 (十进制) = 0xFC18
    T2CON = 0x04;      // 加载定时器 2 的控制寄存器
    T2MOD = 0x00;      // 加载定时器 2 的模式寄存器
    TH2   = 0xFC;      // 加载定时器 2 的高位字节
    RCAP2H = 0xFC;     // 加载定时器 2 的重装捕捉寄存器的低位字节
    TL2   = 0x18;      // 加载定时器 2 的低位字节
}

```

```

RCAP2L = 0x18; // 加载定时器 2 的重装捕捉寄存器的低位字节
ET2     = 1;    // 使能定时器 2 中断
TR2     = 1;    // 启动定时器 2
}

/*
SCH_Start()
通过允许中断来启动调度器
注意：通常在添加了所有定期的任务之后调用，从而使任务保持同步
注意：应该只使能调度器中断！！！
*/
void SCH_Start(void)
{
    EA = 1;
}
/*
SCH_Update()
这是调度器的中断服务程序，初始化函数中的定时器设置决定了它的调用频率
这个版本由定时器 2 中断触发：定时器自动重装
*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    TF2 = 0; // 必须手工清除
    // 注意：计算单位为“时标”（不是毫秒）
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // 检测这里是否有任务
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // 任务需要运行
                SCH_tasks_G[Index].RunMe += 1; // “RunMe” 标志加 1
                if (SCH_tasks_G[Index].Repeat)
                {
                    // 调度定期的任务再次运行
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Repeat;
                }
            }
            else
            {
                // 还没有准备好运行，延迟减 1
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}

```

```

    }
}

/*
-----文件结束-----
*/

```

源程序清单 14.17 用于 8051/8052 的通用的 16 位定时的调度器的部分库

```

/*
2_01_12g.C (v1.00)

***这里是用于标准 8051/8052 的一种调度器***
***使用 T2 定时，16 位自动重装，12MHz 的振荡器-> 1ms (精确) 时标间隔***
*/
#include "2_01_12g.h"
// -----公用变量声明-----
// 任务队列 (参见 Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量
//
// 将用来显示错误代码的端口以及有关错误代码的详细资料参见 Port.H
extern tByte Error_code_G;
/*
SCH_Init_T2()
调度器初始化函数。准备调度器数据结构并且设置定时器以所需的频率中断频率,
必须在使用调度器之前调用这个函数
*/
void SCH_Init_T2(void)
{
    tByte i;
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // - SCH_Delete_Task()将产生一个错误代码
    //   (因为任务队列是空的)
    Error_code_G = 0;
    // 现在设置定时器 2
    // 自动重装的 16 位定时器函数
    // 晶振假定为 12MHz
    // 定时器 2 的精度是 0.000001s (1μs)
    // 要求的定时器 2 溢出为 0.001s (1ms)
    // -需要 1000 个定时器时标
    // 重装值为 65536 - 1000 = 64536 (十进制) = 0xFC18
    T2CON = 0x04;      // 加载定时器 2 的控制寄存器
    T2MOD = 0x00;      // 加载定时器 2 的模式寄存器
    TH2   = 0xFC;      // 加载定时器 2 的高位字节
}

```

```

RCAP2H = 0xFC; // 加载定时器 2 的重装捕捉寄存器的低位字节
TL2    = 0x18; // 加载定时器 2 的低位字节
RCAP2L = 0x18; // 加载定时器 2 的重装捕捉寄存器的低位字节
ET2    = 1;     // 使能定时器 2 中断
TR2    = 1;     // 启动定时器 2
}
/*
SCH_Start()
通过允许中断来启动调度器
注意：通常在添加了所有定期的任务之后调用，从而使任务保持同步
注意：应该只使能调度器中断！！！
*/
void SCH_Start(void)
{
EA = 1;
}
/*
SCH_Update()
这是调度器的中断服务程序，初始化函数中的定时器设置决定了它的调用频率
这个版本由定时器 2 中断触发，定时器自动重装
*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
tByte Index;
TF2 = 0; // 必须手工清除
// 注意：计算单位为“时标”（不是毫秒）
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
// 检测这里是否有任务
if (SCH_tasks_G[Index].pTask)
{
if (SCH_tasks_G[Index].Delay == 0)
{
// 任务需要运行
SCH_tasks_G[Index].RunMe += 1; // “RunMe” 标志加 1
if (SCH_tasks_G[Index].Period)
{
// 调度定期的任务再次运行
SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
}
}
else
{
// 还没有准备好运行，延迟减 1
SCH_tasks_G[Index].Delay -= 1;
}
}
}

```

```
    }  
}  
/*-----*  
---文件结束---*  
-----*/
```

源程序清单 14.18 用于 8051/8052 的通用的 16 位定时的调度器的部分库

例子：更多的 8051 调度器

不同 8051 调度器的详细资料参见 CD。

进阶阅读

许多有关这个领域的进阶阅读的建议请参阅第 13 章的“进阶阅读”一节。

学会以合作的方式思考

引言

在系统中使用合作式调度器有许多好处，其中之一是它将简化开发过程。然而，要最大限度的受益于调度器，需要学会以合作的方式思考。

例如：被调度的应用程序和桌面应用程序之间的一个主要区别是需要仔细地考虑定时和任务运行时间的问题。更具体地说，正如在第14章看到的，使用合作式调度器的应用程序的关键要求是：对于所有任务，无论在何种情况下，任务的运行时间Duration_{task}都必须满足以下条件：

Duration_{task} < 时标间隔

本章中的模式用来帮助满足这个条件。该模式将保证，如果任务不能在规定的时段内完成，就中止该任务。具体地说，这里给出了两个超时模式：

- 循环超时
- 硬件超时

循环超时

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。

问题

如何保证当系统在等待（诸如AD转换或串行数据传送）硬件操作完成时不会“挂起”？

背景知识

Philips 的 8XC552 是一种有许多片内模块的扩展 8051 芯片，它包括一个 8 通道的 10 位模数转换器。Philips 提供的应用手册 (AN93017) 描述了如何使用这个微控制器，该应用手册包

括下面的代码：

```
// 等待，直到 AD 转换结束（检查 ADCI）  
while ((ADCON & ADCI) == 0);
```

这样的代码有可能不可靠，因为在某些情况下，可能由于如下的原因导致系统被“挂起”：

- 如果模数转换器的初始化不正确，不能确定模数转换将被执行。
- 如果模数转换器的输入电压过高，那么可能根本就不运行。
- 如果没有正确初始化变量 ADCON 或 ADCI，那么可能不按要求运行。

当然，这样的问题不只是这个特殊的微控制器或者模数转换器才有。在嵌入式系统中，这样的代码也是常见的。

如果要求系统很可靠，则必须能够保证没有函数会这样挂起。循环超时提供了一种简单而有效的方法来提供这样的保证。

解决方案

循环超时很容易创建，其代码结构的基础是软件延迟，创建如下：

```
unsigned integer Timeout_loop = 0;  
...  
while (++Timeout_loop);
```

这个循环将连续运行直到变量 Timeout_loop 到达它的最大值 65 535（假定为 16 位整数），然后溢出。当溢出发生时，程序将继续。注意，不经过一些模拟研究或样机试验，很难确定这个延迟有多长。然而，可以确信循环最终确实会超时。

这样的循环并不是特别有用。然而，如果再次考虑“背景知识”中给出的模数转换器的例子，则可以很容易地扩展这个方法。回想一下那些最初的代码，如下所示：

```
// 等待，直到 AD 转换结束（检查 ADCI）  
while ((ADCON & ADCI) == 0);
```

这里是这些代码的改进版本，此时加入了一个循环超时：

```
tWord Timeout_loop = 0;  
// 索从模数转换器读取采样值  
// 等待，直到 AD 转换结束（检查 ADCI）  
// -简单的循环超时  
while (((ADCON & ADCI) == 0) && (Timeout_loop != 0));
```

注意：下面这个替代方案同样可用：

```
tWord Timeout_loop = 1;  
// 从模数转换器读取采样值  
// 等待，直到 AD 转换结束（检查 ADCI）  
// -简单的循环超时  
while (((ADCON & ADCI) == 0) && (Timeout_loop != 0))  
{
```

```
Timeout_loop++; // 在使用硬件模拟器时禁止...
}
```

第二种方法的优点是：当该代码运行在硬件模拟器上时，如果需要的话，循环超时可以容易地被注释掉。在两种情况下，现在都能确信循环将不会“永远”继续下去。注意，通过改变加载到循环变量中的初始值，能够改变循环超时的运行时间。源程序清单 15.1 中的文件 TimeoutL.H 出自 CD 上与本章有关的目录中。该文件中包括的一组常数非常近似地给出了特定的超时值。

```
/*
  TimeoutL.H (v1.00)

  用于基于 8051 系列芯片的简单的循环超时延迟
  *这些值并不精确-必须根据系统修改*
  -----
  // -----公用的常数-----
  // 改变这些值来改变循环的运行时间
  // /这些是在 12MHz、12 振荡周期/指令周期的 8051 上各种超时延迟的近似值
  // ***必须针对应用做调整***
  // ***定时随编译程序优化设置而变***
#define LOOP_TIMEOUT_INIT_001ms 65435
#define LOOP_TIMEOUT_INIT_010ms 64535
#define LOOP_TIMEOUT_INIT_500ms 14535
/-----
-----文件结束-----
*/
```

源程序清单 15.1 文件 TimeoutL.H

在下面几节里，举一个如何使用这些文件的例子。

硬件资源

循环超时不使用定时器，占用的 CPU 和存储器开销几乎可以忽略。

可靠性和安全性

使用循环超时能够以极小的成本，极大地改善可靠性和安全性。然而在实际系统中，硬件超时通常是一种更好的解决方案。

可移植性

循环超时将工作在任何平台。然而，应用不同的微控制器和编译器，得到的定时将显著变化。

优缺点小结

- ◎ 与没有任何形式的超时保护的执行代码相比，这要好得多。

- ⑥ 许多应用程序使用一个定时器用于产生 RS232 波特率，使用另一个定时器来运行调度器。在许多 8051 芯片中，没有更多的定时器可以用来实现硬件超时。在这些情况下，使用循环是实现有效的超时特性的唯一实用方式。
- ⑦ 定时很难计算，定时值不可移植。如果有一个空闲的定时器可用，硬件超时始终是一种最佳解决方案。

相关的模式和替代解决方案

正如在“可靠性和安全性”中提到的，硬件超时往往是循环超时的最好的替代方案。此外，硬件看门狗提供了一个替代方案。然而，比较起来该方案相当粗糙，只能检测系统级的错误（而不是任务级的）。

例子：循环超时代码的测试程序

正如已说明的，循环超时必须仔细手动调整以得到精确的延迟值。源程序清单 15.2 中的程序可用来测试这样的超时代码。

```
-----*
Main.C (v1.00)
-----*
测试超时循环
-----*-----*-----*
#include "Main.H"
#include "TimeoutL.H"
// 函数原型
void Test_1ms(void);
void Test_10ms(void);
void Test_500ms(void);
-----*-----*
void main(void)
{
    while(1)
    {
        Test_1ms();
        Test_10ms();
        Test_500ms();
    }
}
-----*-----*
void Test_1ms(void)
{
    tWord Timeout_loop = LOOP_TIMEOUT_INIT_001ms;
    // 简单的循环超时...
    while (++Timeout_loop != 0);
}
-----*-----*
void Test_10ms(void)
```

```

{
    tWord Timeout_loop = LOOP_TIMEOUT_INIT_010ms;
    // 简单的循环超时...
    while (++Timeout_loop != 0);
}

/*
void Test_500ms(void)
{
    tWord Timeout_loop = LOOP_TIMEOUT_INIT_500ms;
    // 简单的循环超时...
    while (++Timeout_loop != 0);
}
*/
----文件结束-----
*/

```

源程序清单 15.2 测试超时代码

程序在 Keil 硬件模拟器中运行，用于测试定时（如图 15.1 所示）。

记住：改变编译程序优化设置或者改变即使在表面上与之无关的程序其余部分也可能改变这些定时，因为这种改变可能会导致编译程序改变可用存储区的使用方式。

对试验代码进行最终测试时，在超时开始计数的时候将一个端口引脚置为 1，并在结束时清除。使用示波器来测量所得到的延迟。

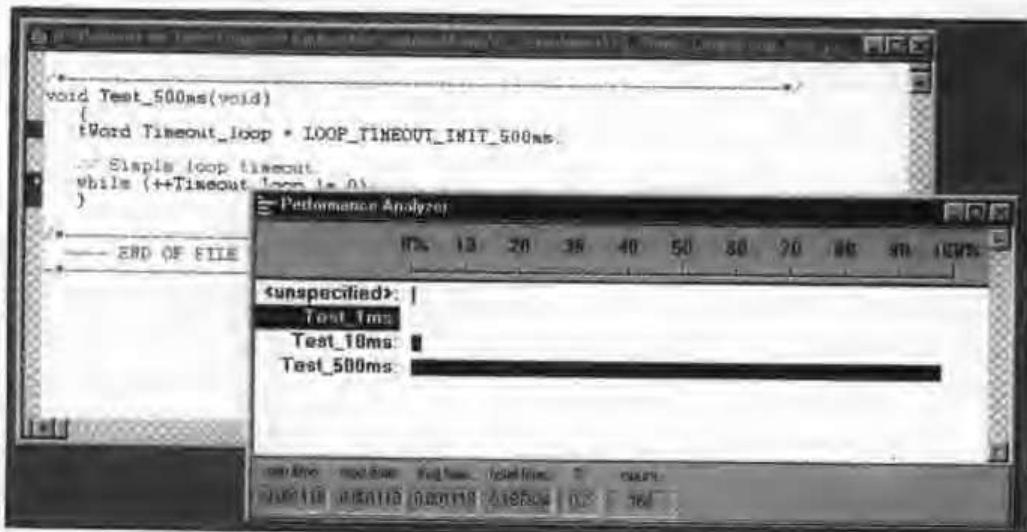


图 15.1 使用 Keil 硬件模拟器测试超时代码

例子：I²C 库中的循环超时

本书将在第 23 章详细讨论 I²C 总线。简而言之，I²C 是一种二线式的串行总线。两根线分

别称为串行数据线（SDA）和串行时钟线（SCL）（如图 15.2 所示）。当总线空闲时，SCL 和 SDA 都为高。

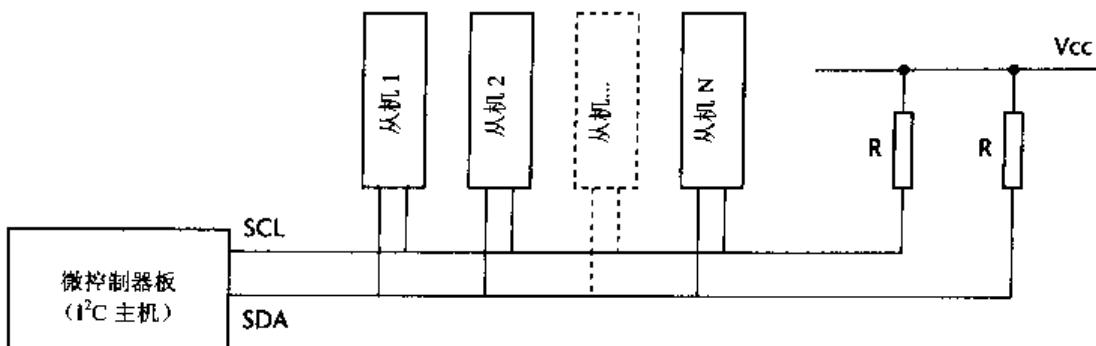


图 15.2 I²C 总线

这里讨论在 I²C 库的一个版本中是如何使用循环超时的。在数据传输的某些阶段，必须“同步时钟”。这意味着需要等待“时钟”线（被一个从机）拉高。一些 I²C 代码库包括类似以下的代码片段来实现这个功能：

```
// 同步时钟
while (_I2C_SCL == 0);
```

当然，由于本模式中介绍的各种原因，这是一种危险的方法。

下面的代码片段使用 1ms 超时的循环超时来改进这些代码：

```
#define LOOP_TIMEOUT_INIT_001ms 65435
...
tLong Timeout_loop = LOOP_TIMEOUT_INIT_001ms;
...
// 试图同步时钟
while (_I2C_SCL == 0) && (++Timeout_loop));
if (!Timeout_loop)
{
    return 1; // 错误：不满足超时条件
}
```

有关 I²C 总线和这个库的详细资料请参阅第 23 章。

进阶阅读

硬件超时

适用场合

- 使用 8051 系列微控制器开发一种嵌入式系统。

- 该系统使用调度器构造一种时间触发结构。

问题

如何生成准确定义的超时特性，以便当某个预期的事件没有发生时，在精确的时间（例如，0.5ms）内响应？

背景知识

相关背景材料参见循环超时。

解决方案

正如在硬件延迟看到的，可以为 8051 系列创建可移植并易于使用的延迟代码，如下所示：

```
// 定义用于 1ms 延迟的定时器 0/定时器 1 的重装值
#define PRELOAD_01ms (65536 - (tWord)(OSC_FREQ / (OSC_PER_INST * 1000)))
#define PRELOAD_01ms_H (PRELOAD_01ms / 256)
#define PRELOAD_01ms_L (PRELOAD_01ms % 256)
// ...
void Hardware_Delay_T0(const tLong MS)
{
    tLong ms;
    // 将定时器 0 配置为 16 位定时器
    TMOD &= 0xF0;      // 清除所有有关 T0 的位 (T1 不变)
    TMOD |= 0x01;      // 设置所需的 T0 的位 (T1 不变)
    ET0 = 0;           // 不使用中断
    for(ms = 0; ms < MS; ms++)
    {
        // 注意，每个循环的延迟值“大约”是 1ms
        // -PRELOAD_values 的详细资料参见 Delay_T0.H
        TH0 = PRELOAD_01ms_H;
        TL0 = PRELOAD_01ms_L;
        TF0 = 0;           // 清除溢出标志
        TR0 = 1;           // 启动定时器 0
        while (TF0 == 0);  // 循环直到定时器 0 溢出 (TF0 == 1)
        TF0 = 0;           // 停止定时器 0
    }
}
```

硬件超时对这种方法做了一些简单的改变，从而能够很容易地产生精确的超时延迟。例如：在循环超时中讨论了从 Philips 的 8XC552 微控制器中的模数转换器读的过程。这里是最初的、有潜在危险的代码：

```
// 等待，直到 AD 转换结束 (检查 ADCI)
while ((ADCON & ADCI) == 0);
```

这里是使用循环超时来解决该问题的一个应用例子：

```
tWord Timeout_loop = 0;
// 从模数转换器读取采样值
// 等待，直到 AD 转换结束（检查 ADCI）
// -简单的循环超时
while (((ADCON & ADCI) == 0) && (++Timeout_loop));
```

使用循环超时可以显著改善这些代码的可靠性，然而计算超时的运行时间却不容易。

这里是一个替代方案，以合理的精度提供了 10ms 的延迟，适用于整个 8051 系列（不用修改代码）：

```
// 将定时器 0 配置为 16 位定时器
TMOD &= 0xF0; // 清除所有有关 T0 的位 (T1 不变)
TMOD |= 0x01; // 设置所需的 T0 的位 (T1 不变)
ET0 = 0; // 不使用中断
// 简单的超时特性——大约 10ms
TH0 = PRELOAD_10ms_H; // PRELOAD 的详细资料参见 Timeout.H
TL0 = PRELOAD_10ms_L;
TF0 = 0; // 清除标志
TR0 = 1; // 启动定时器
while (((ADCON & ADCI) == 0) && !TF0);
```

源程序清单 15.3 中的文件 Timeout.H 中给出了各种可移植的 PRELOAD_宏指令，该文件包含在 CD 上。

注意：同样的 PRELOAD 值可以根据需要用于定时器 0 或定时器 1。

```
/*
  Timeout.H (v1.00)

  用于基于 8051 系列芯片的 T0/T1 的简单超时延迟。必须正确配置定时器以使用这些值。
  详细资料参见第 11 章的内容
*/
// -----公用的常数-----
// 用于简单（硬件）超时的定时器 T_value 值
// - 定时器是 16 位，手动重装（“单次”）
//
// 注意，这些宏指令是可移植的，然而定时是“近似的”，如果需要精确定时，必须手动复核
//
// 定义用于 50 微秒延迟的定时器 0/定时器 1 的初始值
#define T_50micros (65536 - (tWord)((OSC_FREQ / 26000) / (OSC_PER_INST)))
#define T_50micros_H (T_50micros / 256)
#define T_50micros_L (T_50micros % 256)
// 定义用于 500 微秒延迟的定时器 0/定时器 1 的初始值
#define T_500micros (65536 - (tWord)(OSC_FREQ / (OSC_PER_INST * 2000)))
#define T_500micros_H (T_500micros / 256)
#define T_500micros_L (T_500micros % 256)
// 定义用于 1ms 延迟的定时器 0/定时器 1 的初始值
```

```

#define T_01ms (65536 - (tWord)(OSC_FREQ / (OSC_PER_INST * 1000)))
#define T_01ms_H (T_01ms / 256)
#define T_01ms_L (T_01ms % 256)
//
// 定义用于 10ms 延迟的定时器 0/定时器 1 的初始值
#define T_10ms (65536 - (tWord)(OSC_FREQ / (OSC_PER_INST * 100)))
#define T_10ms_H (T_10ms / 256)
#define T_10ms_L (T_10ms % 256)
//
// 定义用于 30ms 延迟的定时器 0/定时器 1 的初始值
#define T_30ms (65536 - (tWord)(OSC_FREQ / (OSC_PER_INST * 33)))
#define T_30ms_H (T_30ms / 256)
#define T_30ms_L (T_30ms % 256)
/*-----*-
-----文件结束----- *
-----*/
```

源程序清单 15.3 文件 Timeout.H

硬件资源

硬件超时需要使用一个定时器。

可靠性和安全性

硬件超时是本书所讨论的最可靠的超时结构。

可移植性

和所有基于定时器的模式一样，这些代码可以很容易地移植到 8051 系列的其他芯片上。同时，它也可以被移植到其他微控制器上。

优缺点小结

- ☺ 使用硬件超时可以得到精确的超时延迟。
- ☹ 系统可用的定时器的数目非常有限。然而，当使用合作式调度器时，任务以合作的方式运行，同一个定时器可以同时用于几个任务。

相关的模式和替代解决方案

不需要使用任何定时器硬件的替代方案参见循环超时。

此外，硬件看门狗提供了一个替代方案。然而，比较起来，该方案相当粗糙，只能检测系统级的错误（而不是任务级的）。

例子：测试硬件超时

源程序清单15.4说明了使用Keil硬件模拟器利用一些硬件超时得到的延迟(参见图15.3)。

```
-----*
Main.C
-----
    测试超时循环
-----*/
#include "Main.H"
#include "TimeoutH.H"
// 函数原型
void Test_50micros(void);
void Test_500micros(void);
void Test_1ms(void);
void Test_5ms(void);
void Test_10ms(void);
void Test_15ms(void);
void Test_20ms(void);
void Test_50ms(void);
// TIMEOUT程序变量&TIMEOUT代码(这里为空)
#define TIMEOUT 0xFF
tByte Error_code_G;
-----
void main(void)
{
    while(1)
    {
        Test_50micros();
        Test_500micros();
        Test_1ms();
        Test_5ms();
        Test_10ms();
        Test_15ms();
        Test_20ms();
        Test_50ms();
    }
}
-----*/
void Test_50micros(void)
{
    // 将定时器0配置为16位定时器
    TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
    TMOD |= 0x01; // 设置所需的T0的位(T1不变)
    ET0 = 0; // 不使用中断
    // 简单的超时特性-大约50μs
    TH0 = T_50micros_H; // T_的详细资料参见Timeout.H
    TL0 = T_50micros_L;
    TF0 = 0; // 清除标志
}
```

```
TR0 = 1; // 启动定时器
while (!TF0);

TR0 = 0;
// 正常情况下需要报告超时 TIMEOUT
// (这里的这个检验用于演示)
if (TF0 == 1)
{
    // 运行超时
    Error_code_G = TIMEOUT;
}
}

/*
void Test_500micros(void)
{
    // 将定时器0配置为16位定时器
    TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
    TMOD |= 0x01; // 设置所需的T0的位(T1不变)
    ET0 = 0; // 不使用中断
    // 简单的超时特性——大约500μs
    TH0 = T_500micros_H; // T_的详细资料参见Timeout.H
    TL0 = T_500micros_L;
    TF0 = 0; // 清除标志
    TR0 = 1; // 启动定时器
    while (!TF0);
    TR0 = 0;
    // 正常情况下需要报告超时 TIMEOUT
    // (这里的这个检验用于演示)
    if (TF0 == 1)
    {
        // 运行超时
        Error_code_G = TIMEOUT;
    }
}

/*
void Test_1ms(void)
{
    // 将定时器0配置为16位定时器
    TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
    TMOD |= 0x01; // 设置所需的T0的位(T1不变)
    ET0 = 0; // 不使用中断
    // 简单的超时特性——大约1ms
    TH0 = T_01ms_H; // T_的详细资料参见Timeout.H
    TL0 = T_01ms_L;
    TF0 = 0; // 清除标志
    TR0 = 1; // 启动定时器
    while (!TF0);
    TR0 = 0;
```

```
// 正常情况下需要报告超时 TIMEOUT
// (这里的这个检验用于演示)
if (TF0 == 1)
{
    {
        // 运行超时
        Error_code_G = TIMEOUT;
    }
}
/*-----*/
void Test_5ms(void)
{
    {
        // 将定时器0配置为16位定时器
        TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
        TMOD |= 0x01; // 设置所需的T0的位(T1不变)
        ET0 = 0; // 不使用中断
        // 简单的超时特性——大约5ms
        TH0 = T_05ms_H; // T_的详细资料参见Timeout.H
        TL0 = T_05ms_L;
        TF0 = 0; // 清除标志
        TR0 = 1; // 启动定时器
        while (!TF0);
        TR0 = 0;
        // 正常情况下需要报告超时 TIMEOUT
        // (这里的这个检验用于演示)
        if (TF0 == 1)
        {
            {
                // 运行超时
                Error_code_G = TIMEOUT;
            }
        }
    }
}
/*-----*/
void Test_10ms(void)
{
    {
        // 将定时器0配置为16位定时器
        TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
        TMOD |= 0x01; // 设置所需的T0的位(T1不变)
        ET0 = 0; // 不使用中断
        // 简单的超时特性——大约10ms
        TH0 = T_10ms_H; // T_的详细资料参见Timeout.H
        TL0 = T_10ms_L;
        TF0 = 0; // 清除标志
        TR0 = 1; // 启动定时器
        while (!TF0);
        TR0 = 0;
        // 正常情况下需要报告超时 TIMEOUT
        // (这里的这个检验用于演示)
        if (TF0 == 1)
        {
            {
                // 运行超时
                Error_code_G = TIMEOUT;
            }
        }
    }
}
```

```

        Error_code_G = TIMEOUT;
    }
}

/*-----*/
void Test_15ms(void)
{
    // 将定时器0配置为16位定时器
    TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
    TMOD |= 0x01; // 设置所需的T0的位(T1不变)
    ET0 = 0; // 不使用中断
    // 简单的超时特性——大约10ms
    TH0 = T_15ms_H; // T_的详细资料参见Timeout.H
    TL0 = T_15ms_L;
    TF0 = 0; // 清除标志
    TR0 = 1; // 启动定时器
    while (!TF0);
    TR0 = 0;
    // 正常情况下需要报告超时TIMEOUT
    // (这里的这个检验用于演示)
    if (TF0 == 1)
    {
        // 运行超时
        Error_code_G = TIMEOUT;
    }
}

/*-----*/
void Test_20ms(void)
{
    // 将定时器0配置为16位定时器
    TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
    TMOD |= 0x01; // 设置所需的T0的位(T1不变)
    ET0 = 0; // 不使用中断
    // 简单的超时特性——大约20ms
    TH0 = T_20ms_H; // T_的详细资料参见Timeout.H
    TL0 = T_20ms_L;
    TF0 = 0; // 清除标志
    TR0 = 1; // 启动定时器
    while (!TF0);
    TR0 = 0;
    // 正常情况下需要报告超时TIMEOUT
    // (这里的这个检验用于演示)
    if (TF0 == 1)
    {
        // 运行超时
        Error_code_G = TIMEOUT;
    }
}

/*-----*/
void Test_50ms(void)

```

```

{
// 将定时器0配置为16位定时器
TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
TMOD |= 0x01; // 设置所需的T0的位(T1不变)
ET0 = 0; // 不使用中断
// 简单的超时特性——大约10ms
TH0 = T_50ms_H; // T_的详细资料参见Timeout.H
TL0 = T_50ms_L;
TF0 = 0; // 清除标志
TR0 = 1; // 启动定时器
while (!TF0);
TR0 = 0;
// 正常情况下需要报告超时 TIMEOUT
// (这里的这个检验用于演示)
if (TF0 == 1)
{
// 运行超时
Error_code_G = TIMEOUT;
}
}
/*-----文件结束-----*/

```

源程序清单 15.4 测试硬件超时**例子：产生基于超时的延迟**

可以很容易地扩展“超时”技术，用作硬件延迟的一种替代方案，参见源程序清单 15.5。

```

Void Delay_50micros(void)
{
// 将定时器0配置为16位定时器
TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
TMOD |= 0x01; // 设置所需的T0的位(T1不变)
ET0 = 0; // 不使用中断
// 简单的超时特性——大约50μs
TH0 = T_50micros_H; // T_的详细资料参见Timeout.H
TL0 = T_50micros_L;
TF0 = 0; // 清除标志
TR0 = 1; // 启动定时器
while (!TF0);
TR0 = 0;
}

```

源程序清单 15.5 使用硬件超时来实现延迟

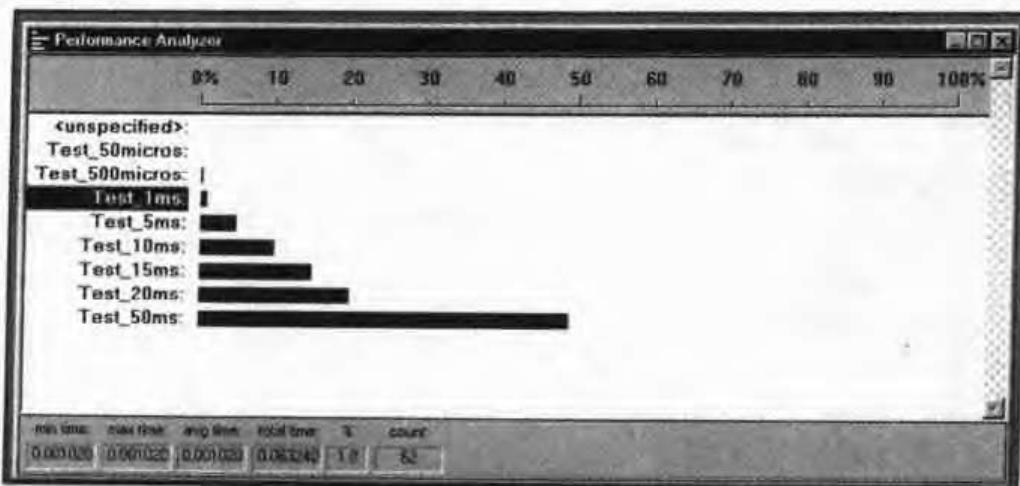


图 15.3 使用 Keil 硬件模拟器测试硬件超时

进阶阅读

Chapter 16

面向任务的设计

引言

许多成功的合作式调度应用程序都应用了本章介绍的两个模式所包含的主要设计特性，这两个模式分别是：

- 多级任务，讨论了将（以很长间隔调度的）长任务转换为（以极短的间隔调度的）非常短的任务的方法。
- 多状态任务，讨论了将多个任务替换为一个任务的方法，该任务根据系统的当前状态执行不同的操作。

多级任务

适用场合

- 使用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。

问题

如何保证长任务不影响调度器的正常运行？

背景知识

相关背景资料参见合作式调度器。

解决方案

正如已经看到的，与抢占式调度器乃至混合式调度器相比，合作式调度器有许多优点。通常只要有可能，总是希望使用合作式解决方案。然而，使用这种解决方案在设计上的主要挑战是如何保证在下一个调度器时标出现之前结束每个任务。实现该目标的一种方法是使用超时

(参见第 15 章的内容)。

另一个方法是使用多级任务。为了有助于理解多级任务的必要性，考虑一个例子(由 Font 1996, 第 13 章改编而来)。假定作为用于金属熔炉的“备用的”温度监测器的一部分，要求以固定的 5s 间隔向台式 PC 发送温度采样值(如图 16.1 所示)。

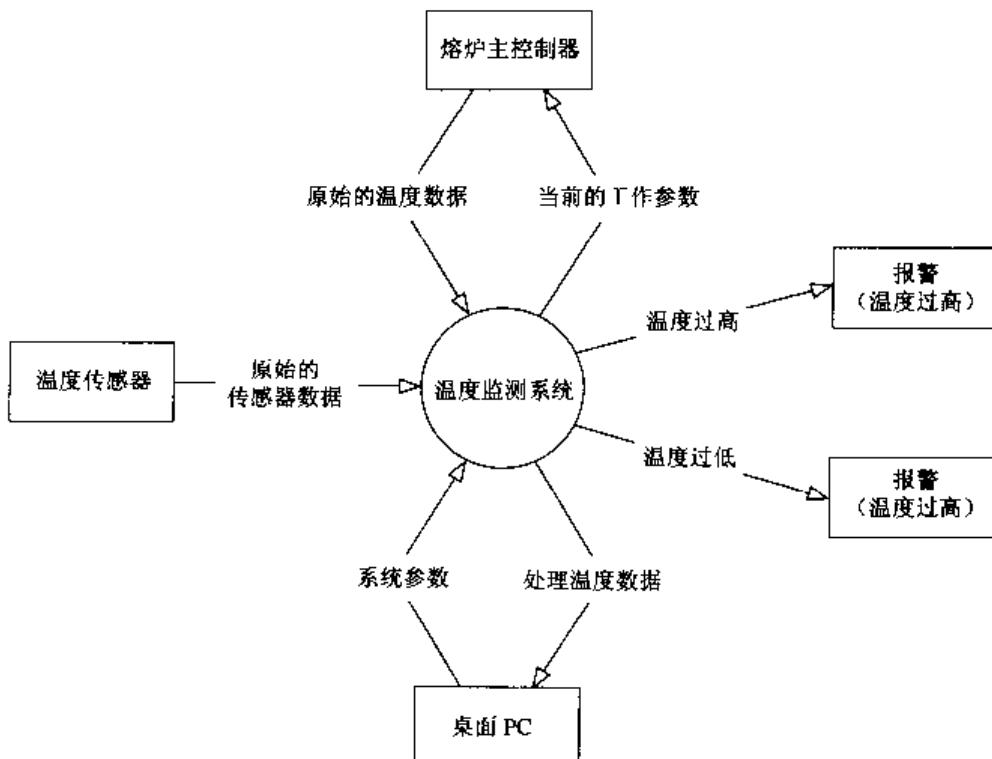


图 16.1 温度监测系统的部分设计

要求的 PC 输出的形式如图 16.2 所示。

```

TIME: 13.10.00 TEMPERATURE (Celsius): 2310
TIME: 13.10.05 TEMPERATURE (Celsius): 2313
TIME: 13.10.10 TEMPERATURE (Celsius): 2317
TIME: 13.10.15 TEMPERATURE (Celsius): 2318
  
```

图 16.2 温度监测系统的输出

乍看起来，实现以要求的 5s 间隔向 PC 发送信息的一种显而易见的方式是建立一个函数(称作 `Send_Temp_Data()`)，该函数将被调度器每隔 5s 运行一次。

然而，这种解决方案可能相当糟糕。问题是需要使用 `Send_Temp_Data()` 向 PC 发送至少 43 个字符。在常见的 9600 波特率(详细资料参见第 18 章)下，发送 1 个字符将需要大约 1ms。因此，该任务的运行时间为 40ms。如果使用这种(长)任务，需要将系统时标间隔设置为大于 40ms。这有可能对系统总体的响应性能造成不良影响。

采用多级任务的可选方案可以避免这个问题。只须将待发送的数据存储在缓冲区中，而不是立刻发送所有数据。每隔（比方说）10ms，调度一个任务来检查该缓冲区，如果有字符准备发送，则发送该字符。这样，所有要求发送的 43 个字符的数据将在 0.5s 内发送给 PC。如果必要的话（很少有这种需要），可以通过每隔 1ms 调度检查缓冲区的任务来减少发送的时间。注意，因为不必等待每个字符被发送，所以从缓冲区发送数据的过程将是非常快的（一般为零点几毫秒）。

总的说来，在这个系统中使用多级任务使我们有可能使用非常短的时标间隔，因此将更为有效地利用微控制器的处理能力。

这里作为例子使用的“RS232”库将在第 18 章详细讨论。

硬件资源

一般说来，使用多级任务能够更有效地利用微控制器的处理能力。

可靠性和安全性

使用多级任务将可以使用更短的时标间隔，从而使系统响应速度更快。

可移植性

这种方法适用于各种嵌入式系统。

优缺点小结

- ◎ 使用多级任务可以使用较短的时标间隔，因此有助于使系统响应速度更快。
- ◎ 使用多级任务能够更有效地利用微控制器的处理能力。

相关的模式和替代解决方案

在本书中，这种基本结构应用于各种模式。包括 PC 连接（RS-232）、LCD 字符面板和开关接口（软件）。

例子：测量转速

假设希望测量一个转轴的转速，并将结果显示在一个（多路复用的）LED 显示上。这个例子可能用于汽车应用或工业应用中。

正如将在第 30 章看到的，测量转速的一种有效方法是将一个合适的旋转编码器连接在轴上（如图 16.3 所示），对固定长度时间内（比方说 100ms 或 1s）产生的脉冲进行计数。根据旋转编码器的详细资料，由计数值可以计算平均转速。

有很多人试图在调度平台上应用以下基本方法：

- 建立一个运行时间为（比方说）100ms 的任务。
- 在这个任务中，对脉冲进行计数。

- 在任务的结尾设置（全局）变量 Pulse_count_G 的值。
- 使用该全局变量刷新 LED 显示。

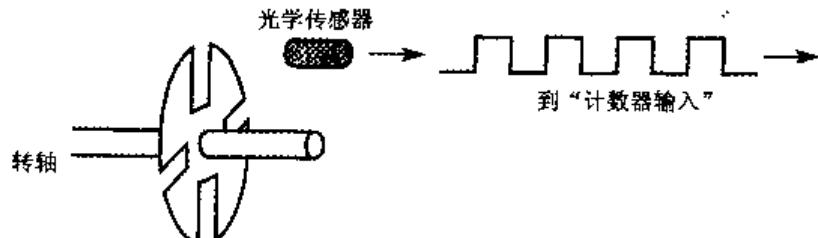


图 16.3 测量转速

这种方法的问题是测量速度需要的 100ms 是一个较长的任务运行时间，在许多嵌入式系统中可能很难做到。例如，假设这里希望在多路复用的 LED 上显示该速度，有可能必须每隔 5ms 刷新 LED 以避免闪烁（详细资料参见多路复用 LED 显示）。

毫无疑问，合作式调度器不能支持每隔 5ms 调用运行时间为 100ms 的任务。

为了解决这个问题，考虑使用多级任务的解决方案。第一种方案将在任务中等待编码器端的脉冲并对其进行计数。然而，这是不必要的。定时器 0 或定时器 1（或定时器 2）将对外部引脚上的脉冲（严格地说，是脉冲的下降沿）计数，这个过程无需用户干涉，不产生中断而且不与其他程序冲突。以此为基础，将转速测量系统设计为一个多级任务，如下所示：

- 建立一个非常短 (<0.1ms) 的任务。
- 每隔（比方说）100ms 调度一次这个任务，与第一个例子一致。
- 在这个任务中，读取当前的脉冲计数，将结果存储在一个全局变量中（使程序的其他地方也可以使用）。
- 复位脉冲计数为 0。
- 100ms 以后重复，等等。

该方法的详细资料将在第 30 章讨论，并介绍不需要定时器硬件而实现类似功能的解决方案。

例子：LCD 库

假设希望刷新 LCD 显示。正如将在 LCD 字符面板看到的，刷新一个典型的 LCD 显示的每个字符大约需要 0.5ms，因此刷新全部 40 个字符的显示大约需要 20ms。往往会因为时间太长而难以接受。

然而，假设每隔 20ms 调度一个 LCD_Update() 函数，且每次运行时只刷新一个显示位置。则在最坏的情况下，将需要总共 800ms 来刷新整个显示，但能够同时完成许多其他的任务。此外，在大多数情况下，只有一部分显示会变化。如果跟踪那些需要修改的字符，通常使用一个每隔 100ms 调用、运行时间为 0.5ms 的多级任务，就完全能够使显示一直保持最新。总的说来，这个例子采用多级任务的解决方案将使整个系统的结构发生很大的变化。

多级 LCD 库的详细资料将在 LCD 字符面板中给出。

进阶阅读

多状态任务

适用场合

- 使用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。

问题

如何根据系统当前状态，在系统中用一个任务替换多个任务，执行所要求的不同行为（以及为什么有时这么做是个好主意）？

背景知识

相关背景资料参见合作式调度器和多状态任务（译者注：应该是多级任务）。

解决方案

多状态任务采用的系统总体结构在许多设计得很好的嵌入式系统中都能见到。

为了理解这种结构的必要性，考虑一个简单的洗衣机控制系统（如图 16.4 所示）。

下面是一些有关所期望的系统运行方式的简短描述：

1. 用户在选择面板上选择一种洗衣程序（例如“羊毛制品”、“棉制品”）。
 2. 用户按下“启动”开关。
 3. 门被锁上。
 4. 水阀被打开，向滚筒里放水。
 5. 如果洗衣程序要用到洗衣粉，洗衣粉舱口盖被打开。当洗衣粉被投放后，关闭洗衣粉舱口盖。
 6. 当检测到“水位已到”时，关闭水阀。
 7. 如果洗衣程序要用到热水，打开热水器。当达到需要的温度时，关掉热水器。
 8. 启动电动机来旋转滚筒。然后电动机将进行一系列运动，以各种速度正反向旋转来清洗衣物。执行运动的精确设置取决于用户选定的洗衣程序。洗涤周期结束后，电动机将停止。
 9. 打开抽水泵排出滚筒中的水。当滚筒排空后，关掉抽水泵。
- 这些经过简化的描述仅用于举例，然而对这里来说已经足够了。
- 以这些描述为基础，将确定实现这个系统需要的一些函数，其暂定的列表如图 16.5 所示。

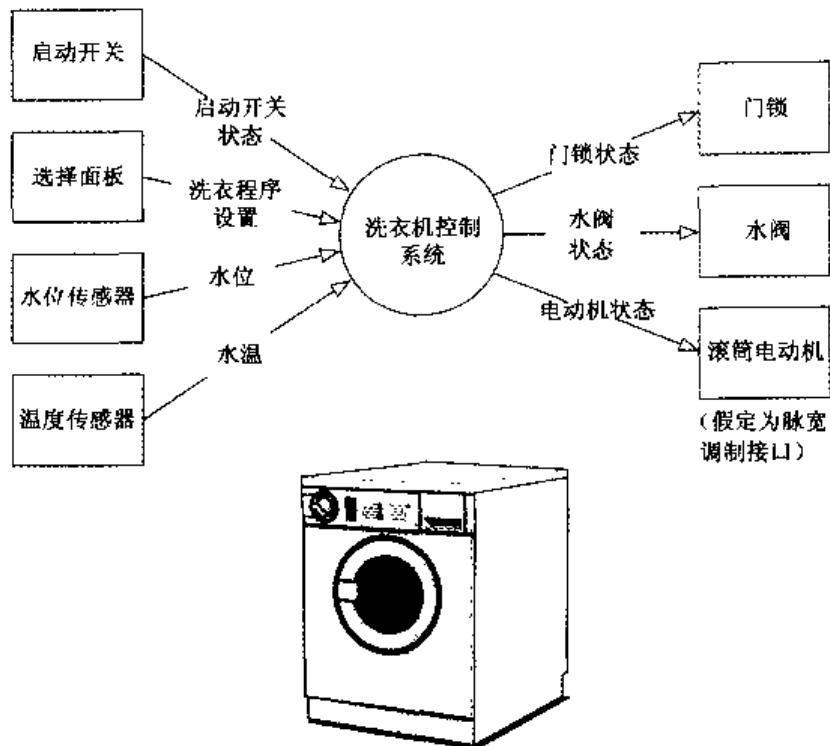


图 16.4 家用洗衣机控制系统接口关系的数据流图

- | | |
|---|--|
| <ul style="list-style-type: none"> ● Read_Selector_Dial() ● Read_Start_Switch() ● Read_Water_Level() ● Read_Water_Temperature() | <ul style="list-style-type: none"> ● Control_Detergent_Hatch() ● Control_Door_Lock() ● Control_Motor() ● Control_Pump() ● Control_Water_Heater() ● Control_Water_Value() |
|---|--|

图 16.5 用于开发一个洗衣机控制系统的暂定函数列表

现在，假设希望确定哪些任务将被（合作式）调度以实现这个系统。根据上面的列表，很容易做出结论，图 16.5 列出的每个函数都应该成为系统中的任务。虽然这样可以工作，但可能将是一种复杂而笨重的系统实现。

用一个例子来说明其原因：函数 Control_water_heater()。只有在洗涤周期的某个时刻才需要加热水。因此，如果把它作为一个任务并且每隔（比方说）100ms 调度它一次，则需要实现如下：

```
void TASK_Control_Water_Heater(void)
{
    if (Switch_on_water_heater_G == 1)
    {

```

```

Water_heater = ON;
return;
}
// 关闭加热器
Water_pin = OFF
}

```

这个任务在运行时将检查标志。如果需要加热水，则将开始加热，否则便停止加热过程。以这种方法创建程序有两个问题：

- 将有大量的任务（在更有实际价值的系统中任务会更多），大多数类似的任务实际上只做极少的工作。对于没有外部存储器的应用程序来说，这是个大问题，因为每个任务都将占用有限的存储器（RAM）资源。
- 是否这些任务中的某一个将设置这个标志（Switch_on_water_heater_G），或是该系统其他任务要求的类似标志，这些问题一点也不清晰。

实际上，在这个应用程序以及许多类似的应用程序中，需要的是一个“系统刷新”任务。正如将看到的，这是一个将被定时调度的任务，并将在需要的时候按要求调用诸如 Control_Water_Heater() 这样的函数。

在洗衣机系统中，这个系统刷新任务如源程序清单 16.1 中的程序所示。

```

/*-----*/
void WASHER_Update(void)
{
    static tWord Time_in_state;
    switch (System_state_G)
    {
        case STRART:
        {
            // 仅用于演示
            P1 = (tByte) System_state_G;
            // 锁门
            WASHER_Control_Door_Lock(ON);
            // 开始向滚筒内放水
            WASHER_Control_Water_Valve(ON);
            // 放洗衣粉（如果有的话）
            if (Detergent_G[Program_G] == 1)
            {
                WASHER_Control_Detergent_Hatch(ON);
            }
            // 准备转向下一个状态
            System_state_G = FILL_DRUM;
            Time_in_state_G = 0;
            break;
        }
        case FILL_DRUM:
        {
            // 仅用于演示

```

```

P1 = (tByte) System_state_G;
// 保持这个状态直到滚筒满
// 注意：这里包含超时功能
if (++Time_in_state_G >= MAX_FILL_DURATION)
{
    // 现在滚筒应该已经放满水...
    System_state_G = ERROR;
}
// 检查水位
if (WASHER_Read_Water_Level() == 1)
{
    // 滚筒是满的
    // 程序是否需要热水？
    if (Hot_Water_G[Program_G] == 1)
    {
        WASHER_Control_Water_Heater(ON);
        // 准备转向下一个状态
        System_state_G = HEAT_WATER;
        Timer_in_state_G = 0
    }
    else
    {
        // 只使用冷水
        // 准备转向下一个状态
        System_state_G = WASH_01;
        Time_in_state_G = 0;
    }
}
break;
}
case HEAT_WATER;
{
    // 仅用于演示
    P1 = (tByte) System_state_G;
    // 保持这个状态直到水热
    // 注意，这里包含超时功能
    if (++Time_in_state_G >= MAX_WATER_HEAT_DURATION)
    {
        // 现在水应该已经加热...
        System_state_G = ERROR;
    }
    // 检查水温
    if (WASHER_Read_Water_Temperature() == 1)
    {
        // 水温满足要求
        // 准备转向下一个状态
        System_state_G = WASH_01;
        Time_in_state_G = 0;
    }
}

```

```
        break;
    }
case WASH_01:
{
    // 仅用于演示
    P1 = (tByte) System_state_G;
    // 所有洗衣程序包含在 WASH_01 内
    // 滚筒慢慢旋转以保证衣服完全湿透
    WASHER_Control_Motor(ON);
    if (++Time_in_state >= WASH_01_DURATION)
    {
        System_state_G = WASH_02;
        Time_in_state = 0;
    }
    break;
}
// 这里省略其余的洗涤阶段...
case WASH_02:
{
    // 仅用于演示
    P1 = (tByte) System_state_G;
    break;
}
case ERROR:
{
    // 仅用于演示
    P1 = (tByte) System_state_G;
    break;
}
}
```

源程序清单 16.1 使用单任务实现洗衣机控制系统的例子

源程序清单 16.1 是一个典型的多状态任务的例子，这种结构的最简形式描述如下：

- 系统要求使用许多不同的函数。
- 函数始终以同样的顺序被调用。
- 函数根据需要由一个任务调用。

注意，也有一些常见的变化。例如，函数可能并不是始终以同样的顺序被调用，调用某组函数的精确顺序常常取决于用户的选择或系统中的其他输入。

硬件资源

这种结构能够非常有效地使用系统资源。

可靠性和安全性

没有特定的可靠性或安全问题。

可移植性

这种高级模式的可移植性非常好。

优缺点小结

- ☺ 多状态任务的简单结构满足许多嵌入式系统的需要。

相关的模式和替代解决方案

多级任务、单任务调度器及一年调度器，都只需极少的 CPU、存储器和功耗要求，但提供非常简单而有效的系统结构。

例子：交通灯

假设希望建立一个用于驱动交通三色灯的系统。将以常见的顺序使用传统的“红”、“黄”和“绿”灯（如图 16.6 所示）。

源程序清单 16.2 介绍了如何建立一个多状态任务来实现这个系统。每隔 1s 调度一次“刷新”函数。

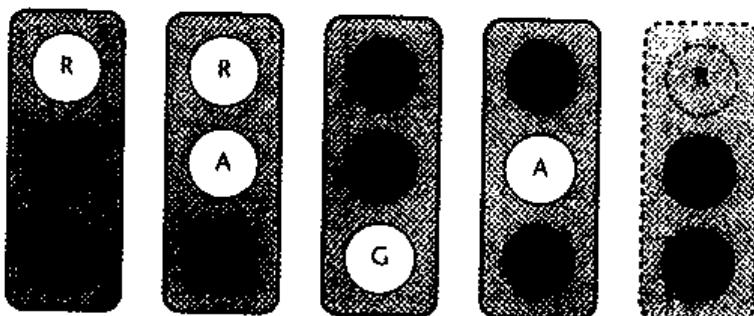


图 16.6 交通灯系统要求的红、黄、绿灯的点亮顺序

```
/*
T_lights.C (v1.00)
-----
交通灯控制程序 (测试版 1.0)
*/
#include "Main.h"
#include "Port.h"
#include "T_lights.h"
// ----- 私有的常数 -----
// 这里可以很容易地改变逻辑
#define ON 0
```

```
#define OFF 1
// 每个可能的点亮状态的时间
// (时间单位为秒——必须每秒调用一次刷新任务)
//
#define RED_DURATION (10)
#define RED_AND_AMBER_DURATION (10)
// 注意:
// GREEN_DURATION 必须等于 RED_DURATION
// AMBER_DURATION 必须等于 RED_AND_AMBER_DURATION
#define GREEN_DURATION RED_DURATION
#define AMBER_DURATION RED_AND_AMBER_DURATION
// -----私有变量-----
// 系统状态
static eLight_State Light_state_G;
/*-----*/
TRAFFIC_LIGHTS_Init()
Prepare for the scheduled traffic light activity.
*/
void TRAFFIC_LIGHTS_Init(const eLight_State START_STATE)
{
    Light_state_G = START_STATE; // 从机是绿色; 主机是红色
}
/*-----*/
TRAFFIC_LIGHTS_Update()
必须每秒调用一次
*/
void TRAFFIC_LIGHTS_Update(void)
{
    static tWord Time_in_state;
    switch (Light_state_G)
    {
        case RED:
        {
            Red_light = ON;
            Amber_light = OFF;
            Green_light = OFF;
            if (++Time_in_state == RED_DURATION)
            {
                Light_state_G = Red_And_Amber;
                Time_in_state = 0;
            }
            break;
        }
        case Red_And_Amber:
        {
            Red_light = ON;
            Amber_light = ON;
            Green_light = OFF;
            if (++Time_in_state == RED_AND_AMBER_DURATION)
```

```
{  
    Light_state_G = Green;  
    Time_in_state = 0;  
}  
break;  
}  
case Green:  
{  
    Red_light = OFF;  
    Amber_light = OFF;  
    Green_light = ON;  
    if (++Time_in_state == GREEN_DURATION)  
    {  
        Light_state_G = AMBER;  
        Time_in_state = 0;  
    }  
break;  
}  
case AMBER:  
{  
    Red_light = OFF;  
    Amber_light = ON;  
    Green_light = OFF;  
    if (++Time_in_state == AMBER_DURATION)  
    {  
        Light_state_G = RED;  
        Time_in_state = 0;  
    }  
break;  
}  
}  
}  
/*-----*  
----文件结束-----  
-----*/
```

源程序清单 16.2 使用一个多状态任务实现交通灯控制系统

进阶阅读

混合式调度器

引言

正如在第13章讨论的，合作式调度器为各种嵌入式系统提供了一个可预测的平台。

在一些场合，可能需要在合作式调度器结构中加入抢占式调度器的一些特性，并小心地加以控制。混合式调度器用来满足这种要求。

混合式调度器

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。

问题

怎样建立并使用混合式调度器？

背景知识

正如已经看到的，在嵌入式系统开发中，时间触发的合作式结构具有许多优点。实际上，贯穿本书已经说明：只要有可能，通常希望使用这种结构。然而，从实用角度出发，合作式调度器也有局限性。尤其是在一些情况下，可能既需要运行长任务（例如每隔 1000ms 运行 100ms 时间），同时又需要运行频繁的短任务（例如每隔 1ms 运行 0.1ms）。在合作式系统中这两个要求会有冲突。无论在何种情况下，所有任务的任务运行时间—— $\text{Duration}_{\text{task}}$ 必须满足以下条件：

$\text{Duration} < \text{Tick interval}$

在本书的各个模式中已经讨论了既能满足“频繁”任务的需要又能满足“长”任务需要的各种方法。例如，通过使用更快的处理器（参见第 3 章）或更快的振荡器（第 4 章）可以缩短任务的运行时间。也可以使用超时（参见第 15 章）或创建多级任务（参见第 16 章）。

然而，这些解决方案不能总是满足要求。本书将进一步讨论两个更强大的解决方案：

- 第6篇将讨论如何使用多处理器系统结构。正如将看到的，使用多个微控制器可以提供真正的多任务处理功能。
- 本章将讨论创建和使用混合式调度器。

在第13章中已经简要介绍了混合式调度器，这种结构综合了抢占式调度器和合作式调度器的特性，允许根据要求使（长）任务优先。然而，这种方式始终是受控的，不需要依赖复杂的上下文切换程序，以及任务间复杂的通信机制（参见图17.1）。

混合式调度器

- 混合式调度器提供了有限的多任务处理功能。

运行：

- 支持多个合作式调度的任务。
- 支持一个抢占式任务（可以中断合作式任务）。

实现：

- 这种调度器很简单，用少量代码即可实现。
- 该调度器必须同时为两个任务分配存储器。
- 该调度器通常完全由高级语言（比如“C”）实现。
- 该调度器不是一种独立的系统，它成为开发人员的代码的一部分。

性能：

- 对外部事件的响应速度快。

可靠性和安全性：

- 只要设计小心，可以和纯粹的合作式调度器一样可靠。

图17.1 混合式调度器的特性

解决方案

警告！

当已经建立了合作式调度器（参见第14章）时，只需做非常少的程序改动就可以把它修改为混合式调度器。可以看到，许多情况下设计人员使用混合式设计仅仅是因为它很容易实现，这通常是错误的。

在混合式调度器中，已经没有了（纯粹的）合作式的调度特性。这可能对设计过程及最终系统的可靠性带来深远的影响。

在试图使用这种调度器之前参考“可靠性和安全性”！

我们希望实现什么

这里描述的混合式调度器形式和（纯粹的）合作式调度器的区别如下：

- 不再要求所有任务都必须在时标间隔之间完成。一个（或更多）的合作式任务的运行时间可以大于时标间隔。

- 与前面讨论的合作式调度器一样，可以调度任意个合作式任务。然而，还可以同时调度一个抢占式任务。
- 抢占式任务抢先（中断）合作式任务。
- 一旦抢占式任务开始运行，将一直运行到完成。即，抢占式任务不能被任何合作式任务打断。抢占式任务可以看作是“最高优先权”的任务。

同时注意：

- 与完全的抢占式解决方案相比，只有一个抢占式任务，而且该任务连续运行直到完成，这将极大地简化系统结构。尤其是不需要实现上下文切换机制。这意味着：（一）该结构仍然非常简单，（二）运行环境可以仍然全部由 C 来实现。
- 和完全的抢占式平台相比，抢占式任务连续运行直到完成同样将简化任务间的通信，将在后面提供更详细的相关资料。
- 应该只有一个短任务（最长的运行时间为时标间隔的 50%，尽可能的短）可以抢占运行，否则将削弱系统的总体性能。

如何实现

在了解如何建立混合式调度器之前，首先回顾合作式调度器的实现。

源程序清单 17.1 显示了合作式调度器的“刷新”函数。

```
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    TF2=0; // 必须手工清除
    // 注意，计算单位为“时标”（不是 ms）
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // 检测这里是否有任务
        if (SCH_tasks_G[Index].Task_p)
        {
            if (SCH_tasks_G[Index].Delay)==0)
            {
                // 任务需要运行
                SCH_tasks_G[Index].RunMe +=1:// 运行标志加 1
                if (SCH_tasks_G[Index].Period)
                {
                    // 调度定期的任务再次运行
                    SCH_tasks_G[Index].Delay=SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // 还没有准备好运行，延迟减 1
                SCH_tasks_G[Index].Delay -=1;
            }
        }
    }
}
```

```

    }
}
}
}
```

源程序清单 17.1 合作式调度器的“刷新”函数

合作式调度器的程序使用在源程序清单 17.2 中说明的调度器数据类型。

```

// 如果可能的话，存储在 DATA 区，以供快速存取
// 每个任务的存储器总和是 7 个字节
typedef data_struct
{
    // 指向任务的指针（必须是一个“void(void)”函数）
    void(code*Task_p)(void)
    // 延迟（时标）直到函数将（下一次）运行
    // - -详细说明参见 SCH_Add_Task()
    tWord Delay;
    // 在连续两次运行之间的间隔（时标）
    // -详细说明参见 SCH_Add_Task()
    tWord Period;
    // 当任务需要运行时（由调度器）加 1
} sTask;
```

源程序清单 17.2 用于合作式调度器的用户定义的数据类型

源程序清单 17.3 显示了混合式调度器的“刷新”函数。

```

void hSCH_Update (void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    TF2 = 0; // 必须手工清除
    // 注意，计算单位为时标(不是 ms)
    for (Index = 0;Index < HSCH_MAX_TASKS;Index++)
    {
        // 检测这里是否有任务
        if(hsch_tasks_G[Index].pTask)
        {
            if(hSCH_tasks_G[Index].Delay == 0)
            {
                // 任务需要运行
                //
                if (hSCH_tasks_G[Index].Co_op)
                {
                    // 如果是合作式任务，RunMe 标志加 1
                    hSCH_tasks_G[Index].RunMe +=1;
                }
            }
            else
            {
                // 如果是抢占式任务，立即运行它
                (*Hsch_tasks_G[Index].pTask)();
                // 运行任务
            }
        }
    }
}
```

```

hSCH_task_G[Index].RunMe -=1; // RunMe 标志复位/减 1
RunMe flag
// 周期性的任务将自动再次运行
// -如果这是个“单次”任务，将它从队列中删除
if (hSCH_tasks_G[Index].period == 0)
{
    hSCH_tasks_G[Index].pTask = 0;
}
}
if(hSCH_tasks_G[Index].Period)
{
    // 调度定期的任务再次运行
    hSCH_tasks_G[Index].Delay = hSCH_tasks_G[Index].Period;
}
}
else
{
    // 还没有准备好运行，延迟减 1
    hSCH_tasks_G_[Index].Delay -= 1
}
}
}

```

源程序清单 17.3 混合式调度器的“刷新”函数

混合式调度器的程序使用源程序清单 17.4 中所示的调度器数据类型。

```

// 如果可能的话，存储在 DATA 区，以供快速存取
// 每个任务的存储器总和是 8 个字节
typedef data struct
{
    // 指向任务的指针(必须是一个“void(void)”函数)
    void(code * Task_p)(void);
    // 延迟(时标)直到函数将(下一次)运行
    // -详细说明参见 SCH_Add_Task()
    tWord Delay;
    // 在连续的运行之间的间隔(时标)
    // -详细说明参见 SCH_Add_Task()
    tWord Period;
    // 当任务需要运行时(由调度器)加 1
    // 如果任务是合作式的，设置为 1
    // 如果任务是抢占式的，设置为 0
    tByte Co_op;
} sTask

```

源程序清单 17.4 混合式调度器的用户定义的数据类型

需要在 SCH_Add_Task() 函数调用(图 17.2)中使用额外的参数，才能将参数 Co_op 设置为合适的值。

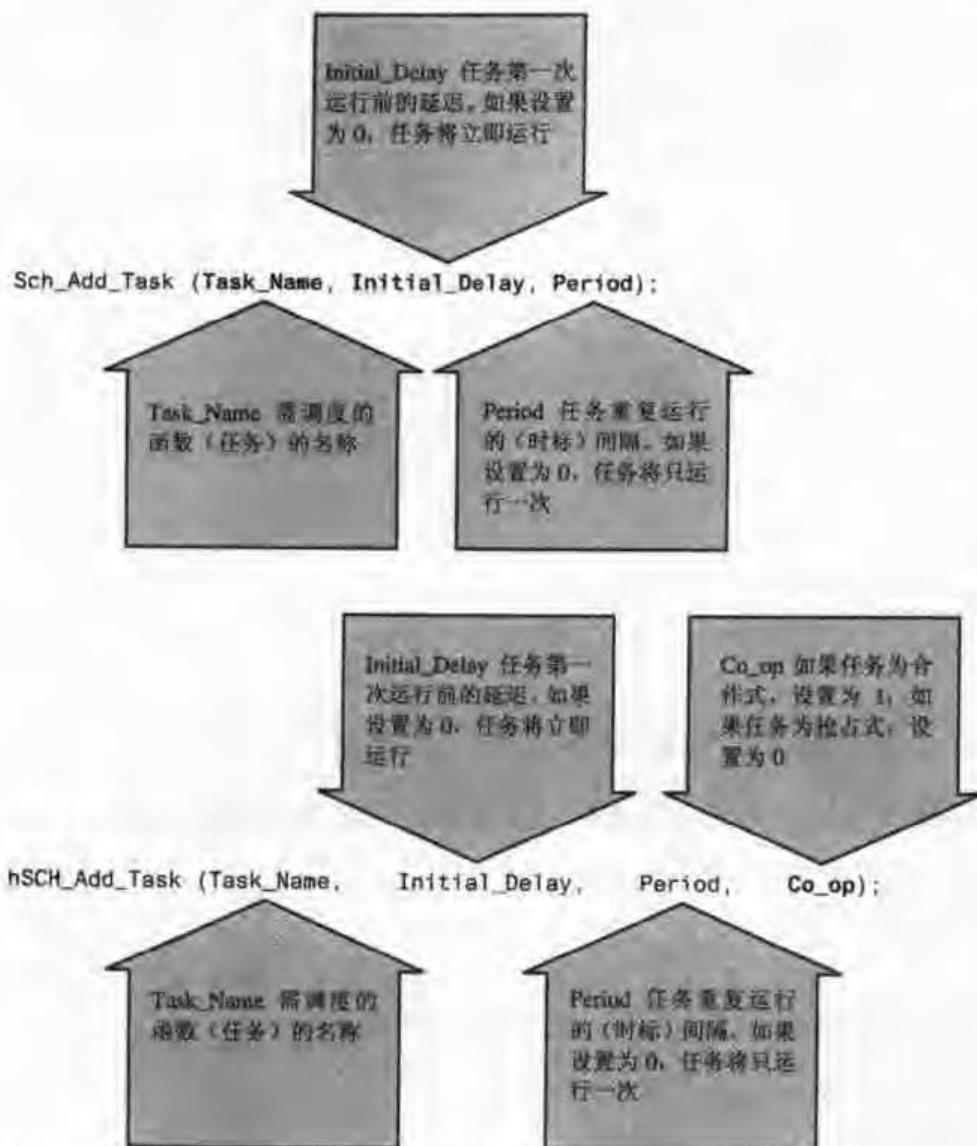


图 17.2 “添加任务”函数的参数。[上面]合作式调度器的版本。[下面]混合式调度器的版本

可靠性和安全性

请在使用这种调度器之前仔细阅读本节内容！

多任务处理（抢占式）系统的问题

第13章讨论了抢占式调度器的问题，包括有关程序关键段的问题。正如先前看到的，在抢占式系统中处理这样的关键代码段主要有两种可行的方法：

- 在开始关键段之前通过禁止调度器来“暂停”调度，当离开关键段时重新允许调度器中断。

- 使用“锁”（或其他形式的“信号灯机制”）来实现类似的结果。

在第13章中可以看到，这两种解决方案都有缺点。尤其是，如果任务在读取锁标志（并发现它是开放的）之后而在设置该标志（来指示资源被使用）之前被中断，第二种解决方案就将出现问题（参见图17.3）。

```
//...
// 准备进入关键段
// 检查锁是否被开放
if (Lock == LOCKED)
{
    return;
}

// 锁被开放
// 进入关键段

// 设置锁
Lock = LOCKED;

// 这里是关键代码 //
```

图17.3 抢占式平台上在检查和设置之间出现上下文切换而导致的问题

混合式调度器不存在该问题，原因如下：

- 就抢占式任务来说，因为它们不能被中断，所以不会在检查和加锁之间出现中断。
- 就合作式任务（有可能被中断）而言，也不会出现该问题，然而其原因稍有不同。

合作式任务可能“在检查和设置锁”之间被间断，然而只有抢占式任务才能中断它。如果抢占式任务中断进来时发现关键段是放开的，将设置锁^①，使用该资源，然后清除锁。即，它将运行直到完成。接着合作式任务将继续，系统的状态将与被抢占式任务中断之前的状态一样。因此不会违反互斥规则。

注意，混合式调度器以一种简单的方法解决了存取关键段代码的问题。与完全的抢占式调度器不同，不要求创建复杂的代码“锁”或“信号灯”之类的结构。

使用混合式调度器的最安全的方法

使用混合式调度器的最可靠的方法如下：

- 建立数量满足要求的合作式任务。很可能因为一个或多个任务的运行时间会大于时标间隔，所以需要使用混合式的调度器。混合式的调度器的实现是安全的，然而必须保证任务不重叠。
- 实现一个抢占式任务。该任务一般（然而并非一定）将在每个时标间隔调用。这种任务的一种不错的应用是用来检查错误或紧急事件。因此，即使系统的主要用途是运行一个1000ms的合作式任务，也可以用抢占式任务来保证系统能够在10ms内响应外部事件。

^① 严格地说，设置锁标志是不必要的，因为不可能有中断。

部事件。

- 记住，抢占式任务能够中断合作式任务。如果有关键代码段，需要实现一种简单的锁定机制（以下例子将详细说明）。
- 抢占式任务必须简短（最长的运行时间为时标间隔的 50%，应尽可能的短），否则将极大削弱系统的总体性能。
- 在所有运行状态下仔细测试该系统，监测错误。

可移植性

混合式调度器和合作式调度器一样容易移植。不要求使用汇编语言或其他的非标准语言特性。

优缺点小结

混合式调度器的优缺点小结概括如下：

- ☺ 能够同时处理“很少发生的长任务”和“频繁发生的短任务”，而合作式调度器不能同时处理。
- ☺ 如果按照指导使用，将是安全并且可预测的。
- ☹ 必须小心使用。

相关的模式和替代解决方案

- 参见合作式调度器
- 参见 SCU 调度器（本地）
- 参见 SCC 调度器

例子：使用 T2 的混合式调度器（1ms 时标）

本节给出了一个完整的混合式调度器的例子（源程序清单 17.5~源程序清单 17.12）。

注意：使用一种简单的闭锁机制来保证任何时刻只有一个任务访问 LED 端口。

```
/*
-----*
Port.H (v1.00)

-----*
项目 2_01_12h 的“端口头文件”（参见第 10 章）
-----*
// ----- Sch51.C -----
// 如果不需要错误报告，将这一行注释掉
#ifndef SCH_REPORT_ERRORS
#define SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1
```

```
#endif
// ----- LED_hyb.C -----
// 两个任务共用这个端口 (P2)
// sbit LED_short_pin = P2^2;
// 将把它当作共享资源，并“锁定”它
#define LED_long_port P2
/*-----*
---文件结束-----
*-----*/
```

源程序清单 17.5 混合式调度器的例子的一部分

```
/*-----*
Main.c (v1.00)

-----*
混合式调度器的演示程序：
/// HYBRID SCHEDULER ///
通用的 16 位自动重装混合式调度器 (使用 T2)
假定采用 12MHz 的振荡器 (->1ms 时标间隔)
***所有定时单位为时标 (而不是 ms) ***
要求的链接程序选项 (详细资料参见第 14 章):
OVERLAY (main ~ (LED_Short_Update, LED_Long_Update),
hSCH_dispatch_tasks ! (LED_Short_Update, LED_Long_Update))
*-----*/
#include "Main.h"
#include "2_01_12h.h"
#include "LED_Hyb.h"
/* ..... */ /* .. */
void main(void)
{
    // 设置调度器
    hSCH_Init_T2();
    LED_Short_Init();
    // 添加“短”任务 (1000ms 亮, 1000ms 灭)
    // 这是一个抢占式任务
    hSCH_Add_Task(LED_Short_Update, 0, 1000, 0);
    // 添加“长”任务 (运行时间 10s)
    // 这是一个合作式任务
    hSCH_Add_Task(LED_Long_Update, 0, 20000, 1);
    // 启动调度器
    hSCH_Start();
    while(1)
    {
        hSCH_Dispatch_Tasks();
    }
}
```

```

    }
/*
-----文件结束-----
*/

```

源程序清单 17.6 混合式调度器的例子的一部分

```

/*
  2_01_12h.H (v1.00)

  - 详细资料参见 2_01_12h.c
*/
#include "Main.h"
#include "hSCH51.H"
// -----公用的函数原型-----
void hSCH_Init_T2(void);
void hSCH_Start(void);

/*
-----文件结束-----
*/

```

源程序清单 17.7 混合式调度器的例子的一部分

```

/*
  2_01_12h.C (v1.00)

  ***这是用于标准 8051/8052 的一种混合式调度器***
  ***使用 T2 定时，16 位自动重装***
  ***12MHz 的振荡器->1ms (精确) 时标间隔***
*/
#include "2_01_12h.h"
// -----公有变量声明-----
// 任务队列 (参见 Sch51.C)
extern sTaskH hSCH_tasks_G[hSCH_MAX_TASKS];
// 错误代码变量
//
// 将用来显示错误代码的端口以及错误代码的详细资料参见 Main.H
extern tByte Error_code_G;
/*
  hSCH_Init_T2()
  调度器初始化函数。准备调度器数据结构并且设置定时器以所需的频率中断。
  必须在使用调度器之前调用这个函数。
*/
void hSCH_Init_T2(void)
{
    tByte i;
    for (i = 0; i < hSCH_MAX_TASKS; i++)
    {
        hSCH_Delete_Task(i);
    }
}
```

```

        }

// 复位全局错误变量
// - hSCH_Delete_Task() 将产生一个错误代码（因为任务队列是空的）
Error_code_G = 0;
// 现在设置定时器 2
// 自动重装的 16 位定时器功能
// 晶振假定为 12MHz
// 定时器 2 的精度是 0.000001s (1 微秒)
// 要求的定时器 2 溢出为 0.001s (1ms)
// --需要 1000 个定时器时标
// 重装值为 65536-1000=64536 (十进制) =0xFC18
T2CON = 0x04; // 加载定时器 2 的控制寄存器
T2MOD = 0x00; // 加载定时器 2 的模式寄存器
TH2 = 0xFC; // 加载定时器 2 的高位字节
RCAP2H = 0xFC; // 加载定时器 2 的重装捕捉寄存器的高位字节
TL2 = 0x18; // 加载定时器 2 的低位字节
RCAP2L = 0x18; // 加载定时器 2 的重装捕捉寄存器的低位字节
ET2 = 1; // 使能定时器 2 中断
TR2 = 1; // 启动定时器 2
}

/*-----*
hsCH_Start()
通过允许中断启动调度器。
注意：通常在添加了所有定期的任务之后调用，从而使任务保持同步
注意：应该只使能调度器中断!!!
-----*/
void hSCH_Start(void)
{
    EA = 1;
}

/*-----*
hsCH_Update
这是调度器的中断服务程序。hSCH_Init() 中的定时器设置决定了调用频率，
这个版本是由定时器 2 中断触发的。定时器自动重装
-----*/
void hSCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    TF2 = 0; // 必须手工清除
    // 注意：计算单位为“时标”（不是毫秒）
    for (Index = 0; Index < hSCH_MAX_TASKS; Index++)
    {
        // 检测这里是否有任务
        if (hSCH_tasks_G[Index].pTask)
        {
            if (hSCH_tasks_G[Index].Delay == 0)
            {
                // 任务需要运行
                //
            }
        }
    }
}

```

```

if (hSCH_tasks_G[Index].Co_op)
{
    // 如果是合作式任务, RunMe 标志加 1
    hSCH_tasks_G[Index].RunMe += 1;
}
else
{
    // 如果是抢占式任务, 立即运行它
    (*hSCH_tasks_G[Index].pTask)(); // 运行任务
    hSCH_tasks_G[Index].RunMe -= 1; // RunMe 标志复位/减 1
    // 周期性的任务将自动的再次运行
    // -如果这是个“单次”任务, 将它从队列中删除
    if (hSCH_tasks_G[Index].Period == 0)
    {
        hSCH_tasks_G[Index].pTask = 0;
    }
}
if (hSCH_tasks_G[Index].Period)
{
    // 调度定期的任务再次运行
    hSCH_tasks_G[Index].Delay = hSCH_tasks_G[Index].Period;
}
else
{
    // 还没有准备好运行, 延迟减 1
    hSCH_tasks_G[Index].Delay -= 1;
}
}

/*
-----文件结束-----
*/

```

源程序清单 17.8 混合式调度器的例子的一部分

```

/*
-----*
* hSCH51.h (v1.00)
*
* -详细资料参见 hSCH51.C
*-----*/
#ifndef _hSCH51_H
#define _hSCH51_H
#include "Main.h"
// -----公用数据类型声明-----
// 如果可能的话, 存储在 DATA 区, 以供快速存取
// 每个任务的存储器总和是 8 个字节
typedef data struct

```

```

{
// 指向任务的指针（必须是一个“void(void)”函数）
// 延迟（时标）直到函数将（下一次）运行
// - 详细说明参见 SCH_Add_Task()
tWord Delay;
// 在连续的运行之间的间隔（时标）。
// - 详细说明参见 SCH_Add_Task()
tWord Period;
// 当任务需要运行时（由调度器）加1
tByte RunMe;
// 如果任务是合作式的，设置为1
// 如果任务是抢占式的，设置为0
tByte Co_op;
} sTaskH;
// -----公用的函数原型-----
// 调度器内核函数
void hSCH_Dispatch_Tasks(void);
tByte hSCH_Add_Task(void (code *)(), tWord, tWord, bit);
bit hSCH_Delete_Task(tByte);
void hSCH_Report_Status(void);
// -----公用的常数-----
// 在程序的运行期间任一时刻请求的任务最大数目
//
// 每个新建项目都必须调整
#define hSCH_MAX_TASKS (2)
#endif
/*-----文件结束-----*/

```

源程序清单 17.9 混合式调度器的例子的一部分

```

/*-----*
hSCH51.C (v1.00)

-----*/
///混合式调度器内核///
***这里是调度器内核函数***
---这个函数可以用于所有8051芯片---
*** hSCH_MAX_TASKS 必须由用户设置 ***
---参见"Sch51.h" ---
***包括省电模式***
---必须确认省电模式被修改以适用于所选定的芯片（通常只有在使用扩展8051，---
---诸如c515c、c509等等才需要）---
/*-----*/
#include "Main.h"
#include "Port.h"
#include "hSch51.h"
// -----公有变量定义-----

```

```

// 任务队列
sTaskH hSCH_tasks_G[hSCH_MAX_TASKS];
// 用来显示错误代码
// 错误代码的详细资料参见 Main.H
// 关于错误端口的详细资料参见“私有常数”（如下）
// - 更详细的资料参考本书第 14 章的内容
tByte Error_code_G = 0;
// -----私有函数原型-----
static void hSCH_Go_To_Sleep(void);
// -----私有变量-----
// 记住自从上一次记录错误以来的时间（见下文）
static tWord Error_tick_count_G;
// 上次的错误代码（在 1 分钟之后复位）
static tByte Last_error_code_G;
/*-----*
这是“调度”函数。当一个任务(函数)需要运行时, hSCH_Dispatch_Tasks()将运行它
这个函数必须被主循环(重复)调用
-----*/
void hSCH_Dispatch_Tasks(void)
{
    tByte Index;
    // 调度(运行)下一个任务(如果有任务就绪)
    for (Index = 0; Index < hSCH_MAX_TASKS; Index++)
    {
        // 只调度合作式任务
        if ((hSCH_tasks_G[Index].Co_op) && (hSCH_tasks_G[Index].RunMe > 0))
        {
            (*hSCH_tasks_G[Index].pTask)(); // 运行任务
            hSCH_tasks_G[Index].RunMe -= 1; // RunMe 标志复位/减 1
            // 周期性的任务将自动的再次运行
            // - 如果这是个“单次”任务, 将它从队列中删除
            if (hSCH_tasks_G[Index].Period == 0)
            {
                // 比通过调用来删除任务更快
                hSCH_tasks_G[Index].pTask = 0;
            }
        }
    }
    // 报告系统状况
    hSCH_Report_Status();
    // 这里调度器进入“空闲模式”
    hSCH_Go_To_Sleep();
}
/*-----*
hSCH_Add_Task()

```

使任务（函数）每隔一定间隔或在用户定义的延迟之后运行

Fn_P - 将被调度的函数的名称

注意：所有被调度的函数必须是“void, void”，即，函数必须没有参数，并且返回类型为 void。

Del - 在任务第一次被运行之前的间隔（时标）

per - Per-如果“Per”为 0，则该函数只被调用一次，由“Del”确定调用的时间。

如果 Per 非 0，那么该函数将按 Per 的值所确定的间隔被重复调用（下面的例子将有助于理解这些）。

Co-op - 如果是合作式任务设置为 1，如果是抢占式任务设置为 0。

RETN: 返回被添加任务在任务队列中的位置。如果返回值是 hSCH_MAX_TASKS，那么该任务不能被加到队列中（空间不够）。如果返回值<hSCH_MAX_TASKS，那么该任务被成功添加。

注意：如果以后要删除任务，将需要这个返回值，参见 hSCH_Delete_Task()。

例子：

```
Task_ID = hSCH_Add_Task(Do_X,1000,0,0);
```

使函数 Do_X() 在 1000 个调度器时标之后运行一次（抢占式任务）。

```
Task_ID = hSCH_Add_Task(Do_X,0,1000,1);
```

使函数 Do_X() 每隔 1000 个调度器时标运行一次（合作式任务）。

```
Task_ID = hSCH_Add_Task(Do_X,300,1000,0);
```

使函数 Do_X() 每隔 1000 个调度器时标运行一次。任务将首先在 T=300 个时标时被执行，然后是 1300 个时标、2300 个时标，等等（抢占式任务）。

```
-----*/  
tByte hSCH_Add_Task(void* Fn_p) // 任务函数指针  
{  
    tWord Del, // 直到任务第一次运行时的时标数  
    tWord Per, // 重复运行之间的时标数  
    bit Co_op) // Co_op / pre_emp  
{  
    tByte Index = 0;  
    // 首先在队列中找到一个空隙(如果有的话)  
    while ((hSCH_tasks_G[Index].pTask != 0) && (Index < hSCH_MAX_TASKS))  
    {  
        Index++;  
    }  
    // 是否已经到达队列的结尾?  
    if (Index == hSCH_MAX_TASKS)  
    {  
        // 任务队列已满  
        // 设置全局错误变量  
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;  
        // 同时返回错误代码  
        return hSCH_MAX_TASKS;  
    }  
  
    // 如果能运行到这里，说明任务队列中有空间
```

```

hSCH_tasks_G[Index].pTask = Fn_p;
hSCH_tasks_G[Index].Delay = Dcl;
hSCH_tasks_G[Index].Period = Per;
hSCH_tasks_G[Index].Co_op = Co_op;
hSCH_tasks_G[Index].RunMe = 0;
return Index; // 返回任务的位置(以便以后删除)
}

/*-----*
hSCH_Delete_Task()
//从调度器删除任务。注意：并不从存储器中删除相关的函数。仅仅是不再由调度器调用这个任务。
参数：TASK_INDEX-任务索引。由hSCH_Add_Task()提供。
返回：RETURN_ERROR或RETURN_NORMAL
-----*/
bit hSCH_Delete_Task(tByte Task_index)
{
    bit Return_code;
    if (hSCH_tasks_G[Task_index].pTask == 0)
    {
        // 这里没有任务
        //
        // 设置全局错误变量
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;
        // ...同时返回错误代码
        Return_code = RETURN_ERROR;
    }
    else
    {
        Return_code = RETURN_NORMAL;
    }
    hSCH_tasks_G[Task_index].pTask = 0;
    hSCH_tasks_G[Task_index].Delay = 0;
    hSCH_tasks_G[Task_index].Period = 0;
    hSCH_tasks_G[Task_index].RunMe = 0;
    return Return_code; // 返回状态
}

/*-----*
hSCH_Report_Status()
用来显示错误代码的简单的函数。
这个版本在连接到端口的LED上显示错误代码。如果需要的话，可以修改为通过串行连接等方式来报告错误。
错误只在有限的时间内显示(1ms时标间隔时，60000时标=1分钟)。此后错误代码被复位为0。这些代码可以很容易的修改为“永远”显示最近的错误。这对于系统可能更为合理。
更加详尽的资料参见第14章。
-----*/
void hSCH_Report_Status(void)

```

```

{
#endif SCH_REPORT_ERRORS
// 只在需要报告错误时适用
// 检查新的错误代码
if (Error_code_G != Last_error_code_G)
{
    // 假定 LED 采用负逻辑
    Error_port = 255 - Error_code_G;
    Last_error_code_G = Error_code_G;
    if (Error_code_G != 0)
    {
        Error_tick_count_G = 60000;
    }
    else
    {
        Error_tick_count_G = 0;
    }
}
else
{
    if (Error_tick_count_G != 0)
    {
        if (--Error_tick_count_G == 0)
        {
            Error_code_G = 0; // 复位错误代码
        }
    }
}
#endif
}
/*-----*/
hsCH_Go_To_Sleep()

本调度器在时钟时标之间将进入“空闲模式”来节省功耗。
下一个时钟时标将使处理器返回到正常操作状态。
注意，如果这个函数由宏来实现，或简单地将这里的代码粘贴到“调度”函数中，
可以有少量的性能改善。
然而，通过采用函数调用的方式来实现，可以在开发期间更容易地使用 Keil
硬件模拟器中的“性能分析器”来估计调度器的性能。这方面的例子参见第 14 章。
***如果使用看门狗的话，可能需要禁止这个功能***
***根据硬件的需要修改***
/*-----*/
void hsCH_Go_To_Sleep()
{
    PCON |= 0x01; // 进入空闲模式（通用 8051 版本）
}

```

```

// 在 80c515/80c505 上，进入空闲模式需要两个连续的指令，用来避免意外的触发
// PCON |= 0x01; // 进入空闲模式 (#1)
// PCON |= 0x20; // 进入空闲模式 (#2)
}

/*
-----文件结束-----
*/

```

源程序清单 17.10 混合式调度器的例子的一部分

```

/*
-----*
LED_Hyb.C (v1.00)

-----*
-详细资料参见 LED_Hyb.C
-----*
// -----公用的函数原型
void LED_Short_Init(void);
void LED_Short_Update(void);
void LED_Long_Update(void);
/*
-----*
-----文件结束-----
-----*

```

源程序清单 17.11 混合式调度器的例子的一部分

```

/*
-----*
LED_Hyb.C (v1.0)

-----*
用于混合式调度器的简单的“闪烁 LED”的测试函数
***具有锁定机制***

/*
-----*
#include "Main.h"
#include "Port.H"
#include "LED_Hyb.h"
#include "Delay_T0.h"
// -----私有的常数-----
// 用于锁定机制
#define LOCKED 1
#define UNLOCKED 0
// -----私有变量定义-----
static bit LED_short_state_G;
// 锁定标志
static bit LED_lock_G = UNLOCKED;
/*
-----*
LED_Flash_Init()
-准备闪烁 LED
-----*

```

第17章 混合式调度器



阅读 例题

阅读 例题

```
if (LED_lock_G == LOCKED)
{
    return;
}
// 端口空闲——锁定它
LED_lock_G = LOCKED;
for (i = 0; i < 5; i++)
{
    LED_long_port = 0x0F;
    Hardware_Delay_T0(1000);
    LED_long_port = 0xF0;
    Hardware_Delay_T0(1000);
}
// 放开该端口
LED_lock_G = UNLOCKED;
}
/*-----文件结束-----*/

```

源程序清单 17.12 混合式调度器的例子的一部分

例子：更进一步的混合式调度器

在语音处理应用程序中使用混合式调度器的例子参见第 34 章的内容。

进阶阅读

有关该领域进阶阅读的更多建议请参阅第 13 章中的“进阶阅读”一节。

Part 4

用户界面

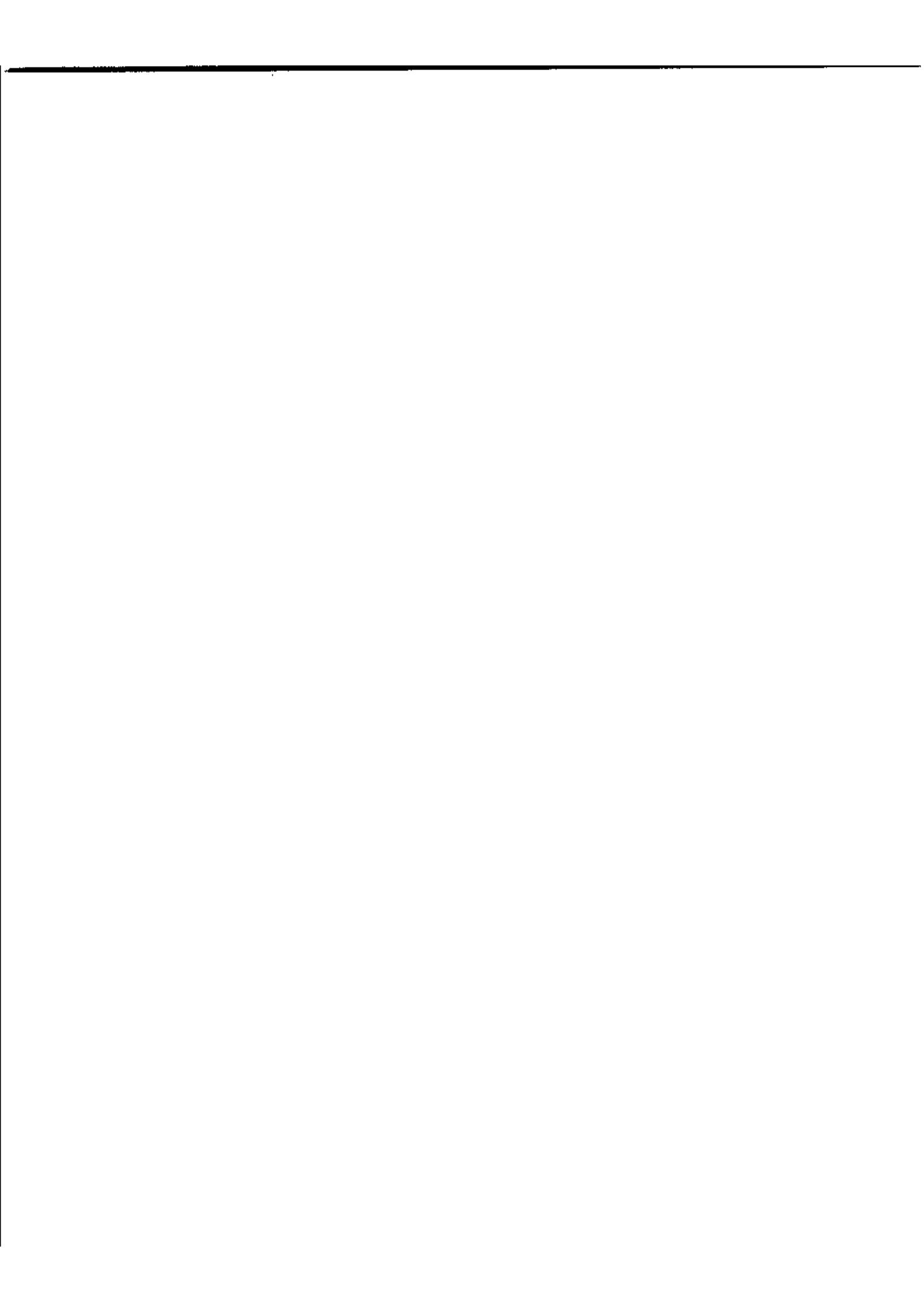
本篇将讨论一些模式，为各种嵌入式系统创建用户界面提供帮助。

第 18 章将介绍 PC 连接 (RS-232)。这个模式讨论了如何使用通用的“RS-232”标准在嵌入式芯片和桌面、笔记本或类似的 PC 之间传递数据。该模式除了本身是一个很有用的模式之外，还说明了用于合作式调度平台的软件开发所要求的许多特性。

第 19 章将探讨从开关读取输入的方法。这里将既考虑纯软件的接口又考虑基于硬件的接口。在第 20 章中，将了解如何把同样的基本技术有效地扩展到键盘操作。

在第 21 章中，将把注意力转向创建 LED 显示。尤其是考虑多路复用 LED 接口，在大多数系统中这是惟一高性价比的解决方案。

最后，在第 22 章中，将讨论如何控制 LCD (文本) 显示。在该章中将具体讨论以 Hitachi 的 HD44780 LCD 控制器芯片为基础的显示操作。



Chapter 18

通过 RS-232 与 PC 通信

引言

本章将介绍PC连接（RS-232）。这个模式描述了如何使用通用的“RS-232”标准在嵌入式芯片和桌面、笔记本或类似的PC之间传递数据。

注意，这里不涉及两个（或更多）微控制器之间的数据传送过程，它们将在第6篇中讨论。

PC 连接（RS-232）

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 正在创建该系统的用户界面。

问题

如何使用“RS-232”将嵌入式系统连接到桌面（或笔记本或手持式）PC？

背景知识

这个模式涉及使用通常被称为 RS-232 的通信协议，在 8051 微控制器和某种形式的个人计算机（PC、笔记本或类似设备）之间传送数据。

本节提供了一些有关 RS-232 的重要的背景资料。然而，首先要了解用于通信领域的一些重要的术语。

什么是“单工”、“半双工”和“全双工”串行通信？

在一个单工的串行通信系统中，一般要求至少有两根线（“信号线”和“地线”）。数据传送

只有一个方向。例如，可以使用单工数据传送将数据从一个简单的数据监测系统传送到 PC 上。

在半双工串行通信系统中，一般同样要求至少有两根线。这里的数据传送是双向的。然而，同一时刻传输只能为一个方向。在上面的数据监测的例子中做一些变化，可以使用半双工通信机制发送信息到嵌入式模块（来设置参数，比如采样速度）。此外，在其他时候，可以使用这个连接将嵌入式装置上的数据下载到 PC 中。

在一个全双工的串行通信系统中，一般要求至少有三根线（信号线 A、信号线 B 和地线）。信号线 A 将传送一个方向上的数据，同时信号线 B 传送另一个方向上的数据。

什么是“RS-232”？

1997 年，电信工业协会发布了被正式地称为 TIA-232 版本 F 的串行通信协议。自从这个协议于 19 世纪 60 年代作为“推荐标准”出现以来，已经被普遍地称为“RS-232”。类似的基准（V.28）由国际电信联盟（ITU）和国际电报电话咨询委员会（CCITT）发布。

“RS-232”标准包括以下详细内容：

- 用于数据传输的协议。
- 信号线上使用的电压。
- 用于连接设备的接插件。

总的说来，该标准易于理解并被广泛使用，其数据传输速率最高为大约 115 或 330kB/s（115/330k 波特率），能够在 15 米或更远的距离内传送数据。

注意，RS-232 是一种点到点的通信标准。与多点的 RS-485 标准（将在第 27 章讨论）不同，RS-232 只用来连接两个芯片。

基本的 RS-232 协议

RS-232 是一种面向字符的协议。即，规定为发送一个一个的 8 位数据块。为了在 RS-232 连接上传送一个字节，通常按如下方式对信息进行编码：

- 发送一个“起始”位。
- 发送数据（8 位）。
- 发送一个（或多个）“停止”位。

下面将分别讨论这些阶段中的每一个。

静态

当 RS-232 上的“发送”线上无数据发送时，这根线将保持为逻辑 1 电平。

起始位

将“发送”线拉低以指示数据传输的起始。

数据

数据往往被编码为 7 位格式的 ASCII 码。首先发送最低位。如果发送 7 位数据，则第八个数据位往往用作简单的奇偶校验位并被传送，用来提供一种基本的逐个字符的错误检测功能。

注意，这里介绍的代码都不使用奇偶校验位，使用所有 8 位来传送数据。

停止位

停止位输出逻辑 1。可以是 1 个、或者较少见的 1.5 或 2 个脉冲宽度。

注意，将在所有的代码例子中使用一个停止位。

异步数据传输和波特率

RS-232 使用异步规约。这指的是时钟信号不随数据发送。通信链路的两端有运行在同样频率的内部时钟。如果必要的话，数据（就 RS-232 而言，为“起始”位）用于同步该时钟，以保证成功地传送数据。

RS-232 通常运行在一系列（受限制的）波特率上。典型的波特率为：75、110、300、1200、2400、4800、9600、14 400、19 200、28 800、33 600、56 000、115 000 和（很少）330 000 波特率。其中，9600 波特率是一种非常“安全的”选择，得到了非常广泛的支持。

RS-232 电平

接收器的阈值电平为 +3V 和 -3V，而信号线使用反逻辑，允许的电压极限是 +/-15V。

注意，不能直接从微控制器的端口引脚上得到这些电压，需要某种形式的接口硬件。例如，Maxim 的 Max232 和 Max233 是广为使用的长线驱动器芯片。将在“解决方案”中讨论如何使用这样的芯片。

流量控制

RS-232 往往使用某种形式的流量控制。这是一种通过软件或硬件实现的协议，使数据的接收器可以通知发送器暂停数据流。这很有必要，例如，如果向 PC 发送数据，而 PC 的 RAM 缓冲区已满。于是 PC 将通知嵌入式系统暂停数据传送，直到缓冲区内的数据被存储到磁盘上。

可以使用硬件握手，然而这需要额外的信号线。最常见的流量控制方法是“Xon/Xoff 控制”，这需要半双工或全双工通信连接，运行如下：

1. 发送器发送一个字节的数据。
2. 如果接收器能够接收更多的数据，它将不做控制。
3. 发送器发送另一个字节的数据。
4. 继续步骤 1~3，直到接收器不能接受更多的数据。然后它将发送一个“Control s”(Xoff) 字符到发送器。
5. 发送器接收“Xoff”命令并暂停数据传输。
6. 当接收器节点可以接收更多的数据时，它将发送一个“Control q”(Xon) 字符到发送器。
7. 发送器将继续数据传输。
8. 继续步骤 1 的过程。

解决方案

本节将讨论如何实现从嵌入式 8051 微控制器到 PC 的 RS-232 连接。

收发器硬件

正如在“背景知识”中提到的，RS-232 通信使用的电压与微控制器使用的电压不兼容。因此，在微控制器电路板和 PC 电缆之间将需要某种形式的电平转换电路。

通常，性价比最高的实现方法是使用一个专用的“收发器”芯片。其中，Max232 (Maxim) 常常被使用。然而，在比较新的设计中，Max233 是一个更好的选择，因为它需要更少的外部模块。

用于最新设计的 Max233 所需的连接如图 18.1 所示，同时显示了(9 针 D 型插座“DB-9”)的标准连接。

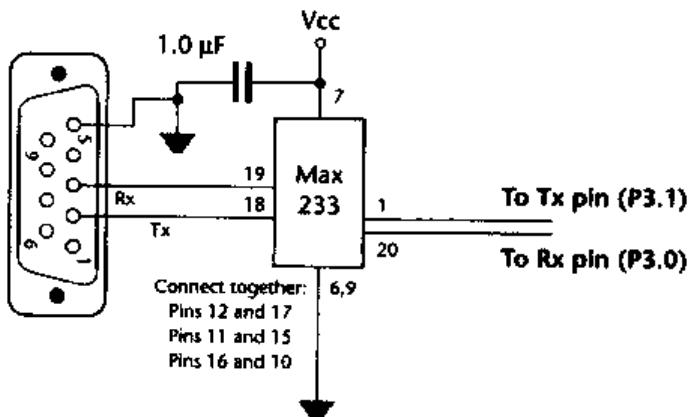


图 18.1 使用 Max233 作为 RS-232 收发器

电缆连接

要使用这里介绍的软件连接 PC，需要一根 3 芯电缆。

在 PC 端使用 DB-9 插座（“孔”），在微控制器端使用 DB-9 插头（“针”）。

电缆连接要求“直连”，如表 18.1 所示。

表 18.1 用来连接微控制器和台式 PC (或类似终端) 所需的“直连”电缆

PC (COM1, COM2) DB-9 接插件		微控制器 DB-9 接插件
RxD-Pin2	到	TxD-Pin2
TxD-Pin3	到	RxD-Pin3
Ground-Pin5	到	Ground-Pin5

软件结构

假设希望以标准的 9600 波特率向 PC 传送数据。即，9600 比特/秒。正如在“背景知识”中讨论的，发送每个字节的数据，加上停止和起始位，共有十位信息需传输（假定使用一个停止位）。因此，发送每个字节需要大约 1ms。

这个问题对所有系统都很重要，特别是那些使用调度器的系统。例如，希望向 PC 发送以下信息：

Current core temperature is 36.678 degrees

那么完成发送这 42 个字符的任务将需要超过 40ms，这么长的时间常常是不可接受的。

解决这个问题的最显而易见的方法是增加波特率。然而，这并不总是可行的，而且它不能解决根本上的问题。

较好的解决方案是将所有数据存入微控制器的缓冲区中。使用一个被定期调度的任务将缓冲区内的数据发送到 PC，通常一次发送一个字节。

下面给出的代码库将使用这种解决方案，这些代码包含在配套的 CD 上。这种结构是多级任务的一个很好的例子，常常用在用户接口库。例如，LCD 字符面板就使用了非常类似的数据结构。

使用片内 U (S) ART 进行 RS-232 通信

决定了 RS-232 库的基本结构之后，需要更详细地讨论如何使用片内串行端口。

这个端口是全双工的，因此能够同时发送与接收。此外它的接收被缓冲，因此能够在接收寄存器中的前一个收到的字节被读走之前开始接收第二个字节（然而，如果接收第二个字节完成时，第一个字节仍然没有被读走，将丢失一个字节）。

串行端口能够以四种方式运行（一个同步方式，三个异步方式）。在这个模式中，主要对方式 1 感兴趣。在这个方式中，发送（通过 TxD）或接收（通过 RxD）10 位数据[一个起始位 (0)，8 个数据位（最低位在前）和一个停止位 (1)]。

注意：当发送或接收完一个字节时，串行接口可以产生中断请求。然而，因为在第 1 章中讨论的理由，这个模式中使用的所有代码都不产生中断。

串行端口寄存器

串行端口控制和状态寄存器是特殊功能寄存器，SCON。这个寄存器包含方式选择位（以及串行端口中断位、TI 和 RI）。

SBUF 是串行接口的接收和发送缓冲器。向 SBUF 写时，将加载发送寄存器并开始发送。从 SBUF 读时，将访问一个在物理上独立的接收寄存器。

产生波特率

有几种不同的方式可以产生用于串行端口的波特率，这取决于串行端口的运行方式。

如上所述，这里主要涉及工作在方式 1 下的串行端口。在这个方式下，波特率由定时器 1 或定时器 2 的溢出频率决定。因为在第 14 章中讨论的理由，假定如果有定时器 2，通常将用它来驱动调度器。因此，集中考虑使用定时器 1 来产生波特率。

波特率取决于定时器 1 的溢出频率以及 SMOD 的值，如下所示：

$$\text{Baud rate (Mode1)} = \frac{2^{\text{SMOD}} \times \text{Frequency}_{\text{oscillator}}}{32 \times \text{Instructions}_{\text{cycle}} \times (256 - TH1)}$$

这里：

SMOD 是 PCON 寄存器中的“波特率加倍”位

Frequency_{oscillator} 是振荡器/谐振器频率

Instructions_{cycle} 每个机器指令的振荡周期数（例如 12 或 6）

TH1 是定时器 1 的重装值

注意：定时器 1 使用 8 位自动重装模式，并且应该禁止产生中断。

需要特别重视的是，通常不可能使用定时器 1 生成标准波特率（例如 9600），除非使用 11.0592MHz 的晶体振荡器。

要了解为什么是这样，假定 *SMOD=0* (*SMOD=1* 时同样)，每个指令 12 个周期，而需要 9600 的波特率。等式现在变成：

$$9600 = \frac{11059200}{32 \times 12 \times (256 - TH1)}$$

这可以变成：

$$\frac{11059200}{9600 \times 384} = 256 - TH1$$

或

$$256 - TH1 = 3$$

这样，如果设置 *TH1* 为 253 (0xFD)，则将得到精确的 9600 波特率。如果使用 12MHz 的振荡器重复这些计算，将得到：

$$\frac{12000000}{9600 \times 384} = 256 - TH1$$

或

$$256 - TH1 = 3.25520833333$$

这样，*TH1* 要求的值为 252.7447916667。

最接近的整数是 253，这意味着实际的波特率大约是 10 417 波特率，比要求的 9600 波特率超出 8% 以上。

记住，这是一种异步规约，其正常运行取决于连接的两端运行同样的波特率。实际上，连接两端的波特率的差异在 5% 以内时就可以工作，但不能再多。

虽然可以允许有 5% 的裕度，但是使波特率尽可能地接近标准值总是个好方法。因为在实际环境中，PC 的振荡器和嵌入式系统的振荡器之间可能有显著的温度差异。这将导致 PC 和微控制器上的波特率的“漂移”，即使它们在开始时是完全相同的。因此，如果在开始时波特率就不匹配，那么通信可能在正常使用期间彻底失败。这类“莫名其妙的故障”导致许多开发人员度过不眠之夜（“我想不明白！在实验室里的所有测试中它都是正常工作的！”）。此外还应注意，当使用异步串行连接（诸如 RS-232、RS-485 或 CAN）时，必须使用晶体振荡器（而不是陶瓷谐振器）。陶瓷谐振器不够稳定，不适用于这种场合。

详细资料参见晶体振荡器和陶瓷谐振器。

处理11.0592MHz的问题

在调度应用中，使用11.0592MHz的晶体振荡器是不理想的。11.0592MHz可以产生精确波特率，然而它不是调度器中的理想时标源。特别是，由11.0592MHz晶振驱动的8051芯片不能生成精确的1ms时标。

这个问题有至少四种解决方案：

- 第一种解决方案是使用11.0592MHz晶振，产生5ms的时标间隔。注意，这个晶体频率能够精确地产生5ms间隔，正如CD上的例子所表明的。
- 第二种解决方案是使用定时器2来产生波特率。与定时器1相比较，定时器2可以在更宽的振荡器频率范围内产生精确波特率。毫无疑问，在单处理器系统中，这意味着定时器2不能同时用来产生调度器时标，而需要使用定时器0或定时器1来产生。
- 第三种解决方案是使用最新的8051芯片上的专用波特率发生器。这将产生非常标准的波特率（即使是使用12MHz的晶体）。此外，使用专用波特率发生器可以释放一个定时器使其用作他用。同样，CD上有一个库使用内部波特率发生器。
- 第四种解决方案是使用两个微控制器和一个共享时钟调度器，图18.2说明了这种方法。

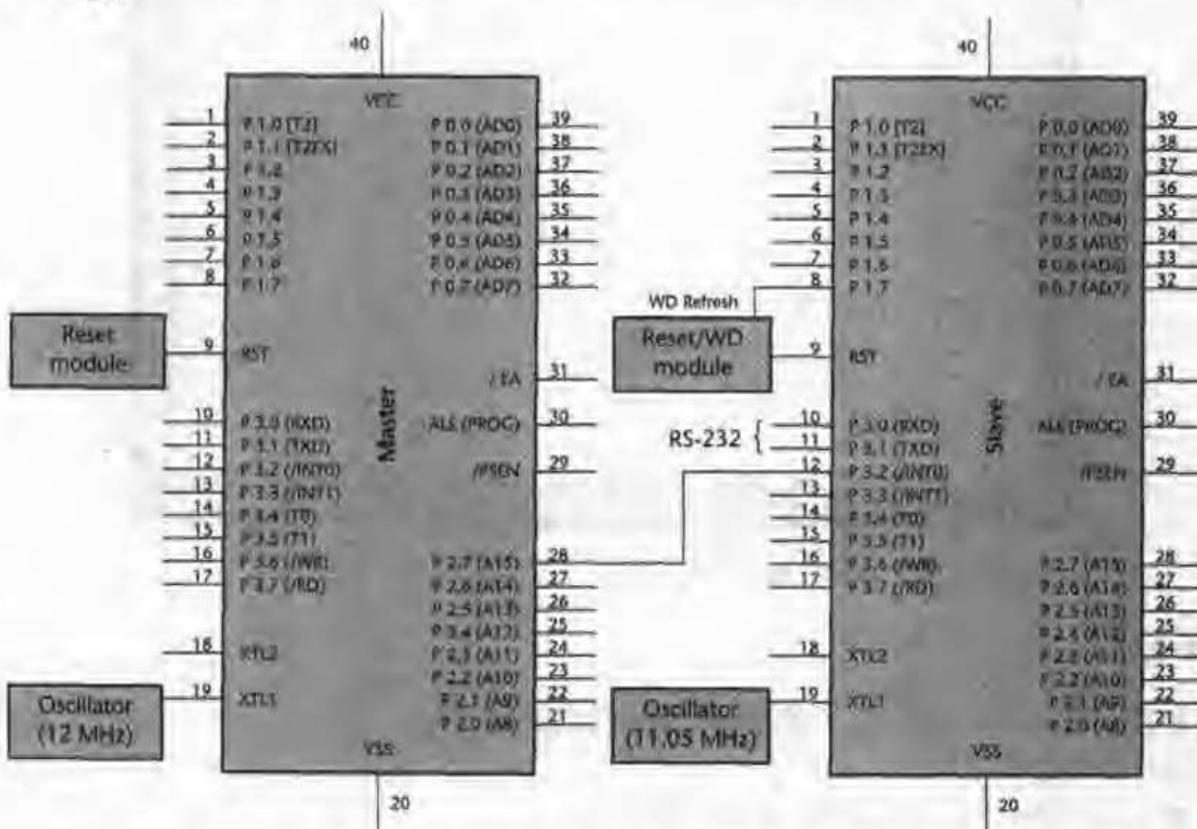


图18.2 使用两个处理器和一个共享时钟（中断）调度器来提供精确的1ms时标间隔（通过主机上的12MHz晶振）和精确的波特率（通过从机上的11.0592MHz晶振）

PC 软件

在这个模式中主要关注微控制器上的软件以便向 PC 传送数据。显而易见，这也需要 PC 上的相应软件。

如果台式机运行 Windows (95、98、NT 或 2000)，那么一种简单而有效的方案是使用所有这些操作系统都包含的“超级终端”应用程序。图 18.3 显示了正在运行一个应用例子的“超级终端”应用程序。

虽然超级终端（或其他平台上的类似终端仿真程序）很有用，但它们并不适用于所有场合。

如果需要专用的 PC 代码用于存储或分析嵌入式系统的数据，则需要自己来写。编写这样的程序超出了本书的范围，有关这方面的详细资料和有用的代码例子可以参考 Axelson (1998)。

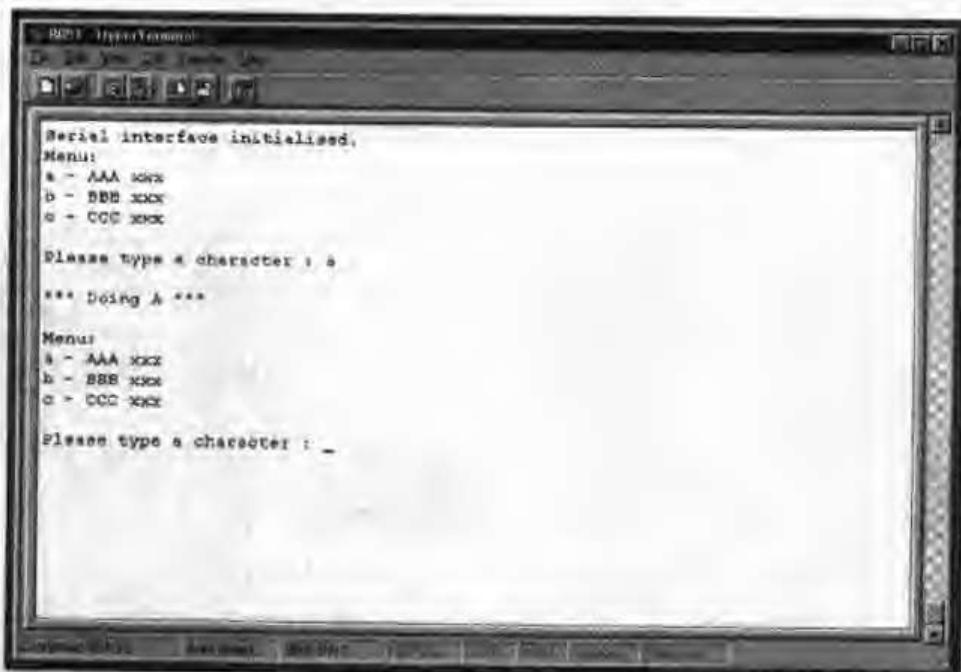


图 18.3 使用“超级终端”显示从嵌入式微控制器发送来的信息

硬件资源

使用本模式中描述的 PC 连接库有下列硬件资源问题：

- 需要使用 UART。
- 需要使用两个端口引脚 (P3.0 和 P3.1)。
- 需要使用一个通用定时器或一个专用的波特率发生器电路。

但总体说来，存储器开销是最大的问题，尤其是在只有内部存储器的场合。正如在“解决方案”中讨论的，这个问题的根源在于：所使用的（向 PC 发送数据的）结构基于缓冲区，用

户根据需要向缓冲区中写入数据。然后这个缓冲区将由一个被调度的任务^①每次传送一个字符逐渐清空。

缓冲区本身非常简单：

```
// 发送缓冲器的长度  
#define PC_LINK_TRAN_BUFFER_LENGTH 100  
static tByte Tran_buffer[PC_LINK_TRAN_BUFFER_LENGTH];
```

如果内存不够用，则有几种选择：

- 可以减小缓冲区的大小。这意味着应用中的任务必须将数据分解为更小的数据块写入。一般情况下，这带来的主要问题是必须使用更短的字符串。
- 可以提高波特率，并且修改代码使得每次“刷新”函数运行时，调度器发送一个字节以上的数据。
- 如果只使用片内存储器，则可以选择有额外的片内 RAM 的 8051 芯片。例如：Dallas 和 Infineon 生产了许多这样的芯片。在片外数据存储器中讨论了如何使用这种存储器。

可靠性和安全性

本节中将讨论 RS-232 通信的可靠性和安全性问题。

关于 printf()

通常不推荐使用标准库函数，比如 printf()。因为：

- 这个函数立即向 UART 发送数据。因此，发送的运行时间通常太长，在合作式调度应用中处理将不太安全。
- 大多数 printf() 的实现不包括超时，使用这个函数时如果出现错误，很可能将使整个应用“挂起”。

如果决定使用 printf()，请参考第 10 章中使用这个函数的简单的库。

总体评论

RS-232 协议没有提供充分的错误检查手段，需要人工实现它。一个简单的方法是重要的数据发送两遍（或更多），并且比较这两个（或更多）版本。如果有不符合的地方，则再次发送数据。显而易见，这种方法极大地增加了对通信带宽的要求。

可移植性

UART 属于 8051 内核的一部分。因此，如果系统有一个基于 UART 的 RS-232 接口，通常它将是可移植的。然而，使用非内核特性（比如专用的波特率发生器或一个以上的串行端口）

^① 注意：“发送”和“接收”通道都有相应的缓冲区，通常发送缓冲器比接收缓冲器大。因为在大多数情况下数据流的方向是从微控制器到 PC。因此，这里主要关注发送缓冲器。然而，类似的讨论和解决方案也适用于那些信息流的主要方向是从 PC 到微控制器的情况。

将显著地降低可移植性。

优缺点小结

- ◎ 对 RS-232 的支持是 8051 内核的一部分，基于 RS-232 的系统的可移植性很好。
- ◎ 在 PC 端，RS-232 无所不在，每个 PC 都有一个或多个 RS-232 端口。
- ◎ 使用新的收发器芯片，连接长度可以达到 30 米（100 英尺）。
- ◎ 因为有硬件支持，通常 RS-232 的软件开销很低。
- ◎ RS-232 是一种点对点协议（不同于 RS-485，参见第 27 章的内容）。同时只能将一个微控制器直接连接到 PC 上。
- ◎ RS-232 几乎没有硬件级错误检查（不同于 CAN，参见第 28 章的内容）。如果希望确定 PC 端接收的数据是正确的，需要在软件中执行检查。

相关的模式和替代解决方案

有几种相关模式和一些替代方案将在这里讨论。

选择振荡器或谐振器

如上所述，当使用异步串行连接时，通常必须使用晶体振荡器（而不是陶瓷谐振器）。陶瓷谐振器不够稳定，不适用于这种场合。详细资料参见晶体振荡器和陶瓷谐振器。

多点通信

将多个微控制器与一台 PC 连接在一起很容易实现。使用共享时钟调度器 [参见 SCI 调度器（数据）和 SCC 调度器] 将控制器连接在一起，使用其中的一个芯片上的串行端口通过 RS-232 连接向 PC 发送数据。如果需要的话，可以以这种方法向多台 PC 发送数据。

USB

自从 20 世纪 60 年代以来，RS-232 就被广为使用了。它是通用而有效的，然而也有其局限性。逐渐的，越来越多的 PC 附带有 USB（通用串行总线）端口。

现在可以使用 USB 在 PC 和 8051 微控制器之间传送数据。例如，Infineon 的 C541 是一种支持片内 USB 的 8051 芯片。注意，该芯片不仅仅包括 USB 控制器，还包含 USB 驱动器。不同于 RS-232，它不需要外部收发器硬件（如图 18.4 所示）。

其他 USB 兼容的 8051 芯片包括 Cypress Semiconductor^② 的 EZ-USB 系列和 Texas Instruments^③ 的 TUSB3200。

注意，对于那些不支持 USB 的 8051 微控制器，可以考虑的一种备选方案是 FTDI^④ 的 FT8U232AM USB UART 芯片。对于 8051 而言，USB UART 就像是一种普通的 RS-232 收发

^② www.cypress.com

^③ www.ti.com

^④ www.ftdi.co.uk

是。

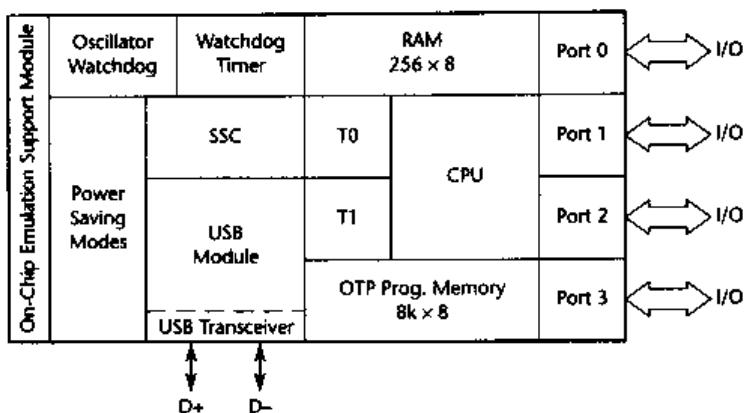


图 18.4 具有片内 USB 支持的 Infineon 的 C541 微控制器的结构 (转载自 Infineon)

例子：用于（通用）8051/8052 的 PC 连接库

这个例子说明了如何将标准 8051 芯片连接到 PC 上(源程序清单 18.1~源程序清单 18.5)。在这个例子中，软件是由菜单控制的。用户可以通过 PC 上的终端仿真程序选择在微控制器上运行三种不同的任务 (参见图 18.5)。

在这个库中，调度器由定时器 2 驱动，而定时器 1 用于产生波特率。

像往常一样，这个例子中的所有文件都包含在 CD 上与本章相关的目录中。

```
/*
 *-----*
 * Port.H (v1.01)
 *-----*
 * 项目 IO_T2_T1 的“端口头文件”(参见第 10 章)
 // ----- Sch51.C -----
 // 如果不需要错误报告，将这一行注释掉
 #define SCH_REPORT_ERRORS
 #ifdef SCH_REPORT_ERRORS
 // 将用来显示错误代码的端口
 // 只在报告错误时使用
 #define Error_port P1
 #endif
 // ----- Lnk_IO.C -----
 // 引脚 3.0 和 3.1 用于 RS-232 标准接口
 /*-----*
 *-----文件结束-----
 *-----*/
```

源程序清单 18.1 菜单控制的通用 PC 连接库的一部分

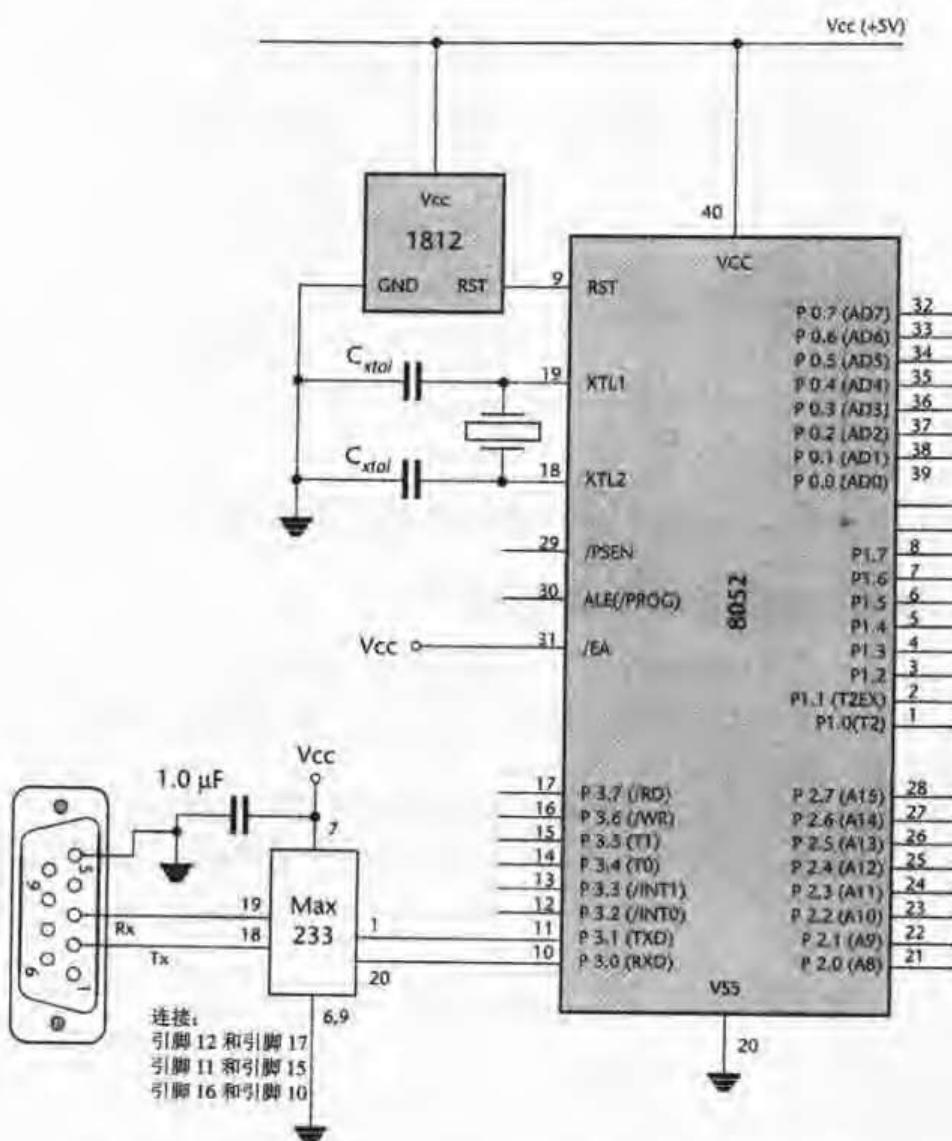


图 18.5 将 8051 微控制器与 Max233 (RS-232) 收发器连接

```

/*
Main.c (v1.00)

用于 PC 连接 (RS-232) 模式的演示程序
链接程序选项:
OVERLAY (main ~ (MENU_Command_Processor),
SCH_Dispatch_Tasks ! (MENU_Command_Processor))
*/
#include "Main.h"
#include "2_05_11g.h"
#include "lnk_io_A.h"
#include "Menu_A.h"
/*
*/

```

```
/*
void main(void)
{
    // 设置调度器
    SCH_Init_T2();
    // 波特率设置为 9600, 通用 8051 版本
    PC_LINK_Init_T1(9600);
    // 必须调度这个任务 (10x~100x 每秒)
    //
    // 定时单位为时标 (5ms 间隔), 而不是毫秒
    SCH_Add_Task(MENU_Command_Processor, 10, 2);
    SCH_Start();
    while(1)
    {
        // 在 P4 上显示错误代码 (参见 Sch51.C)
        SCH_Dispatch_Tasks();
    }
}
*/
-----文件结束-----
*/
```

源程序清单 18.2 菜单控制的通用 PC 连接库的一部分

```
/*
Menu_A.C (v1.00)

通过串行 (UART) 连接与桌面/笔记本 PC 交互操作 (例如数据下载) 的简单的框架
使用 (Windows95、98、2000 中的) “超级终端”或其他操作系统上的类似的终端仿真程序
终端选项:
- 数据位=8
- 奇偶校验=无
- 停止位=1
- 流量控制=Xon/Xoff
*/
#include "Main.h"
#include "2_05_11g.h"
#include "Menu_A.h"
#include "lnk_IO_A.h"
/*
MENU_Command_Processor()
这个函数是主菜单的“命令处理”函数
每隔 (大约) 10ms 调度一次
*/
void MENU_Command_Processor(void)
{
    static bit First_time_only;
    char Ch;
    if (First_time_only == 0)
    {
```

```
First_time_only = 1;
MENU_Show_Menu();
}

// 检查用户输入
PC_LINK_Update();
Ch = PC_LINK_Get_Char_From_Buffer();

if (Ch != PC_LINK_IO_NO_CHAR)
{
    MENU_Perform_Task(Ch);
    MENU_Show_Menu();
}
}

/*-----*
MENU_Show_Menu()
通过串行连接在 PC 屏幕上显示菜单选项
根据你的应用需要来修改
-----*/
void MENU_Show_Menu(void)
{
    PC_LINK_Write_String_To_Buffer("Menu:\n");
    PC_LINK_Write_String_To_Buffer("a - AA x\n");
    PC_LINK_Write_String_To_Buffer("b - BB x\n");
    PC_LINK_Write_String_To_Buffer("c - CC x\n\n");
    PC_LINK_Write_String_To_Buffer("Please type a character : ");
}

/*-----*
MENU_Perform_Task()
执行所需的用户任务根据你的应用需要来修改
-----*/
void MENU_Perform_Task(char c)
{
    // 回显菜单选项
    PC_LINK_Write_Char_To_Buffer(c);
    PC_LINK_Write_Char_To_Buffer('\n');
    // 执行任务,
    switch (c)
    {
        case 'a':
        case 'A':
            {
                Function_A();
                break;
            }
        case 'b':
        case 'B':
            {
                Function_B();
                break;
            }
    }
}
```

```

        }
    case 'c':
    case 'C':
        {
            Function_C();
        }
    }
/*-----*
 Placeholder function
-*-----*/
void Function_A(void)
{
    PC_LINK_Write_String_To_Buffer("\n*** Doing A ***\n\n");
P1 = 'A';
}
/*-----*
 Placeholder function
-*-----*/
void Function_B(void)
{
    PC_LINK_Write_String_To_Buffer("\n*** Doing B ***\n\n");
P1 = 'B';
}
/*-----*
 Placeholder function
-*-----*/
void Function_C(void)
{
    PC_LINK_IO_Write_String_To_Buffer("\n*** Doing C ***\n\n");
P1 = 'C';
}
/*-----*
 ---文件结束---
-*-----*/

```

源程序清单18.3 (菜单控制的)通用PC连接库的一部分

```

/*-----*
Lnk_IO_A.C(v1.00)
-----*
// 简单的PC连接库
使用USART和引脚3.1(Tx)和3.0(Rx)。详细资料参见上下文
-*-----*/
#include "Main.h"
#include "Lnk_IO_A.h"
//-----公有变量声明-----
extern tByte In_read_index_G;
extern tByte In_waiting_index_G;
extern tByte Out_written_index_G;

```

```

extern tByte Out_waiting_index_G;
/*-----*/
PC_LINK_Init_T1()
This(generic)version uses T1 for baud rate generation
/*-----*/
void PC_LINK_Init_T1(const tWord BAUD_RATE)
//这个(通用)版本使用T1来产生波特率。
{
    PCON &=0x7F; // 设置SHOD位为0(波特率不加倍)
    //使能接收。
    //8位数据、1个起始位和1个停止位
    //波特率可变(异步)
    //只有当收到一个有效的停止位时接收标志才置为1
    //TI位置为1(发送缓冲器空)
    SCON = 0x72;
    TMOD|=0x20;// T1为模式2、8位自动重装
    TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100 *3125)
        /((tLong)BAUD_RATE * OSC_PER_INST * 1000)));
    TL1 = TH1;
    TR1 = 1;//启动定时器
    TI = 1;//发送第一个字符(空)
    //设置用于读/写的缓冲区
    In_read_index_G = 0;
    In_waiting_index_G = 0
    Out_written_index_G = 0
    Out_waiting_index_G = 0
    PC_LINK_Write_String_To_Buffer("Serial interface initialized.\n");
    //不使能中断
    ES = 0;
}
/*-----文件结束-----*/
/*-----*/

```

源程序清单 18.4 菜单控制的通用 PC 连接库的一部分

```

/*-----*/
Lnk_IO.C(V1.00)
-----用于8051系列的简单的PC连接库的内核文件
使用USART,引脚3.1(Tx)和3.0(Rx)。详细资料参见上下文
/*-----*/
#include "Main.h"
#include "Lnk_io.h"
// -----Public variable definition -----
//公有变量定义
tByte In_read_index_G;// 已经读取的数据输入缓冲区
tByte In_waiting_index_G;// 还没有读取的数据输入缓冲区
tByte out_written_index_G;// 已经写入的数据输入缓冲区
tByte out_waiting_index_G;// 还没有写入的数据输入缓冲区

```

第18章 通过RS-232与PC通信

```
-----公有变量声明-----
// 错误代码变量
//
// 将用来显示错误代码的端口以及错误代码的详细资料参见 Main.H
extern tByte Error_code_G;
-----私有函数原型-----
void PC_LINK_Send_Char(const char);
-----私有的常数-----
// 接收缓冲区的长度
#define PC_LINK_RECV_BUFFER_LENGTH 8
// 发送缓冲区的长度
#define PC_LINK_RECV_BUFFER_LENGTH 100
// available in buffer
#define PC_LINK_NO_CHAR 127
#define XON 0x11
#define XOFF 0x13
-----私有变量-----
static tByte Recv_buffer[PC_LINK_RECV_BUFFER_LENGTH];
static tByte Tran_buffer[PC_LINK_RECV_BUFFER_LENGTH];
//PC_LINK_Get_Char 的返回值, 如果缓冲区中没有字符
PC_LINK_Update()
/*
// 在UART(硬件)接收缓冲区中检查字符发送软件发送缓冲区中的下一个字符
*/
void PC_LINK_Update(void)
{
// 这里处理发送字节
// 是否有数据准备发送?
if(Out_written_index_G< Out_written_index_G)
{
    PC_LINK_Send_Char(Tran_buffer[Out_written_index_G]);
    Out_written_index_G++;
}
else
{
// 没有数据需要发送-仅仅复位缓冲区指针
Out_waiting_index_G = 0;
Out_written_index_G = 0
}
// 这里只处理接收字节
// ->检查 RI 标志
if (RI == 1)
{
// 只有当收到一个有效的停止位时接收标志才置为 1
// -> 已有数据需要被读取到接收缓冲区中
// 如果旧的数据已经读取, 希望读取指针 0
// (简单的循环缓冲区)
if (In_waiting_index_G == In_read_index_G)
```

```

    {
    In_waiting_index _G] = 0;
    In_read_index _G] =0;
    }
// 从 USART 缓冲区中读取数据
Recv_buffer[In_waiting_index_G] = SBUF;
if (In_waiting_index_G.< PC_LINK_RECV_BUFFER_LENGTH)
{
    // 加 1 且缓冲区不溢出
    In_waiting_index_G++;
}
RI =0; // 清除 RI 标志
}
*/
PC_LINK_Write_Char_To_Buffer()
在“写”缓冲区中存储字符，为后面的发送作准备
*/
void PC_LINK_Write_Char_To_Buffer(const char CHARACTER)
{
    // “只有”当缓冲区中有空间时才写入
    if (Out_waiting_index_G<PC_LINK_TRAN_BUFFER_LENGTH)
    {
        // Tran_buffer[Out_waiting_index_G] = CHARACTER;
        Out_waiting_index_G++;
    }
    else
    {
        // 写入缓冲区已满
        // 增加 PC_LINK_TRAN_BUFFER_LENGTH 的大小
        // 或提高 UART “刷新”函数被调用的频率
        // 或减少发送给 PC 的数据
        Error_code_G = ERROR_USART_WRITE_CHAR;
    }
}
*/
PC_LINK_Write_String_To_Buffer()
向字符缓冲区复制一个（以 null 结束的）字符串
*/
void PC_LINK_Write_String_To_Buffer(const char*const STR_PTR)
{
    tByte i = 0;
    while(STR_PTR[i] != '\0')
    {
        PC_LINK_Write_Char_To_Buffer(STR_PTR[i]);
    }
}
/*

```

```

PC_LINK_Get_Char_From_Buffer()
Retrieves a character from the (input) buffer, if available
*/
char PC_LINK_Get_Char_From_Buffer(void)// 如果有的话，从（输入）缓冲区中取回字符
{
    char Ch = PC_LINK_NO_CHAR;
    // 如果缓冲区中有新的数据
    if (In_read_index_G<In_waiting_index_G)
    {
        Ch = Rcv_buffer[In_read_index_G];
        if (In_read_index_G < PC_LINK_RECV_BUFFER_LENGTH)
        {
            In_read_index_G++;
        }
    }
    return Ch;
}

/*
PC_LINK_Send_Char()
基于Keil的例子代码，添加了（循环）超时实现了xon/off控制
*/
void PC_LINK_Send_Char(const char CHARACTER)
{
    tLong Timeout1 = 0;
    tLong Timeout2 = 0;
    if (CHARACTER == '\n')
    {
        if(RI)
        {
            if(SBUF == XOFF)
            {
                Timeout2 = 0
                do{
                    RI = 0
                    // 等待UART（有简单的超时）
                    Timeout1 = 0;
                    while((++Timeout1) &&(RI == 0));
                    if(Timeout1 == 0)
                    {
                        // UART没有响应——出错
                        Error_code_G = ERROR_USART_TI;
                        return;
                    }
                }while((++Timeout2) && (SBUF != XON));
                if (Timeout2 == 0)
                // USART没有响应——出错
                Error_code_G = ERROR_USART_TI;
                return;
            }
        }
    }
}

```

```

        RI = 0
    }
}
Timeout1 = 0;
while((++Timeout1) && (TI == 0));
if (Tineort1 == 0)
{
    // USART 没有响应——出错
Error_code_G = ERROR_USART_TI;
return;
}
TI = 0;
SBUF = 0X0d;// 输出 CR
}
if(RI)
{
    if(SBUF == XOFF)
    {
        Timeout2 = 0;
        do {
        RI = 0;
        // 等待 USART (有简单的超时)
        Timeout1 = 0;
        while((++Timeort1) && (RI == 0));
        if(Timeout1 == 0)
        {
            // USART 没有响应——出错
            Error_code_G = ERROR_RSART_TI;
            return;
        }
        } while((++Timeout2) && (SBUF != XON));
        RI = 0;
    }
}
Timeout = 0;
while((++ Timeout1) && (TI == 0));
if (Timeout1 == 0)
{
    // UART 没有响应——出错
Error_code_G = ERROR_USART_TI;
return;
}
TI = 0;
SBUF = CHARACTER;
}
/*
----文件结束-----
*/

```

源程序清单 18.5 菜单控制的通用 PC 连接库的一部分

例子：使用专用的波特率发生器

正如在“解决方案”中表明的，一些 8051 微控制器有专用于产生波特率的定时器。这是一个有用的功能，因为这样通常可以有三个定时器可供调度器用来创建延迟等功能。

在这个例子中，将介绍用于 Infineon 的 C515C 的“只有输出”的库（源程序清单 18.6~源程序清单 18.11），这里将使用 C515C 的波特率发生器。在这个例子中，将在 PC 或类似终端的屏幕上显示消逝的时间。

在这个例子中有两个要点要注意：

- 与前面介绍的“全双工”库相比较，这个库更小而且更简单。
- 微控制器运行在 10MHz。尽管如此，内部波特率发生器可以产生非常精确的波特率用于串行连接，这是一种非常有用的特性。

```
/*
Port.H (v1.01)
-----
项目 0_T2_IN 的“端口头文件”（参见第 10 章）
*/
// ----- Sch51.C -----
// 如果不需要错误报告，则将这一行注释掉
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1
#endif
// ----- Lnk_IO.C -----
// 引脚 3.0 和引脚 3.1 用于 RS-232 标准接口
/*
----- 文件结束 -----
*/
```

源程序清单 18.6 使用 Infineon 的 C515C 上专用于产生波特率的定时器的例子的一部分

```
/*
Main.c (v1.00)
-----
用于基于 c515c 的只有输出的 PC 连接库的测试程序
链接程序选项
OVERLAY (main ~ (PC_LINK_Update,Elapsed_Time_RS232_Update),
          SCH_Dispatch_Tasks ! (PC_LINK_Update,Elapsed_Time_LCD_Update))
*/
#include "Main.h"
#include "2_01_101.h"
#include "lnk_O_B.h"
#include "Elap_232.h"
/* ..... */
```

```

/*
void main(void)
{
    // 设置调度器
    SCH_Init_T2();
    // 波特率设置为 9600，使用内部波特率发生器
    PC_LINK_Init_Internal(9600);
    // 准备消逝时间的库
    Elapsed_Time_RS232_Init();
    // 必须调度这个任务（这里~100x 每秒就已经足够）
    // -这里向 PC 写数据
    //
    // 定时单位为时标 (1ms 间隔)
    SCH_Add_Task(PC_LINK_Update, 10, 10);

    // 每秒刷新一次时间
    SCH_Add_Task(Elapsed_Time_RS232_Update, 1000, 1000);

    SCH_Start();
    while(1)
    {
        // 在 P4 上显示错误代码 (参见 Sch51.C)
        SCH_Dispatch_Tasks();
    }
}
/*
-----文件结束-----
*/

```

源程序清单 18.7 使用 Infineon 的 C515C 上专用于产生波特率的定时器的例子的一部分

```

/*
Elap_232.C (v1.00)

用于跟踪消逝时间的简单的库函数通过 RS232 连接在 PC 屏幕上显示时间的演示版本
*/
#include "Main.h"
#include "Elap_232.h"
#include "lnk_O.h"
// -----公有变量定义-----
tByte Hou_G = 0;
tByte Min_G = 0;
tByte Sec_G = 0;
// -----公有变量声明-----
// 参见 Char_Map.c
extern const char code CHAR_MAP_G[10];
/*
Elapsed_Time_RS232_Init()

```

```
// 用于在PC屏幕上显示消逝时间的简单的库的初始化函数
/*
void Elapsed_Time_RS232_Init(void)
{
    char Time_Str[] = "Elapsed time";
    PC_LINK_Write_String_To_Buffer(Time_Str);
}

/*
Elapsed_Time_RS232_Update()
// 用于在PC屏幕上显示消逝时间的函数
***必须每秒调度一次***
*/
void Elapsed_Time_RS232_Update(void)
{
    char Time_Str[30] = "\rElapsed time: ";
    if (++Sec_G == 60)
    {
        Sec_G = 0;
        if (++Min_G == 60)
        {
            Min_G = 0;

            if (++Hou_G == 24)
            {
                Hou_G = 0;
            }
        }
    }

    Time_Str[15] = CHAR_MAP_G[Hou_G / 10];
    Time_Str[16] = CHAR_MAP_G[Hou_G % 10];
    Time_Str[18] = CHAR_MAP_G[Min_G / 10];
    Time_Str[19] = CHAR_MAP_G[Min_G % 10];
    Time_Str[21] = CHAR_MAP_G[Sec_G / 10];
    Time_Str[22] = CHAR_MAP_G[Sec_G % 10];
    // 在这个版本中不显示秒,
    // 使用秒数据来打开和关闭小时和分钟之间的冒号
    if ((Sec_G % 2) == 0)
    {
        Time_Str[17] = ':';
        Time_Str[20] = ':';
    }
    else
    {
        Time_Str[17] = ' ';
        Time_Str[20] = ' ';
    }
}
```

```

    }
    PC_LINK_Write_String_To_Buffer(Time_Str);
}
/*-----文件结束-----*/
/*-----*/
```

源程序清单 18.8 使用 Infineon 的 C515C 上专用于产生波特率的定时器的例子的一部分

```

/*-----*
Char_Map.C (v1.00)

这个保存在 ROM 中的查找表格用来把“整数”值转换为相应的字符代码,
用于 LCD 显示和 RS232 连接
*-----*/
// -----公用的常数-----
const char code CHAR_MAP_G[10]
    = {'0','1','2','3','4','5','6','7','8','9'};

/*-----文件结束-----*/
/*-----*/
```

源程序清单 18.9 使用 Infineon 的 C515C 上专用于产生波特率的定时器的例子的一部分

```

/*-----*
Lnk_o_B.c (v1.00)

简单的 PC 连接库的 B 版本（用于 c515c、内部波特率）使用 USART 和引脚 3.1 (Tx)
详细资料参见上下文
*-----*/
#include "Main.h"
#include "Lnk_O_B.h"
//-----公有变量声明-----
extern tByte Out_written_index_G;
extern tByte Out_waiting_index_G;
/*-----*
PC_LINK_Init_Internal
//这个版本使用 C5x5 系列的内部波特率发生器。
*-----*/
void PC_LINK_Init_Internal(const tWord BAUD_RATE)
{
    tWORD SRELplus1024;
    tword SRELx;
    PCON &= 0X7F; // 设置 SHOD 位为 0 (波特率不加倍)
    // 禁止接收器
    // 8 位数据、1 个起始位、1 个停止位和可变波特率 (异步)
    SCON = 0X42;
    // 使用内部波特率发生器 (80c5x5 系列)
    ADCON0 |= 0X80;
```

```

// 设置波特率(开始)
SRELplus1024 = (tWord) (((tLong)OSC_FREQ / 100*3125 )
    / ((tLong) BAUD_RATE*1000));
SRELx = 1024 - SRELplus1024;
SRELL = (tByte) (SRELx& 0X00FF);
SRELL = (tByte)(SRELx & 0X0300)>>8;
TI = 1;
// 设置波特率(结束)
// 设置用于写入的缓冲区
Out_written_index_G = 0;
Out_waiting_index_G = 0;
PC_LINK_Write_String_To_Buffer("Serial interface initialized.\n");
// 不使能串行中断
ES = 0;
}
/*-----文件结束-----*/

```

源程序清单18.10 使用Infineon的C515C上专用于产生波特率的定时器的例子的一部分

```

/*-----*
 Lnk_0.c(v1.00)

 用于8051系列的简单的只写PC连接库的内核文件(向PC发送数据,不能从PC接收数据)
 使用USART和引脚3.1(TX),
 详细资料参见上下文
 *-----*/
#include "Main.h"
#include "Lnk_0.h"
//-----公有变量定义-----
tByte Out_written_index_G; // 已经写入的数据输入缓冲区
tByte Out_Waiting_index_G; // 还没有写入的数据输入缓冲区
//-----公有变量声明-----
错误代码变量
//将用来显示错误代码的端口以及错误代码的详细资料参见Port.H
extern tByte Error_code_G;
//-----私有的常数-----
// 发送缓冲区的长度
#define PC_LINK_TRAN_BUFFER_LENGTH 100
//-----私有变量-----
static tByte Tran_buffer[PC_LINK_TRAN_BUFFER_LENGTH];
/*-----*
 PC_LINK_Update()
 //将软件发送缓冲区中的下一个字符发送,注意,这是只输出的库(不能接收字符)
 *-----*/
void PC_LINK_Update(void)
{
    // 这里处理发送字节
}

```

```

// 是否有数据准备发送?
if(Out_written_index_G<Out_waiting_index_G)
{
    PC_LINK_Send_Char(Tran_buffer[Out_written_index_G]);
    Out_written_index_G++;
}
else
{
    // 没有数据需要发送-仅仅复位缓冲区指针
    Out_waiting_index_G = 0;
    Out_written_index_G = 0;
}
}

/*-----*
PC_LINK_Write_Char_To_Buffer()
在“写”缓冲区中存储字符，为后面的发送作准备
-*-----*/
void PC_LINK_Write_Char_To_Buffer(const char CHARACTER)
{
    // “只有”当缓冲区中有空间时，才写入
    if (Out_waiting_index_G<PC_LINK_TRAN_BUFFER_LENGTH)
    {
        Tran_buffer[Out_waiting_index_G] =CHARACTER;
        Out_waiting_index_G++;
    }
    else
    {
        // 写入缓冲区已满
        // 增加 PC_LINK_TRAN_BUFFER_LENGTH 的大小
        // 或提高 UART “刷新”函数被调用的频率
        // 或减少发送给 PC 的数据
        Error_code_G = ERROR_USART_WRITE_CHAR;
    }
}

/*-----*
PC_LINK_Send_Char()
基于 Keil 的例子代码，添加了（循环）超时
实现了 Xon/Off 控制
-*-----*/
void PC_LINK_Send_Char(const char CHARACTER)
{
    tLong Timeout1 =0;
    if (CHARACTER == '\n')
    {
        Timeout1 = 0;
        while (((++Timeout1) && (TI == 0)));
        if (Timeout1 == 0)
        {
            // USART 没有响应-出错
        }
    }
}

```

```

        Error_code_G = ERROR_USART_TI;
        return;
    }
    TI = 0;
    SBUF = 0xd; // 输出 CR
}
Timeout1 = 0;
while ((++Timeout1) && (TI == 0));
if (Timeout1 == 0)
{
    // USART 没有响应——出错
    Error_code_G = ERROR_USART_TI;
    return;
}
TI = 0;
SBUF = CHARACTER;
}

/*-----*
PC_LINK_Write_String_To_Buffer()
向字符缓冲区复制一个（以 null 结束的）字符串（然后缓冲区中的内容将通过串行连接传送）
-*-----*/
void PC_LINK_Write_String_To_Buffer(const char* const STR_PTR)
{
    tByte i = 0;
    while (STR_PTR[i] != '\0')
    {
        PC_LINK_Write_Char_To_Buffer(STR_PTR[i]);
        i++;
    }
}
/*-----*
-----文件结束-----
-*-----*/

```

源程序清单 18.11 使用 Infineon 的 C515C 上专用于产生波特率的定时器的例子的一部分

例子：用于（通用）8051/8052 的只有输出的库

这个例子再次说明了如何将标准 8051 芯片连接到 PC 上。

不同于第一个例子，这里的软件是“只写”的。数据从微控制器传送到 PC，然而不允许反向传送。为了说明这个例子，软件在 PC 上通过终端仿真程序显示消逝的时间。

在图 18.6 中给出了这个软件运行在 Keil 硬件模拟器上的结果。毫无疑问，它同样可以使超级终端（或类似软件）显示其输出，如图 18.3 所示。然而，在程序开发的早期使用硬件模拟器是很有用的。

仍然可以在这里使用图 18.2 中显示的硬件来运行这个程序。

这个例子的源代码包含在 CD 上。



图 18.6 简单的“只有输出”的 PC 连接库在 Keil 硬件模拟器上的运行输出

进阶阅读

Axelson (1998) 是一本非常值得一读的书，该书讨论了 RS-232 通信的使用。虽然该书也讨论了 8051 微控制器，但是其主要的讨论集中于通信连接的 PC 端。Axelson 提供了一些有用的代码例子，通过改编可以用于许多基于 PC 的数据采集和监测系统。

Chapter 19

开关接口

引言

读取一个或多个按键开关的状态是嵌入式系统中很常见的要求。本章中的四个模式描述了对这个问题的不同解决方案。

这些模式如下所示：

- 开关接口（软件）

使用最少的外部硬件。读取一个开关，消除抖动并报告它的状态。

- 开关接口（硬件）

使用外部硬件来完成开关防抖。虽然增加了成本[和开关接口（软件）相比]，然而是一种对ESD、恶意破坏等干扰的高层次的保护措施。

- 通断开关

这个模式建立在开关接口（软件）或开关接口（硬件）之上，通过软件提供下列特性：

假设开关处于断开状态。在被按下以前一直保持这个状态。当被按下时，状态改变为接通。在下一次被按下以前一直保持这个状态。当被按下时，状态又改变为断开。

这类特性可以用来只使用一个（非锁存的）开关来控制一台机器。

- 多状态开关

这个模式建立在开关接口（软件）或开关接口（硬件）之上，通过软件提供下列特性：

假设开关处于断开状态。开关被按下并暂时保持，开关状态改变为“接通状态1”。用户继续按下开关，开关状态变成“接通状态2”，等等。在任何时候，释放开关都将使开关返回断开状态。

也就是说，开关状态取决于开关被按下的延续时间。可以支持任意个“接通”状态（通常三个以上的接通状态将使用户非常容易混淆）。

例如，当设置时钟的时间时，这类特性非常有用。按下设置开关将使显示的时间开始慢慢地改变，持续地按下开关将导致显示的时间增加得更快。

注意：因为在开关接口（软件）中讨论的理由，在“可靠性和安全性”中，将只涉及

这些模式中的按键开关，而不讨论“乒乓开关”或“锁存开关”（并且不推荐使用这样的开关）。

开关接口（软件）

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 正在创建该系统的用户界面。

问题

如何将 8051 微控制器的端口引脚连接到机械开关上（例如，简单的按键开关或机电继电器）？

背景知识

考虑图 19.1 所示的简单按键开关。在这两个方案中，按下开关都将导致输入端口的电压由 Vcc（左右）变为 0V。

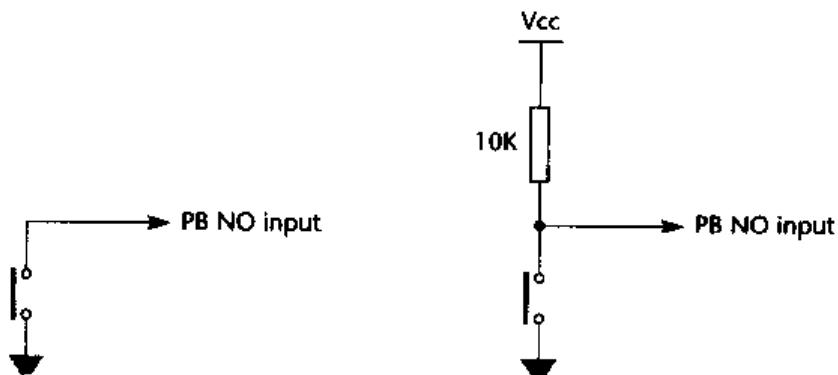


图 19.1 “常开”按键开关输入的例子

注意，左边没有电阻的方案适用于具有内部上拉电阻的端口引脚，8051 系列通常就是这样。然而端口 0 没有内部上拉电阻，必须使用右边的方案。

在理想情况下，电压将以图 19.2（上面）所示的形式改变。然而实际上，所有机械开关的触点在闭合或打开之后都会抖动（即在一段短时间内重复地导通、断开）。因此，实际的输入波形更像是图 19.2（下面）所示的波形。通常，开关抖动小于 20ms。然而大型的机械开关的抖动特性可以在 50ms 以上。

这些抖动相当于多次按下理想化的开关，这带来了各种潜在问题：

- 如何区分开关被按下了一次还是多次。例如，将从键盘读取“AAAAAA”，而不是“A”。

- 如何对按下开关的次数进行计数。
- 如何区分开关的按下与释放。例如，如果开关被按下一次后经过一段时间被释放，抖动将使开关看起来像是被再次按下。

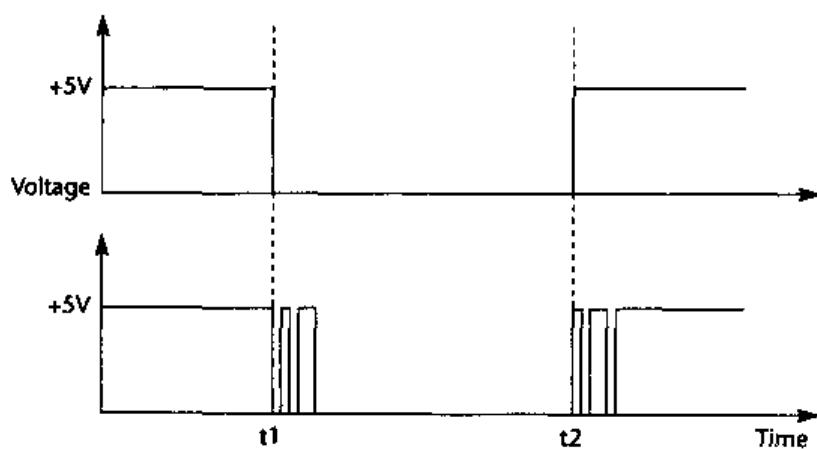


图 19.2 图 19.1 中所示的开关产生的电压信号

注意，上面为在 t_1 时刻按下开关并且在 t_2 时刻释放开关产生的理想化波形。下面为实际波形，显示了在按下开关之后的上升沿抖动，和在释放开关之后的下降沿抖动。

解决方案

实际上，检查开关的输入很简单，如下所示：

1. 取有关的端口引脚。
2. 如果检测到开关被按下，将在 20ms 之后再次读取这个引脚。
3. 如果第二次的读取确认了第一次的结果，那么认为开关确实被按下。

注意：“20ms”的数值将取决于所使用的开关。开关的数据手册将提供这些信息。如果没有数据手册，则可以试验不同的数值或直接使用示波器测量。

我们将在例子中说明这些基本操作。

硬件资源

通过软件读取开关输入将占用非常少的 CPU 负荷和存储器资源。

可靠性和安全性

这里讨论一些有关开关的可靠性和安全性的问题。

锁存开关和按键开关的比较

本章的所有例子（并且贯穿本书）都集中讨论使用按键开关（而不是“锁存开关”）。

为了说明为什么是这样，将讨论设计基于开关的接口的两种可能方案（图 19.3）。第一种接口设计使用“锁存”开关。具体地说，这是一个有 5 个位置的旋转开关。第二种设计使用两

个按键开关和五个 LED 来提供类似的功能。

当讨论在第一次上电和遇到紧急情况的场合时，可以立即发现按键解决方案的一些优点。

就按键解决方案而言，软件设计人员能够控制初始状态。例如，如果希望系统起始时总是处于“状态 2”（如图 19.3 所示），这将很容易实现。类似的，假如状态 5 与控制一台机器有关。如果机器出现问题，则可能需要将系统状态转为状态 4。同样，可以通过软件来控制这种转变，并通过点亮相应的 LED 将该变化通知用户。

旋转开关接口与之有很大不同。如果用户在接通电源之前将开关转到位置 5，那么在上电时，如何将系统状态转为状态 2？并且即使确实将状态改变为状态 2，仍然无法将这个变化显示给用户，因为程序不能移动旋转开关。类似的，在刚才讨论的紧急情况下，也许能通过软件控制将机器关闭，然而无法向用户显示当前的系统状态是状态 4。

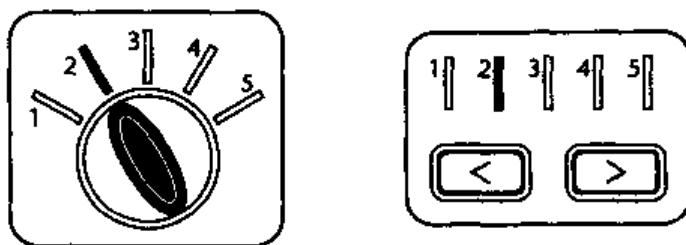


图 19.3 两种基于开关的接口

注意，由于在正文中讨论的原因，不推荐使用（左边的）锁存解决方案。

不仅是多状态旋转开关，所有的锁存开关都有这些问题。

按键解决方案的优点可以总结为：它使软件可以控制硬件，反之则不行。这是嵌入式系统的一种良好的一般设计准则。

开关抖动时间

在电机工程术语中，典型的微控制器系统里用作输入的开关是“干燥的”。这指的是，因为它们只处理小电流（<20mA）和低电压（<10V），它们的（电的）“磨损”很小，这样应该可以具有较长的寿命。

然而，开关包含机械部件，其性能不可避免地将随寿命而变化。即使优质的开关也将在使用过程中改变其性能。特别是，抖动时间将随寿命而增加。因此，在确定相应的抖动时间时，重要的是不要听天由命或只对新开关进行试验。应该考虑到长期使用的影响，而在新开关的基础上加一些裕度（至少 20%）。

使用单极开关

前面已经讨论了按键开关（PB）的优点。然而，只使用按键开关不足以保证可靠性。例如，再次考虑图 19.1 中的开关。这类开关往往被称为是“常开”（NO）。如果这种常开开关被移去或损坏，使得连接永久打开，将无法在软件中检测到。这将造成安全性或可靠性问题。

在一些情况下，使用“常闭”(NC)按键开关(图19.4)是有好处的。使用常闭按键开关，可以检测到开关被移去或接线被破坏，虽然通常无法将这些情况和正常的(持续)按下开关的情况区分开来。

通过使用“双刀双掷”按键开关(PB-DPDT)(图19.5)可以提供更可靠的解决方案。这将产生两个输入，如果开关没有损坏并且接线正确，这两个输入将始终具有相反的逻辑。使用这样的输入设备，可以检测各种开关故障(包括开关被移去)以及导线损坏(包括导线被切断)。

注意：与单刀开关输入相比，这种开关需要两个输入引脚以及稍多的软件，

超出范围的输入和ESD

与所有只基于软件的输入方法相似，开关接口(软件)不提供对超出范围输入的保护。如果(错误地或故意地)在开关上加 $+/-20V$ ，将可能“烧坏”微控制器。

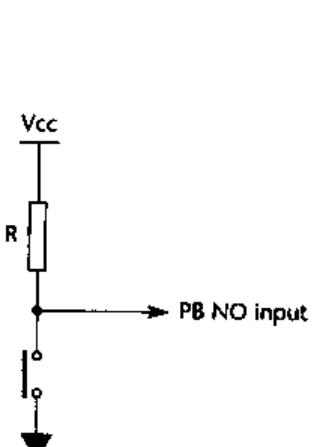


图19.4 常闭按键开关输入的例子

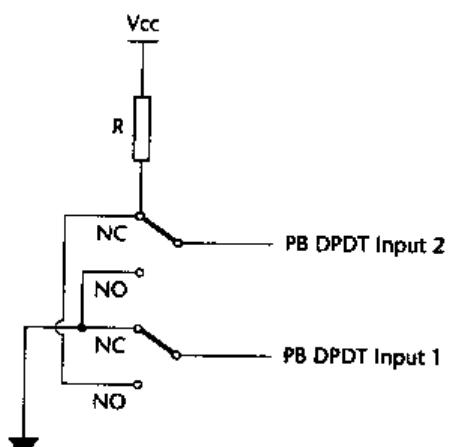


图19.5 双刀双掷按键开关输入的例子

注意，在图中显示的配置下，当开关打开时输入1的电压为0V，而输入2的电压为Vcc。当开关闭合时，这些值将被颠倒。

类似的问题在发生静电放电(ESD)时也会出现。在恶劣环境(例如工业系统、汽车应用)下，ESD可能造成严重问题，一些应用要求符合这些领域的国际标准(例如IEC 1000-4-2)。

在这些应用场合用软件实现保护几乎无能为力。关于这个问题的讨论和解决方案参见“开关接口(硬件)”一节。

可移植性

所有关于这个模式的讨论都不是只针对8051的，这个模式可以毫无困难地用于其他微控制器系列。

优缺点小结

- ☺ 开关接口(软件)需要极少的外部硬件。
- ☺ 它非常灵活。例如，程序员几乎不改动代码就可以加入“自动重复”功能。

- ⑤ 实现简单而便宜。
- ⑥ 与所有只基于软件的输入方法相似，开关接口（软件）不提供对超出范围输入或静电放电（ESD）的保护。这个问题的讨论和解决方案参见“开关接口（硬件）”一节。

相关的模式和替代解决方案

读取开关输入是折中硬件成本和软件复杂性的一个范例。“开关接口（硬件）”一节讨论了使用硬件的一种替代方案。

如果开关（或其他输入设备）没有抖动（例如，当输入来自固态继电器或其他形式的“电子开关”时），那么可以通过使用端口输入/输出模式（第 10 章）来大大降低处理这些输入的复杂性。

例子：控制 LED 闪烁

在这个例子中，使用按键开关来控制 LED 的闪烁，在这里，LED 只有当开关被按下时才闪烁。

所用的硬件如图 19.6 所示，软件框架在源程序清单 19.1～源程序清单 19.4 中给出。

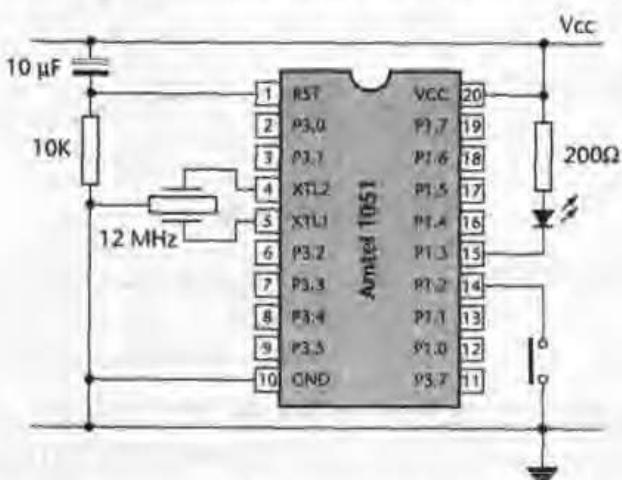


图 19.6 用来测试简单的基于软件的开关接口的一些硬件

```
/*
Port.H (v1.00)
-----*
项目 SWITCH_A 的“端口头文件”（参见第 10 章）
*/
// ----- Sch51.C -----
// 如果不需要错误报告，那么将这一行注释掉
// #define SCH_REPORT_ERRORS
// #ifdef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
```

```

#define Error_port P2
#endif
// -----Swit_A.C -----
// 连接一个按键开关到这个引脚[到地]
// -软件消抖
sbit Sw_pin = P1^2; // 开关引脚
// ----- LED_Swit.C -----
// +5V 串接适当的电阻连接 LED 到这个引脚
// [详细资料参见第 7 章]
sbit LED_pin = P1^3;
/*-----文件结束-----*/
*/

```

源程序清单 19.1 用来展示简单的基于软件的开关接口的部分软件

```

/*-----*
Main.c (v1.00)
-----*
开关接口(软件)模式的演示程序, 详细资料参见第 19 章
要求的链接程序选项(详细资料参见正文):
OVERLAY (main ~ (SWITCH_Update, LED_Flash_Switch_Update),
SCH_Dispatch_Tasks ! (SWITCH_Update,LED_Flash_Switch_Update))
-----*/
#include "Main.h"
#include "Swit_A.h"
#include "0_01_12g.H"
#include "LED_Swit.h"
/* ..... */
/* ..... */
void main(void)
{
    {
        // 设置调度器
        SCH_Init_T0();
        // 设置开关引脚
        SWITCH_Init();

        // 为“Flash_LED”任务作准备
        LED_Flash_Switch_Init();
        // 添加一个“SWITCH_Update”任务, 每隔 200ms
        // 调度器定时单位为时标
        // [1ms 时标间隔——参见 Sch “初始化”函数]
        SCH_Add_Task(SWITCH_Update, 0, 200);
        // 添加 LED 任务
        // 这里, LED 将只在开关被按下时闪烁...
        SCH_Add_Task(LED_Flash_Switch_Update, 5, 1000);
        SCH_Start();
        while(1)
        {
            SCH_Dispatch_Tasks();
        }
    }
}

```

```

        }
    }
/*-----*
---文件结束-----
*-----*/

```

源程序清单 19.2 用来显示简单的基于软件的开关接口的部分软件

```

/*-----*
 LED_Swit.C (v1.00)

 用于调度器的简单的“闪烁 LED”的测试函数（通过按下开关来控制）
*-----*/
#include "Main.h"
#include "Port.h"
#include "LED_Swit.h"
// -----公有变量声明-----
extern bit Sw_pressed_G;
// -----私有变量-----
static bit LED_state_G;
/*-----*
 LED_Flash_Switch_Init()
 - See below.
*-----*/
void LED_Flash_Switch_Init(void)
{
    LED_state_G = 0;
}
/*-----*
 LED_Flash_Switch_Update()
 在指定端口引脚上闪烁 LED（或产生脉冲给蜂鸣器，等等）
 必须按需要的闪烁频率的两倍计时。这样，对于 1Hz 的闪烁（0.5 秒亮，0.5 秒灭）
 必须以 2Hz 计时
*-----*/
void LED_Flash_Switch_Update(void)
{
    // 如果开关没有被按下，则什么也不做
    if (!Sw_pressed_G)
    {
        return;
    }
    // 使 LED 从灭变亮（反之亦然）
    if (LED_state_G == 1)
    {
        LED_state_G = 0;
        LED_pin = 0;
    }
    else
    {
        LED_state_G = 1;
    }
}

```

第19章 开关接口

```
    LED_pin = 1;
}
}

/*
---文件结束---
*/

```

源程序清单 19.3 用来展示简单的基于软件的开关接口的部分软件

```
/*
-----*
SWIT_A.C (v1.00)

简单的开关接口代码，包括软件消除抖动
*-----*/
#include "Main.h"
#include "Port.h"
#include "Swit_A.h"
// -----公有变量定义-----
bit Sw_pressed_G = 0; // 当前的开关状态
// -----私有的常数-----
// 可以使用常开或常闭开关（或其他接线变化）
#define SW_PRESSED (0)
// 要得到正确的消抖特性，SW_THRES 必须>1
#define SW_THRES (3)
*-----*
SWITCH_Init()
开关库的初始化函数
*-----*
void SWITCH_Init(void)
{
    Sw_pin = 1; // 使用这个引脚用于输入
}
*-----*
SWITCH_Update()
这是主要的开关函数
应该每隔 50~500ms 调度一次
*-----*
void SWITCH_Update(void)
{
    static tByte Duration;
    if (Sw_pin == SW_PRESSED)
    {
        Duration += 1;
        if (Duration > SW_THRES)
        {
            Duration = SW_THRES;
            Sw_pressed_G = 1; // 开关被按下...
            return;
        }
    }
    // 开关被按下，然而时间不够长
}
```

```

    Sw_pressed_G = 0;
    return;
}
// 开关没有被按下——复位计数
Duration = 0;
Sw_pressed_G = 0; // 开关没有被按下...
}
/*-----*
---文件结束-----
*-----*/

```

源程序清单 19.4 用来展示简单的基于软件的开关接口的部分软件

进阶阅读

开关接口（硬件）

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 正在创建该系统的用户界面。

问题

如何创建用于恶劣环境下(例如工业应用和汽车应用)的具有很强抗干扰能力的开关接口?

背景知识

考虑下列情况:

- 嵌入式系统用于可能有能量较高的静电放电的工业环境中,如何保证器件仍然运行正常?
- 汽车上的安全系统可能受到故意破坏。具体而言,WWW 网站上有报告揭示了小偷已经发现通过把 12V 的汽车蓄电池直接连到某个系统开关就有可能使类似的安全防范系统无效。如何保证系统更不易受到干扰?

总之,开关接口(软件)模式描述的方法只适用于“安全的”系统。在恶劣环境下,需要一种更不易受干扰的解决方案。这必须是基于硬件的。

解决方案

正如在“背景知识”中表明的,创建不易受干扰的开关接口需要使用片外硬件。

传统上，使用包括 J-K 触发器、高阻抗 CMOS 门或电阻电容积分器等各种方法来消除开关抖动。例如，Huang (2000) 提供了这些方法的详细资料。总之，这些方法虽然能够完成消抖操作，但是对 ESD 和类似的干扰只能提供非常有限的防护。正如在“开关接口（软件）”一节中看到的，因为在调度应用中开关消抖的处理是微不足道的，所以只为开关消抖而增加外部硬件成本的做法通常难以被接受。

近年来，市场上出现了几种用于保护和开关消抖的专用芯片，其中，Maxim 的 6816/6817/ 6818 系列是不错的例子（图 19.7）。

Maxim 对这些芯片的描述如下：

- Max6816/Max6817/Max6818 分别是单、双、八开关消抖芯片，为数字系统提供净化过的机械开关接口。输入一个或多个抖动的机械开关信号，在短暂的预置延迟之后生成净化的数字输出。打开和闭合开关的抖动都被移去。
- 输入最多可以超出电源电压 $\pm 25\text{V}$ 。
- 对输入引脚的 ESD 保护：

$\pm 15\text{kV}$	人体模式
$\pm 8\text{kV}$	IEC 1000-4-2, 触点放电
$\pm 15\text{kV}$	IEC 1000-4-2, 气隙放电
- 使用 $+2.7\text{V} \sim +5.5\text{V}$ 的单电源。
- 有单 (Max6816)、双 (Max6817) 和八 (Max6818) 开关可选。
- 不需要外部元件。
- 消耗 $6\mu\text{A}$ 电流。

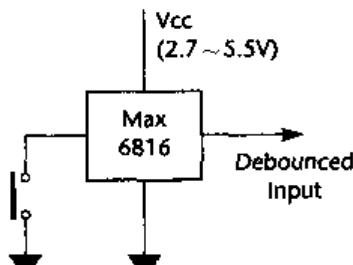


图 19.7 Max6816 的典型应用电路

硬件资源

读取消抖后的开关输入将占用非常少的 CPU 负荷和存储器资源。

可靠性和安全性

因为在“解决方案”中讨论的理由，这种方法是创建开关接口的一种非常可靠的方法。

如需要更高的可靠性，尤其是在恶意破坏的情况下应用时，参见“开关接口（软件）”一节中关于使用多极开关的讨论。

可移植性

这些技术本质上是可移植的。

优缺点小结

- ◎ 极大地提高了在恶劣环境下的可靠性（和只基于软件的解决方案相比）。

⑧ 增加了成本和硬件复杂性。

相关的模式和替代解决方案

参见“开关接口（软件）”一节。

例子：在恶劣环境下读取 8 个输入开关

图 19.8 说明了使用 Max6818 读取连接到 8051 芯片的端口 1 上的 8 个开关的输入，结果在端口 2 上显示。

使用超级循环模式（第 9 章）可以很容易地满足软件上的要求，如源程序清单 19.5 所示。

```
void main(void)
{
    while(1)
    {
        P2 = 01;
    }
}
```

源程序清单 19.5 很普通的超级循环开关接口

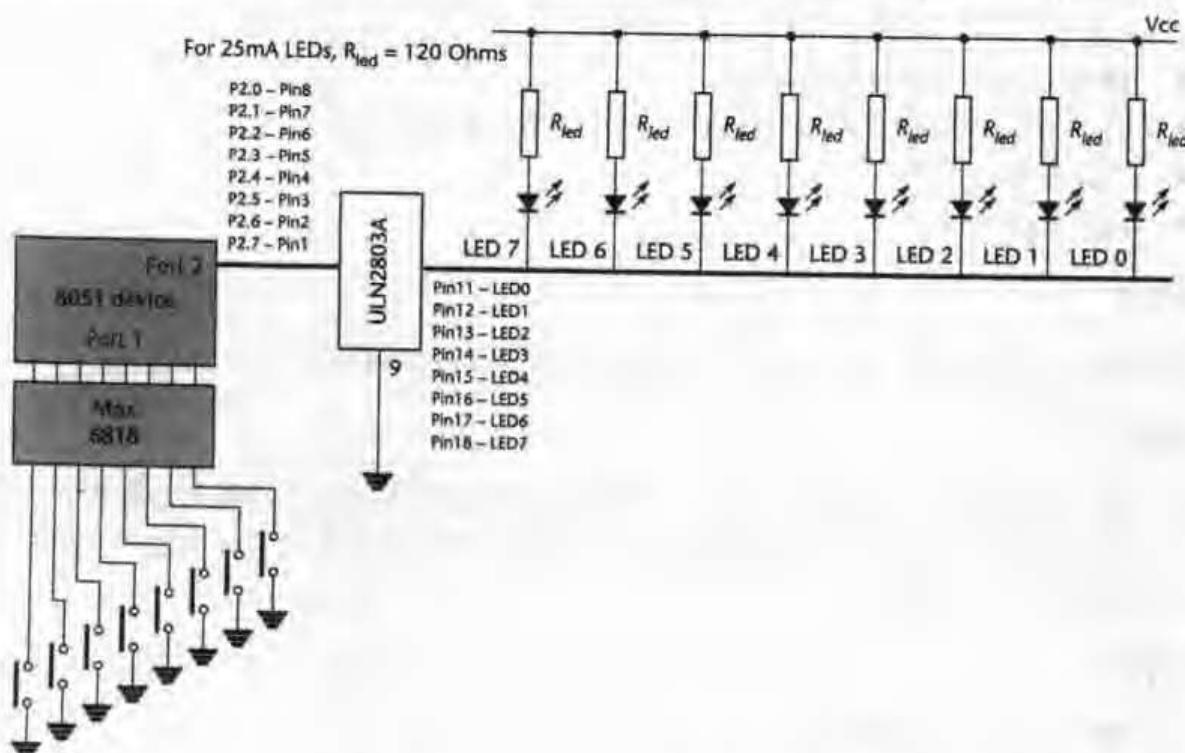


图 19.8 使用 Max6818 对 8 个开关消抖

进阶阅读

通断开关

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 正在创建该系统的用户界面。

问题

如何从一个连接到微控制器端口引脚的按键开关获得如图 19.9 所示的特性？

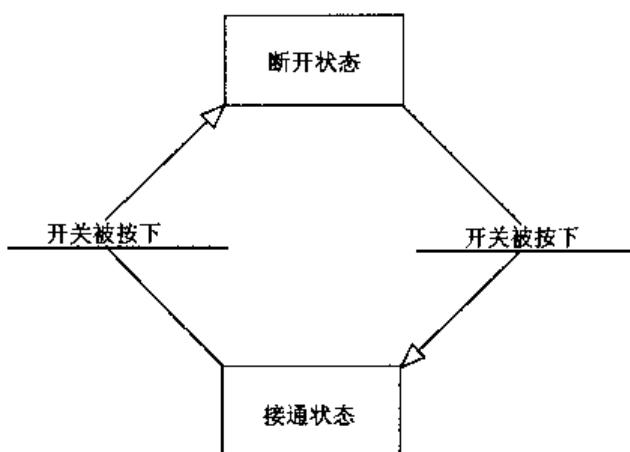


图 19.9 使用一个按键开关创建“通断”（锁存）特性

背景知识

考虑当使用一个开关来打开和关闭一台设备时出现的问题（图 19.10）。

要求开关运行如下：

- 用户按下开关来开启一台设备。
- 设备按要求运行。
- 用户再次按下开关来关闭设备。

这好像非常简单。然而，假设使用开关接口（软件）模式或开关接口（硬件）模式中给出的基本方法来读取开关，可以概括如下：

1. 读取有关的端口引脚。
2. 如果检测到开关被按下，将在 100ms 之后再次读取这个引脚。

3. 如果第二次读取确认了第一次的结果，那么认为开关确实被按下。

可能发生以下情况：

- 用户按下开关来开启设备。
- 检查开关：它被按下。
- 在（比方说）100ms 之后再次检查开关。第二次检查确认了第一次的状态，则启动设备。
- 100ms 之后再次检查开关：它仍然被按下。
- 100ms 之后再次检查开关。第二次检查确认了第一次的状态，则再次关闭设备。
- 如此等等。

出现这个问题是因为用户通常将会等待，直到设备开始工作才将手指从开关上移开。在最好的情况下，按下开关也很可能将维持大约 500ms。除非采取预防措施，否则设备将会“来回”启动关闭。

解决方案

创建通断开关的最简单的方法是添加针对现有接口代码的“开关阻塞”计数器。作用如下：

1. 每当发现开关被按下时，便“阻塞”它，比方说 1 秒钟。
2. 当开关被阻塞时，忽略开关状态的任何变化。这样，如果用户按下开关保持半秒钟，就不会出现前面的问题。

下面的“通断开关”的例子说明了在实际情况中这些是如何实现的。

硬件资源

读取开关输入占用非常少的 CPU 负荷和存储器资源。

可靠性和安全性

在恶劣环境中，这些代码应该基于开关接口（硬件）模式。

注意，在特别关注安全的场合，阻塞开关输入的方法可能不合适。一种替代方案是保留基本的开关处理方法，然而使用两个开关（图 19.11）。

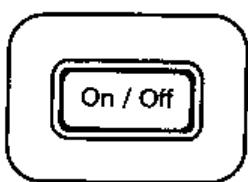


图 19.10 “通断”问题

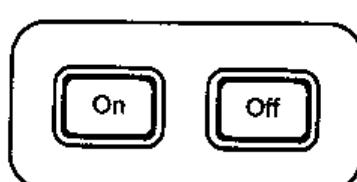


图 19.11 解决“通断”问题的一种方法

在某些情况下，使用两个开关可以使对输入变化的反应更快速，并且有时更可靠。

可移植性

这个模式可以毫不费力地改编以用于其他微控制器。

优缺点小结

◎ 是一种用一个开关实现“通断”特性的简单方法。

相关的模式和替代解决方案

这个模式基于开关接口（软件）模式及开关接口（硬件）模式。替代方案参见“可靠性和安全性”。

例子：控制 LED 闪烁

在这个例子中，使用图 19.12 中的硬件来说明如何创建通断开关接口。这个例子中的主要软件文档在源程序清单 19.6~源程序清单 19.9 中给出。和往常一样，CD 上包含了这个项目的所有源文件。

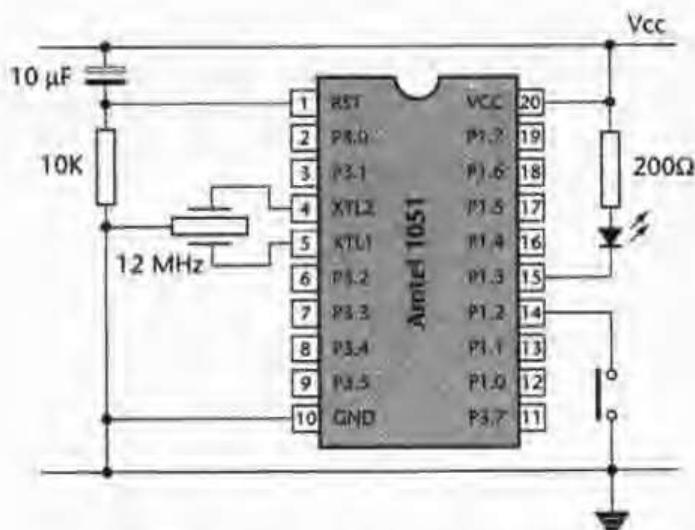


图 19.12 用于“通断”LED 演示的硬件

```
/*
Port.H (v1.00)

-----*
项目 ON_OFF 的“端口头文件”（参见第 10 章）
-----*/
// ----- Sch51.C -----
// 如果不需要错误报告，那么将这一行注释掉
// #define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
```

```

// 只在报告错误时使用
#define Error_port P1
#endif
// -----Swit_C.C -----
// 连接一个按键开关到这个引脚[到地]
// - 软件消抖
sbit Sw_pin = P1^2; // 开关引脚
// ----- LED_Swit.C -----
// +5V 串接适当的电阻连接 LED 到这个引脚
// [详细资料参见第 7 章]
sbit LED_pin = P1^3;
/*
-----文件结束-----
*/

```

源程序清单 19.6 基于软件的开关接口示例的一部分

```

/*-----*
Main.c (v1.00)

用于通断开关模式的演示程序,
详细资料参见第 19 章
要求的链接程序选项(详细资料参见正文):
OVERLAY (main ~ (SWITCH_ON_OFF_Update, LED_Flash_Switch_Update),
SCH_Dispatch_Tasks ! (SWITCH_ON_OFF_Update,LED_Flash_Switch_Update))
-*-----*/
#include "Main.h"
#include "Swit_C.h"
#include "0_01_12g.H"
#include "LED_Swit.h"
/* ..... */
/* ..... */
void main(void)
{
    //
    // 设置调度器
    SCH_Init_T0();
    // 设置开关引脚
    SWITCH_ON_OFF_Init();
    // 为“Flash_LED”任务作准备
    LED_Flash_Switch_Init();
    // 添加一个“SWITCH_ON_OFF_Update”任务, 每隔 200ms
    // 调度器定时单位为时标。
    // [1ms 时标间隔——参见 Sch “初始化”函数]
    SCH_Add_Task(SWITCH_ON_OFF_Update, 0, 200);
    // 添加 LED 任务
    // 这里, LED 将只在开关在接通状态时闪烁
    SCH_Add_Task(LED_Flash_Switch_Update, 0, 1000);
    SCH_Start();
    while(1)
    {

```

```

        SCH_Dispatch_Tasks();
    }
}

/*
-----文件结束-----
*/

```

源程序清单 19.7 基于软件的开关接口示例的一部分

```

/*
-----*
 LED_Swit.C (v1.00)
 -----
 用于调度器的简单的“闪烁 LED”的测试函数（通过按下开关控制）
-----*/
#include "Main.h"
#include "Port.h"
#include "LED_Swit.h"
// -----公有变量声明-----
extern bit Sw_pressed_G;
// -----私有变量-----
static bit LED_state_G;
/*-----*
 LED_Flash_Switch_Init()
 - See below.
-----*/
void LED_Flash_Switch_Init(void)
{
    LED_state_G = 0;
}
/*-----*
 LED_Flash_Switch_Update()
 // 在指定的端口引脚上闪烁 LED（或产生脉冲给蜂鸣器，等等）
必须按需要的闪烁频率的两倍计时。这样，对于 1Hz 的闪烁（0.5s 亮，0.5s 灭）
必须以 2Hz 计时
-----*/
void LED_Flash_Switch_Update(void)
{
    // 如果开关没有被按下，则什么也不做
    if (!Sw_pressed_G)
    {
        return;
    }
    // 使 LED 从灭变亮（反之亦然）
    if (LED_state_G == 1)
    {
        LED_state_G = 0;
        LED_pin = 0;
    }
    else
    {

```

```

    LED_state_G = 1;
    LED_pin = 1;
}
}

/*
-----文件结束-----
*/

```

源程序清单 19.8 基于软件的开关接口示例的一部分

```

/*
-----*
SWIT_C.C (v1.00)

通断开关代码，包括软件消除抖动
*-----*/
#include "Main.h"
#include "Port.h"
#include "Swit_C.h"
// -----公有变量定义-----
bit Sw_pressed_G = 0; // 当前的开关状态
// -----私有的常数-----
// 可以使用常开或常闭开关（或其他接线变化）
#define SW_PRESSED (0)
// 要得到正确的消抖特性，SW_THRES 必须>1
#define SW_THRES (3)
// -----私有变量-----
static tByte Sw_press_duration_G = 0;
static tByte Sw_blocked_G = 0;
/*-----*
FUNCTION: SWITCH_ON_OFF_Init()
开关库的初始化函数
*-----*/
void SWITCH_ON_OFF_Init(void)
{
    Sw_pin = 1;           // 使用这个引脚用于输入
    Sw_pressed_G = 0;     // 开关最初是关闭的
    Sw_press_duration_G = 0;
    Sw_blocked_G = 0;
}
/*-----*
FUNCTION: SWITCH_ON_OFF_Update()
这是主要的通断开关函数，应该每隔 50~500ms 调度一次
*-----*/
void SWITCH_ON_OFF_Update(void)
{
    // 如果开关被阻塞，计数减 1 并且不检查开关引脚状态而返回
    // 这使用户有时间将手指从开关上移开，否则如果保持按下开关大于 0.4 秒，灯将会再次关闭
    if (Sw_blocked_G)
    {
        Sw_blocked_G--;
    }
}

```

```
    return;
}
if (Sw_pin == SW_PRESSED)
{
    Sw_press_duration_G += 1;
    if (Sw_press_duration_G > SW_THRES)
    {
        Sw_press_duration_G = SW_THRES;
        // 改变开关状态
        if (Sw_pressed_G == 1)
        {
            Sw_pressed_G = 0; // 改变开关状态为关闭
        }
        else
        {
            Sw_pressed_G = 1; // 改变开关状态为打开
        }
    }
    // 1 秒内不允许其他改变
    Sw_blocked_G = 5;
    return;
}
// 开关被按下，然而时间不够长
return;
}

// 开关没有被按下——复位计数
Sw_press_duration_G = 0;
}
/*
-----文件结束-----
*/
```

源程序清单 19.9 基于软件的开关接口示例的一部分

进阶阅读

多状态开关

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 正在创建该系统的用户界面。

问题

如何从一个连接到微控制器端口引脚的按键开关获得如图 19.13 所示的特性？

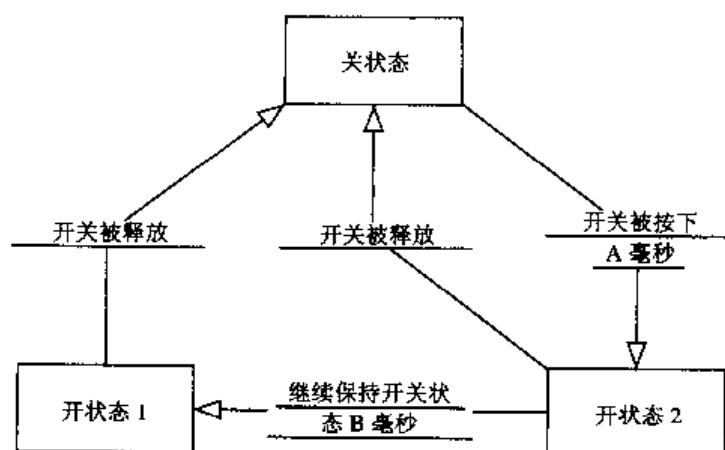


图 19.13 一个三态开关

背景知识

解决方案

虽然正如在“通断开关”中所讨论的，持续按下开关有时可能带来问题，但是它们也是形成基于开关的更复杂接口的基础，使我们能够将两态的输入设备变为三态（或更多状态）的输入设备。

例如，假如有一个实时时钟，只有两个按键（“前进”和“后退”）来设置时间。为了避免使这个过程变得非常乏味，可以规定短暂地按下“前进”按键将使显示的时间缓慢地增加，而持续地按下（比方说长于 5 秒），将使显示时间增加得更快。

为了实现这种特性，可以运行如下：

- 记录持续按下开关的时间。
- 当时间超过一个阈值（称做延续时间 A）时，认为这是正常地按下开关。
- 当时间超过一个更大的阈值（称做延续时间 B）时，认为这是持续地按下开关。

注意，可以添加更多的“级别”，然而级别大于两个可能会使用户混淆。

完整实现的详细资料参见随后的例子。

硬件资源

创建多状态开关占用非常少的 CPU 负荷和存储器资源。

可靠性和安全性

使用多状态开关通常没有可靠性或安全性问题。关于开关接口安全性的一般讨论请参考

“开关接口（软件）”和“开关接口（硬件）”两节。

可移植性

这个模式可以毫不费力地改编以用于其他微控制器。

优缺点小结

⑤ 是提高许多系统的可用性的一种高性价比的方法。

相关的模式和替代解决方案

参见“通断开关”。

例子：计数器

在这个例子中，通过一个增加速度取决于开关按下延续时间的计数器来说明多状态开关的主要特性（源程序清单 19.10~源程序清单 19.14）。

所需的硬件如图 19.14 所示。

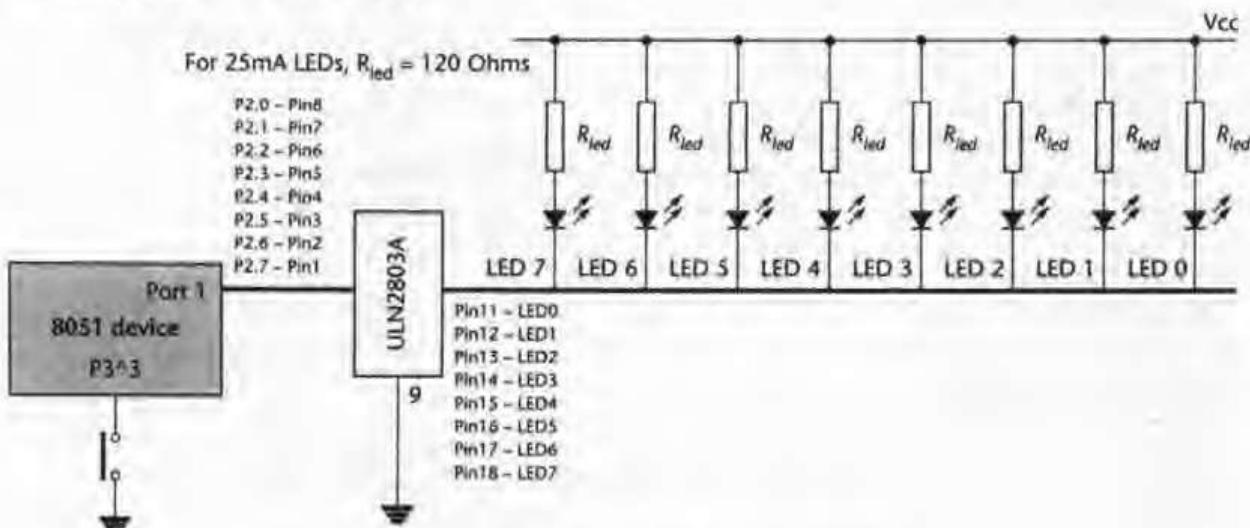


图 19.14 用来说明多状态开关的一组 LED

主要的软件文档在源程序清单 18.6~源程序清单 18.11 中给出，CD 上包含了这个例子的所有源文件。

```

/*
Port.H (v1.00)

// 项目 MULTI_S 的“端口头文件”（参见第 10 章）
*/
// ----- Bargraph.C -----
// +5V 串接适当的电阻连接 LED 到这个引脚

```

```

// [详细资料参见第 7 章]
// 如果需要的话，这 8 个端口引脚可以分布在几个端口上
sbit Pin0 = P2^0;
sbit Pin1 = P2^1;
sbit Pin2 = P2^2;
sbit Pin3 = P2^3;
sbit Pin4 = P2^4;
sbit Pin5 = P2^5;
sbit Pin6 = P2^6;
sbit Pin7 = P2^7;
// ----- Swit_D.C -----
// 连接一个按键开关到这个引脚[到地]
// -软件消抖
sbit Sw_pin = P3^3; // 按下它开始调整时间
/*
----- 文件结束 -----
*/

```

源程序清单 19.10 演示使用多状态开关的程序例子的一部分

```

/*-----*
Main.c (v1.00)

-----*
用于多状态开关模式的演示程序
-详细资料参见第 19 章
要求的链接程序选项（详细资料参见正文）：
OVERLAY (main ~ (SWITCH_MS_Update, COUNTER_Update),
SCH_Dispatch_Tasks ! (SWITCH_MS_Update, COUNTER_Update))
-*-----*-----*/
#include "Main.h"
#include "Swit_D.h"
#include "2_01_12g.H"
#include "Counter.h"
#include "Bargraph.h"
/* ..... */ /* ..... */
void main(void)
{
    // 设置调度器
    SCH_Init_T2();
    // 设置显示
    BARGRAPH_Init();
    // 设置开关引脚
    SWITCH_MS_Init();
    // 添加一个“SWITCH_MS_Update”任务，每隔 200ms
    // -定时单位为时标{50ms 时标间隔——参见 Sch “初始化” 函数}
    SCH_Add_Task(SWITCH_MS_Update, 0, 4);
    // 添加一个“COUNTER_Update”任务，每隔 1000ms
    SCH_Add_Task(COUNTER_Update, 0, 20);
    SCH_Start();
}

```

```
while(1)
{
    SCH_Dispatch_Tasks();
}
/*-----文件结束-----*/

```

源程序清单 19.11 演示使用多状态开关的程序例子的一部分

```
/*-----*
Bargraph.c (v1.00)

简单的柱形图库，参见第 10 章
*-----*/
#include "Main.h"
#include "Port.h"
#include "Bargraph.h"
// -----公有变量声明-----
// 待显示数据
extern tByte Data_G;
// -----私有的常数-----
#define BARGRAPH_ON (1)
#define BARGRAPH_OFF (0)
// -----私有变量-----
// 这些变量存储用来刷新显示的阈值
// 准备显示柱形图
static tBargraph M9_1_G;
static tBargraph M9_2_G;
static tBargraph M9_3_G;
static tBargraph M9_4_G;
static tBargraph M9_5_G;
static tBargraph M9_6_G;
static tBargraph M9_7_G;
static tBargraph M9_8_G;
/*-----*
BARGRAPH_Init()
Prepare for the bargraph display.
*-----*/
void BARGRAPH_Init(void)
{
    Pin0 = BARGRAPH_OFF;
    Pin1 = BARGRAPH_OFF;
    Pin2 = BARGRAPH_OFF;
    Pin3 = BARGRAPH_OFF;
    Pin4 = BARGRAPH_OFF;
    Pin5 = BARGRAPH_OFF;
    Pin6 = BARGRAPH_OFF;
    Pin7 = BARGRAPH_OFF;
```

```

// 使用线性刻度显示数据
// 记住，有“9”种可能的输出状态
// - 一次完成所有计算
M9_1_G = (BARGRAPH_MAX - BARGRAPH_MIN) / 9;
M9_2_G = M9_1_G * 2;
M9_3_G = M9_1_G * 3;
M9_4_G = M9_1_G * 4;
M9_5_G = M9_1_G * 5;
M9_6_G = M9_1_G * 6;
M9_7_G = M9_1_G * 7;
M9_8_G = M9_1_G * 8;
}
/*-----*
BARGRAPH_Update()
刷新柱形图显示
-*-----*/
void BARGRAPH_Update(void)
{
    tBarGraph Data = Data_G - BARGRAPH_MIN;
    Pin0 = ((Data >= M9_1_G) == BARGRAPH_ON);
    Pin1 = ((Data >= M9_2_G) == BARGRAPH_ON);
    Pin2 = ((Data >= M9_3_G) == BARGRAPH_ON);
    Pin3 = ((Data >= M9_4_G) == BARGRAPH_ON);
    Pin4 = ((Data >= M9_5_G) == BARGRAPH_ON);
    Pin5 = ((Data >= M9_6_G) == BARGRAPH_ON);
    Pin6 = ((Data >= M9_7_G) == BARGRAPH_ON);
    Pin7 = ((Data >= M9_8_G) == BARGRAPH_ON);
}
/*-----*
----文件结束-----
-*-----*/

```

源程序清单 19.12 演示使用多状态开关的程序例子的一部分

```

/*-----*
Counter.C (v1.00)

简单的“计数”函数，用来说明多状态开关的使用
-*-----*/
#include "Main.h"
#include "Counter.h"
#include "BarGraph.h"
// -----公有变量定义-----
tBarGraph Data_G;
// -----公有变量声明-----
extern tByte Sw_Status_G;
/*-----*
COUNTER_Update()
简单的计数函数(用于演示)
-*-----*/

```

```

void COUNTER_Update(void)
{
    Data_G += Sw_status_G;
    if (Data_G > BARGRAPH_MAX)
    {
        Data_G = 0;
    }
    BARGRAPH_Update();
}
/*-----*
----文件结束-----
*-----*/

```

源程序清单 19.13 演示使用多状态开关的程序例子的一部分

```

/*-----*
SWIT_D.C (v1.00)
-----*
4 状态开关接口代码，包括软件消除抖动
*-----*/
#include "Main.h"
#include "Port.h"
#include "Swit_D.h"
// -----公有变量-----

tByte Sw_status_G; // 当前的开关状态
// -----私有的常数-----
// 要得到正确的消抖特性，SW_THRES 必须>1
#define SW_THRES (1)
#define SW_THRES_X2 (SW_THRES + SW_THRES + SW_THRES + SW_THRES)
#define SW_THRES_X3 (SW_THRES_X2 + SW_THRES_X2)
// 可以使用常开或常闭开关（或其他接线变化）
#define SW_PRESSED (0)
// -----私有变量-----
static tByte Sw_press_duration_G = 0;
/*-----*
SWITCH_MS_Init()
开关库的初始化函数
*-----*/
void SWITCH_MS_Init(void)
{
    Sw_pin = 1; // 使用这个引脚用于输入
    Sw_status_G = 0; // 开关最初是关闭的
    Sw_press_duration_G = 0;
}
/*-----*
SWITCH_MS_Update()
这是主要的开关函数，应该每隔 50~500ms 调度一次
Sw_press_duration_G 的改变取决于开关按下的延续时间
*-----*/

```

```
void SWITCH_MS_Update(void)
{
    if (Sw_pin == SW_PRESSED)
    {
        Sw_press_duration_G += 1;
        if (Sw_press_duration_G > (SW_THRES_X3))
        {
            Sw_press_duration_G = SW_THRES_X3;
            Sw_status_G = 3; // 开关被按下很长时间...
            return;
        }
        if (Sw_press_duration_G > (SW_THRES_X2))
        {
            Sw_status_G = 2; // 开关被按下中等时间...
            return;
        }
        // SW_THRES 必须>1 用于软件消除抖动
        if (Sw_press_duration_G > SW_THRES)
        {
            Sw_status_G = 1; // 开关被按下较短时间...
            return;
        }
        // 开关被按下, 然而时间不够长
        Sw_status_G = 0;
        return;
    }
    // 开关没有被按下——复位计数
    Sw_press_duration_G = 0;
    Sw_status_G = 0; // 开关没有被按下
}
/*-----*
 *-----文件结束-----*
 */
```

源程序清单 19.14 演示使用多状态开关的程序例子的一部分

进阶阅读

Chapter 20

键盘接口

引言

键盘是嵌入式系统的一种常见模块。

本章将讨论如何创建基于键盘的可靠的用户界面。

键盘接口

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 正在创建该系统的用户界面。

问题

如何将类似图 20.1 所示的小型键盘连接到系统中？

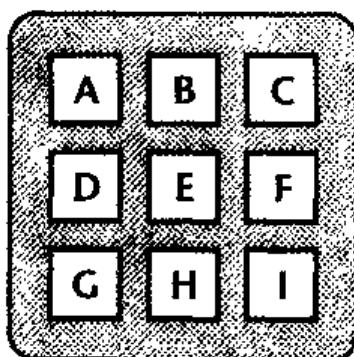


图 20.1 三行 × 三列键盘的一个例子

背景知识

读取单个开关的背景资料参见第 19 章。

解决方案

基础知识

这里关注由矩阵开关组成的键盘，其布置类似于如图 20.2 所示。注意如下所列的一些要点：

- 矩阵布置用来节省端口引脚。如果有 R 行和 C 列按键，如果使用矩阵布置则需要“R + C”个引脚，而如果使用单独的开关则需要“R × C”个引脚。如果需要六个或以上按键，那么矩阵布置将减少对引脚的需要。

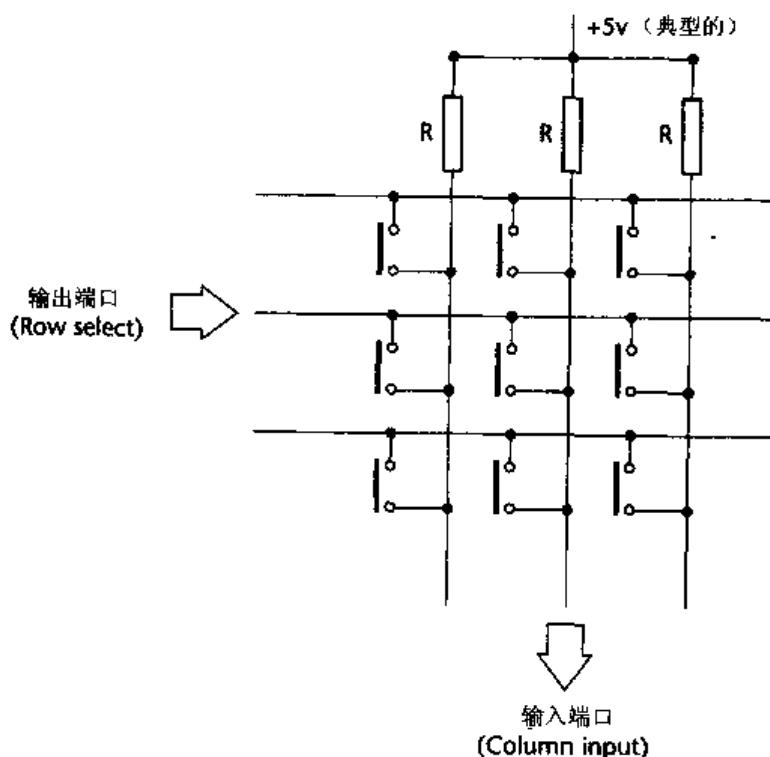


图 20.2 键盘接口的原理图

注意，像往常一样，如果端口有内部上拉，则可以省略上拉电阻。

许多键盘有 12 个按键（0~9 加上两个功能键：一般为“#”和“*”），使用矩阵布置需要七个端口引脚。

- 当被按下和释放时，按键可能抖动。
- 键按下的延续时间通常将至少为 500ms。
- 通常不允许按键“自动重复”，因为那样用户非常容易混淆。

- 可能希望用户按下一个或多个“功能键”与其他按键的组合。

键盘扫描

键盘代码的核心是扫描函数。一般将轮流扫描每一列，并确定是否该列有键被按下。

例如：考虑如图 20.3 所示的键盘。

在图 20.3 中，键盘行和列附近的数字表明了应该连接的端口引脚。这里引脚 0、1 和 2（列）将作为输出引脚。在扫描期间写入这些引脚并输出。引脚 3、4、5 和 6 将作为输入引脚。在扫描期间读取这些引脚。

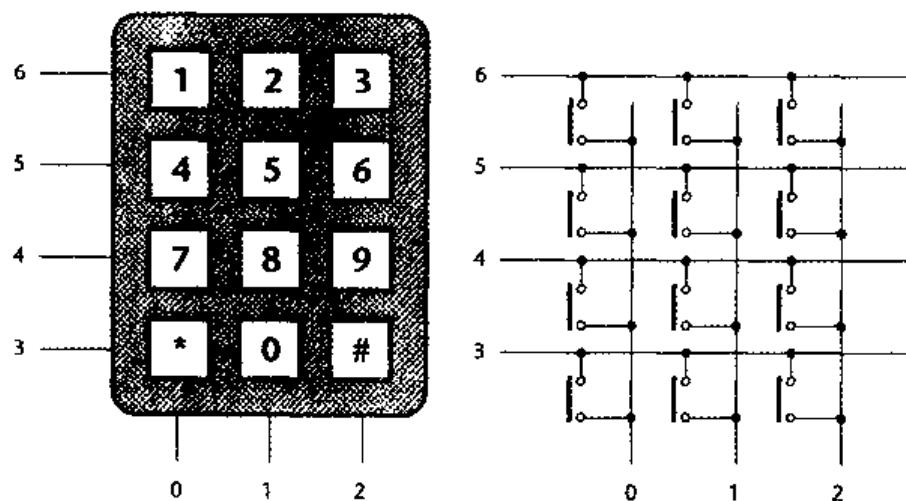


图 20.3 典型的键盘连接

注意，键盘行和列附近的数字表明了应该连接的端口引脚，通常不需要上拉电阻，详细资料参考图 20.2。

假设希望知道按键“V”是否被按下。可以如下进行：

- 设置相应的输出引脚（在这个例子中是引脚 6）为逻辑 0，其余的输出引脚为逻辑 1。
- 读取所需输入的引脚（在这个例子中是引脚 0）。

如果这个引脚为逻辑 1，那么按键“V”没有被按下：逻辑 1 是通过端口引脚的上拉电阻得到的。

如果这个引脚为逻辑 0，那么按键正在被按下（需要考虑抖动）：读取的逻辑 0 电压是由引脚 6 输出的逻辑 0 电压产生的。

- 考虑到开关抖动，必须在（比方说）200ms 之后重复读取。

每个按键都必须重复这个过程。源程序清单 20.1 说明了完成整个键盘扫描的一种方法。

```
#define KEYPAD_PORT P2
sbit C1 = KEYPAD_PORT^0;
sbit C2 = KEYPAD_PORT^1;
sbit C3 = KEYPAD_PORT^2;

sbit R1 = KEYPAD_PORT^6;
sbit R2 = KEYPAD_PORT^5;
```

```

sbit R3 = KEYPAD_PORT^4;
sbit R4 = KEYPAD_PORT^3;
...
bit KEYPAD_Scan(char* const pKey, char* const pFuncKey)
{
    static data char Old_Key;
    char Key = KEYPAD_NO_NEW_DATA;
    char Fn_key = (char) 0x00;
    C1 = 0; // 扫描列 1
    if (R1 == 0) Key = '1';
    if (R2 == 0) Key = '4';
    if (R3 == 0) Key = '7';
    if (R4 == 0) Fn_Key = '*';
    C1 = 1;
    C2 = 0; // 扫描列 2
    if (R1 == 0) Key = '2';
    if (R2 == 0) Key = '5';
    if (R3 == 0) Key = '8';
    if (R4 == 0) Key = '0';
    C2 = 1;
    C3 = 0; // 扫描列 3
    if (R1 == 0) Key = '3';
    if (R2 == 0) Key = '6';
    if (R3 == 0) Key = '9';
    if (R4 == 0) Fn_Key = '#';
    C3 = 1;
    if (Key == KEYPAD_NO_NEW_DATA)
    {
        // 没有按键被按下（或只是功能键）
        Old_Key = KEYPAD_NO_NEW_DATA;
        Last_valid_key_G = KEYPAD_NO_NEW_DATA;
        return 0; // No new data
    }
    // 一个按键被按下。通过检查两次来消抖
    if (Key == Old_Key)
    {
        // 检测到一个有效的（经过消抖）键按下
        // 必须是一个新的按键才有效，不允许“自动重复”
        if (Key != Last_valid_key_G)
        {
            // 新的按键！
            *pKey = Key;
            Last_valid_key_G = Key;
            // 功能键是否也被按下？
            if (Fn_key)

```

```
{  
    // 功能键与另一个按键被同时按下  
    *pFuncKey = Fn_key;  
}  
else  
{  
    *pFuncKey = (char) 0x00;  
}  
return 1;  
}  
}  
// 没有新的数据  
Old_Key = Key;  
return 0;  
}
```

源程序清单 20.1 用于键盘扫描的代码例子

功能键

注意，可能同时按下多个按键。源程序清单 20.1 中的功能键（“#”和“*”）说明了如何使用多键按下的功能。

在这个模式中给出的例子的主代码说明了功能键的使用。

缓冲区设置

在大多数被调度的键盘程序中，使用一个大的缓冲区很有用处。这样，即使系统不能立即处理按键事件，它们也不会丢失。

可以有许多种缓冲区设置。这个模式给出的例子的主代码说明了一种可行的方案。

硬件资源

键盘扫描不需要片内模块（比如定时器等等），但是需要占用 CPU 和存储器开销。

可靠性和安全性

键盘扫描是一种基于软件的技术，与开关接口（软件）紧密相关。在恶劣环境中，使用键盘比使用有硬件接口的多个单独开关[参见开关接口（硬件）]的解决方案的可靠性低。

然而，如果需要较大的键盘，诸如有大约 30 个按键的 QWERTY 键盘时，使用单独的开关可能是不切实际的。在这种情况下，最安全的方法是使用共享时钟调度器（参见第 6 篇），并使用另一个微控制器连接键盘。这个“键盘微控制器”应该使用单独的电源，并且与系统主板隔离。

可移植性

代码非常容易移植，并且这种键盘扫描方法的使用非常广泛。

优缺点小结

- ◎ 多路复用键盘容易使用并且便宜。
- ◎ 因为模式基于软件，所以对 ESD 和恶意损坏的保护措施很少。如果必须保证主处理器的可靠性，则可以使用一个单独、隔离的键盘处理器。
- ◎ 在恶劣环境中，避免使用多路复用键盘而使用单独的开关将更可靠。

相关的模式和替代解决方案

参见开关接口（软件）及开关接口（硬件）。

例子：支持功能键和缓冲区的键盘库

这里给出用于 8051 系列的完整的键盘库。该库支持两个功能键和缓冲区功能。该库用来处理如图 20.3 中所示的键盘，但是可以很容易地修改以适以用于不同的键盘布局和大小。

演示程序运行在 Infineon 的 c515c 微控制器上。然而，所有键盘代码都不只是针对 515 的。只要使用合适版本的调度器和 PC 连接库（包含在 CD 上），这个键盘库就能被用于任何 8051 系列芯片。

图 20.4 显示了典型的程序输出

源程序清单 20.2~源程序清单 20.5 中给出了主要的库文件，这个例子的所有源文件参见 CD。

```
Serial interface initialized.
Keypad test code = READY
1234567890
*1
*1
#2
#5
09823741122
```

图 20.4 本节给出的键盘输入程序的典型输出

```
/*
-----*
Port.H (v1.00)
-----*/
// 项目 Key_232 的“端口头文件”（参见第 10 章）
/*
-----*----- Sch51.C -----*/
//
```

```

// 如果不需要错误报告, 将这一行注释掉
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1
#endif
// ----- Keypad.C -----
#define KEYPAD_PORT P5
sbit C1 = KEYPAD_PORT^0;
sbit C2 = KEYPAD_PORT^1;
sbit C3 = KEYPAD_PORT^2;
sbit R1 = KEYPAD_PORT^6;
sbit R2 = KEYPAD_PORT^5;
sbit R3 = KEYPAD_PORT^4;
sbit R4 = KEYPAD_PORT^3;
// ----- Lnk_O.C -----
// 引脚 3.0 和引脚 3.1 用于 RS-232 标准接口
/*
----- 文件结束 -----
*/

```

源程序清单 20.2 用于键盘接口软件的一部分

```

/*
Main.c (v1.00)

用于键盘库的测试程序, 基于 c515c
在串行 (RS232) 连接上发送输出 (到 PC)
链接程序选项:
OVERLAY (main ~ (PC_LINK_Update, Keypad_RS232_Update),
SCH_Dispatch_Tasks ! (PC_LINK_Update, Keypad_RS232_Update))
*/
#include "Main.h"
#include "2_01_10i.h"
#include "B_O_lnk.h"
#include "Keypad.h"
#include "Keyp_232.h"
/* ..... */
/* ..... */
void main(void)
{
    //
    // 设置调度器
    SCH_Init_T2();
    // 波特率设置为 9600, 使用内部波特率发生器
    PC_LINK_Init_Internal(9600);
    // 准备键盘
    KEYPAD_Init();
    // 准备键盘->RS232 库
    Keypad_RS232_Init();
}

```

```

// 必须调度这个任务（这里 100x 每秒就已经足够）
// -这里向 PC 写数据
//
// 定时单位为时标 (1ms 间隔)
SCH_Add_Task(PC_LINK_Update, 10, 10);
// 每隔 50ms 读取键盘
SCH_Add_Task(Keypad_RS232_Update, 0, 50);
SCH_Start();
while(1)
{
    SCH_Dispatch_Tasks();
}
/*
-----文件结束-----
*/

```

源程序清单 20.3 用于键盘接口软件的一部分

```

/*
Keypad.C (v1.00)

-----简单的键盘库，用于 3 列 x4 行键盘。
按键布置为：
Key arrangement is: -----
      1  2  3
      4  5  6
      7  8  9
      *  0  #

-----支持两个功能键（“*”和“#”），详细资料参见第 19 章相关内容
*/
#include "Main.h"
#include "Port.h"
#include "Keypad.h"
// -----私有函数原型-----
bit KEYPAD_Scan(char* const, char* const);
// -----私有的常数-----
#define KEYPAD_RECV_BUFFER_LENGTH 6
// 任何有效的字符即可，不能和键盘上的按键相同
#define KEYPAD_NO_NEW_DATA (char) '-'
// -----私有变量-----
static char KEYPAD_recv_buffer[KEYPAD_RECV_BUFFER_LENGTH+1][2];
static tByte KEYPAD_in_read_index; // 已经读取的数据输入缓冲区
static tByte KEYPAD_in_waiting_index; // 还没有读取的数据输入缓冲区
static char Last_valid_key_G = KEYPAD_NO_NEW_DATA;
/*
KEYPAD_Init()
初始化键盘
*/

```

```
void KEYPAD_Init(void)
{
    KEYPAD_in_read_index = 0;
    KEYPAD_in_waiting_index = 0;
}
/*-----*/
KEYPAD_Update()
用于键盘库的主要的“刷新”函数
必须调度这个函数(大约每隔50~200ms)
-*-----*/
void KEYPAD_Update(void)
{
    char Key, FnKey;
    // 这里扫描键盘...
    if (KEYPAD_Scan(&Key, &FnKey) == 0)
    {
        // 没有新的按键数据-返回
        return;
    }
    // 如果旧的数据已经读取, 希望读取指针0
    // (简单的循环缓冲区)
    if (KEYPAD_in_waiting_index == KEYPAD_in_read_index)
    {
        KEYPAD_in_waiting_index = 0;
        KEYPAD_in_read_index = 0;
    }
    // 加载键盘数据到缓冲区
    KEYPAD_recv_buffer[KEYPAD_in_waiting_index][0] = Key;
    KEYPAD_recv_buffer[KEYPAD_in_waiting_index][1] = FnKey;
    if (KEYPAD_in_waiting_index < KEYPAD_RECV_BUFFER_LENGTH)
    {
        // 加1且缓冲区不溢出
        KEYPAD_in_waiting_index++;
    }
}
/*-----*/
KEYPAD_Get_Char_From_Buffer()
刷新函数将数据复制到键盘缓冲区这里从缓冲区读取数据
-*-----*/
bit KEYPAD_Get_Data_From_Buffer(char* const pKey, char* const pFuncKey)
{
    // 如果缓冲区中有新数据
    if (KEYPAD_in_read_index < KEYPAD_in_waiting_index)
    {
        *pKey = KEYPAD_recv_buffer[KEYPAD_in_read_index][0];
        *pFuncKey = KEYPAD_recv_buffer[KEYPAD_in_read_index][1];
        KEYPAD_in_read_index++;
        return 1;
    }
}
```

```

    return 0;
}
/*-----*
 * Function KEYPAD_Clear_Buffer()
 *-----*/
void KEYPAD_Clear_Buffer(void)
{
    KEYPAD_in_waiting_index = 0;
    KEYPAD_in_read_index = 0;
}
/*-----*
 * KEYPAD_Scan()
 *-----*/
这个函数由被调度的键盘函数调用，必须根据需要修改以匹配按键标签
包含两个“功能键”可以与其他列的任何按键同时使用
根据需要修改！
/*-----*/
bit KEYPAD_Scan(char* const pKey, char* const pFuncKey)
{
    static data char Old_Key;
    char Key = KEYPAD_NO_NEW_DATA;
    char Fn_key = (char) 0x00;
    C1 = 0; // 扫描列 1
    if (R1 == 0) Key = '1';
    if (R2 == 0) Key = '4';
    if (R3 == 0) Key = '7';
    if (R4 == 0) Fn_key = '*';
    C1 = 1;
    C2 = 0; // 扫描列 2
    if (R1 == 0) Key = '2';
    if (R2 == 0) Key = '5';
    if (R3 == 0) Key = '8';
    if (R4 == 0) Key = '0';
    C2 = 1;
    C3 = 0; // 扫描列 3
    if (R1 == 0) Key = '3';
    if (R2 == 0) Key = '6';
    if (R3 == 0) Key = '9';
    if (R4 == 0) Fn_key = '#';
    C3 = 1;
    if (Key == KEYPAD_NO_NEW_DATA)
    {
        // 没有按键被按下（或只是功能键）
        Old_Key = KEYPAD_NO_NEW_DATA;
        Last_valid_key_G = KEYPAD_NO_NEW_DATA;
        return 0; // 没有新的数据
    }
    // 一个按键被按下，通过检查两次来消抖
    if (Key == Old_Key)
    {

```

```

// 检测到一个有效的（经过抖动）键按下

// 必须是一个新的按键才有效，不允许“自动重复”
if (Key != Last_valid_key_G)
{
    //
    // 新的按键！
    *pKey = Key;
    Last_valid_key_G = Key;
    // 功能键是否也被按下？
    if (Fn_key)
    {
        //
        // 功能键与另一个按键被同时按下
        *pFuncKey = Fn_key;
    }
    else
    {
        *
        *pFuncKey = (char) 0x00;
    }
    return 1;
}
}

// 没有新的数据
Old_Key = Key;
return 0;
}

/*
-----文件结束-----
*/

```

源程序清单 20.4 用于键盘接口软件的一部分

```

/*
Keyp_232.C (v1.00)

通过串行 (RS232) 连接将键盘输入传送到 PC 上的简单的演示函数
*/
#include "Main.h"
#include "Keypad.h"
#include "Keyp_232.h"
#include "lnk_O.h"
// -----私有变量-----
tByte Count_G;
/*
Keypad_RS232_Init()
通过串行连接显示键盘输入的简单的库的初始化函数
*/
void Keypad_RS232_Init(void)
{
    PC_LINK_Write_String_To_Buffer("Keypad test code - READY\n");
    Count_G = 0;
}

```

```

    KEYPAD_Clear_Buffer();
}
/*-----*/
Keypad_RS232_Update()
通过串行连接显示键盘输入的函数
-*-
void Keypad_RS232_Update(void)
{
    char Key, FnKey;
    // 刷新键盘缓冲区
    KEYPAD_Update();
    // 键盘缓冲区中是否有新数据?
    if (KEYPAD_Get_Data_From_Buffer(&Key, &FnKey) == 0)
    {
        // 没有新的数据
        return;
    }
    // 功能键与另一个按键被同时按下
    if (FnKey)
    {
        PC_LINK_Write_Char_To_Buffer('\n');
        PC_LINK_Write_Char_To_Buffer(FnKey);
        PC_LINK_Write_Char_To_Buffer(Key);
        PC_LINK_Write_Char_To_Buffer('\n');
        Count_G = 0;
    }
    else
    {
        // 一个普通的按键(不是功能键)被按下
        PC_LINK_Write_Char_To_Buffer(Key);
        if (++Count_G == 10)
        {
            PC_LINK_Write_Char_To_Buffer('\n');
            Count_G = 0;
        }
    }
}
/*-----*/
----文件结束-----
-*-

```

源程序清单 20.5 用于键盘接口软件的一部分

例子：键盘和 LCD

键盘和 LCD 接口设计的完整例子参见第 22 章。

进阶阅读

Chapter 21

多路复用 LED 显示

引言

许多嵌入式系统都包含了由多段 LED 显示组成的用户界面。在本章中，讨论如何将这种显示连接到 8051 系列微控制器上。

多路复用 LED 显示

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 正在创建该系统的用户界面。

问题

如何在一个或多个多段 LED 上显示信息？

背景知识

本节介绍一些必要的背景资料。

什么是多段 LED？

多个 LED 往往排列成多段显示，八段显示组合（参见图 21.1）和类似的七段显示组合（去掉小数点）特别常见。

这样的显示排列为“共阴极”或“共阳极”封装，各种封装的 LED 连接如图 21.2 所示。

这两种情况下，除公共引脚之外，必须为每个 LED 提供合适的信号，以产生需要的数字（参见“解决方案”）。

每个段需要的电流从 2mA（非常小的显示）到大约 60mA（非常大的显示，100mm 以上）。

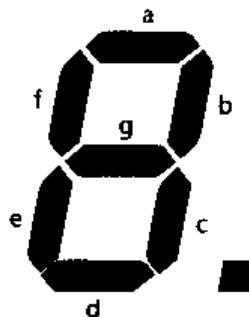


图 21.1 通常称为七段显示的分段

注意，正如这里所示，常常还包含一个小数点，这样，“七段”显示实际上有八个段。

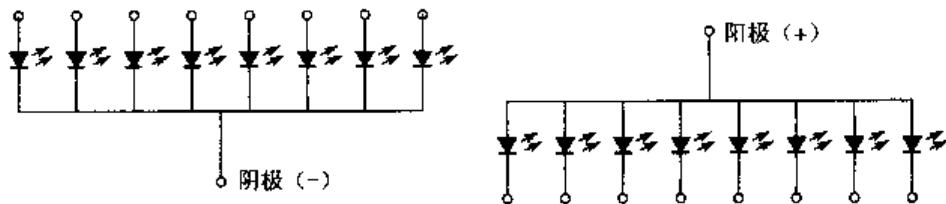


图 21.2 共阴极和共阳极 LED 显示的内部排列

注意，有八个段，因为假定有一个小数点。

基本的硬件要求（驱动一位数字）

由于在 IC 缓冲器中讨论的理由，通常无法直接如图 21.3 所示将多段 LED 连接到微控制器的端口上。这是因为端口无法可靠地提供所需的电流（一般每个数字大约需要 80mA）。

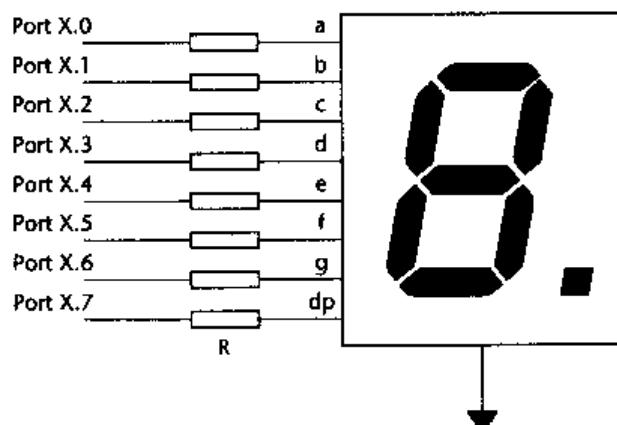


图 21.3 试图通过微控制器端口直接驱动一个共阳极显示

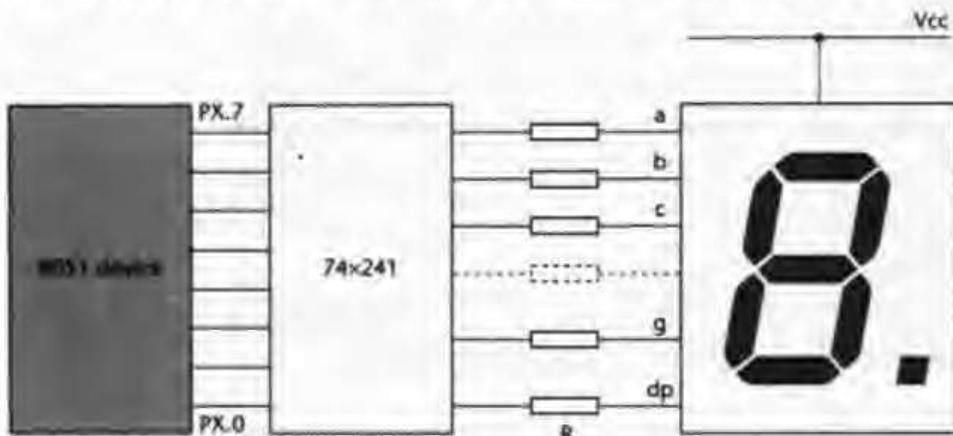
注意，在大多数情况下将无法工作。

在大多数情况下，在端口和多段 LED 之间需要某种形式的缓冲或驱动芯片。

对于小的显示（每段大约 9mA），简单的 240 和 241 缓冲器（参见 IC 缓冲器）是一种高性价比的解决方案。图 21.4 显示了使用 74x241（正相缓冲器）来驱动器一个小的八段 LED 显

示。注意，可以使用74x240（反相）缓冲器的替代方案。

在大多数情况下（比如这里关注的多路复用显示），基本的240/241缓冲器不能提供足够的电流，通常需要使用IC驱动芯片。例如，图21.5显示了通过八缓冲器UDN2585A将一个共阴极的LED数字连接到端口上。正如在IC缓冲器中讨论的，这个器件的每个通道都可以同时提供高达120mA的电流（最高25V），即使对于大的LED显示这也足够了。



$$R = \frac{V_{cc} - V_{diode}}{I_{diode}}$$

注意，电阻值可能不相同（小数点可能需要不同的电阻值）

图21.4 使用74x241来驱动多段LED显示

注意，在一些显示中，可能要求不同的电阻值来驱动小数点。具体需要核对数据手册。同时还要注意，正如在IC缓冲器中讨论的，如果使用CMOS逻辑，不需要在缓冲输出端添加上拉电阻。

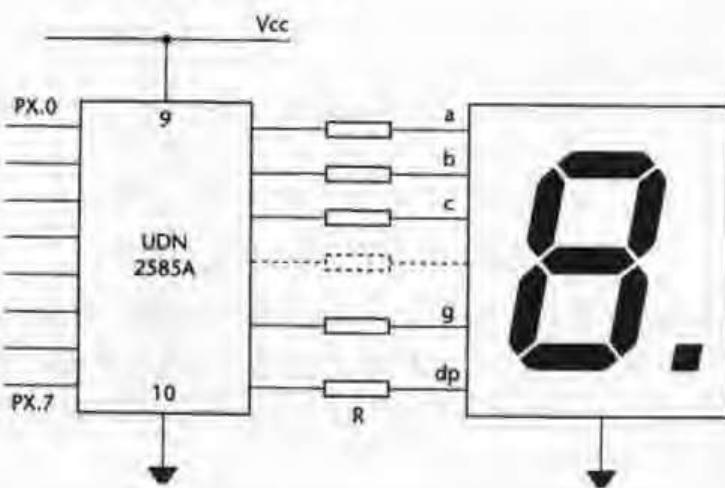


图21.5 使用UDN2585A来驱动LED显示

注意，这是一种反相（电流源）缓冲器，输入为逻辑0时将点亮相应的LED段。

这里，对应逻辑0，2585的输出大约为Vcc。因此，如图21.4所示再次计算电阻值。

多个数字的硬件

正如在“背景知识”中讨论的，驱动一个多段显示很简单。然而，如果需要驱动四位数字，例如，用于显示一个简单的数字时钟。使用在“背景知识”中讨论的方法将需要四个空闲的端口，但在许多系统中并没有四个端口可用。此外，使用四个端口往往使得每个数字都需要独立的驱动（缓冲）电路和排阻，这将显著增加系统的成本。

这里讨论的最常见的解决方案是采用多路复用显示。这指的是（在这个例子中）对于每个显示只驱动四分之一的时间。只要在 20~50Hz 之间循环所有的显示，用户通常便不会注意到显示并不是同时被驱动的。即使在简单的实现中，也可以采用这种基本的方法实现仅用两个端口来驱动最多八个 LED（每个 LED 都含小数点）。更普通的是使用 12 个端口引脚来驱动四个 LED 数字。在许多 8051 系统中，即使使用外部存储器，端口 1 和端口 3 的大多数引脚也是可用的。因此这种方法可用于许多系统。

用于创建高性价比的多路复用显示的一种简单的电路如图 21.6 所示。这里，端口 X 提供驱动每个数字所需的信息，通过端口 Y 选择受控的数字。

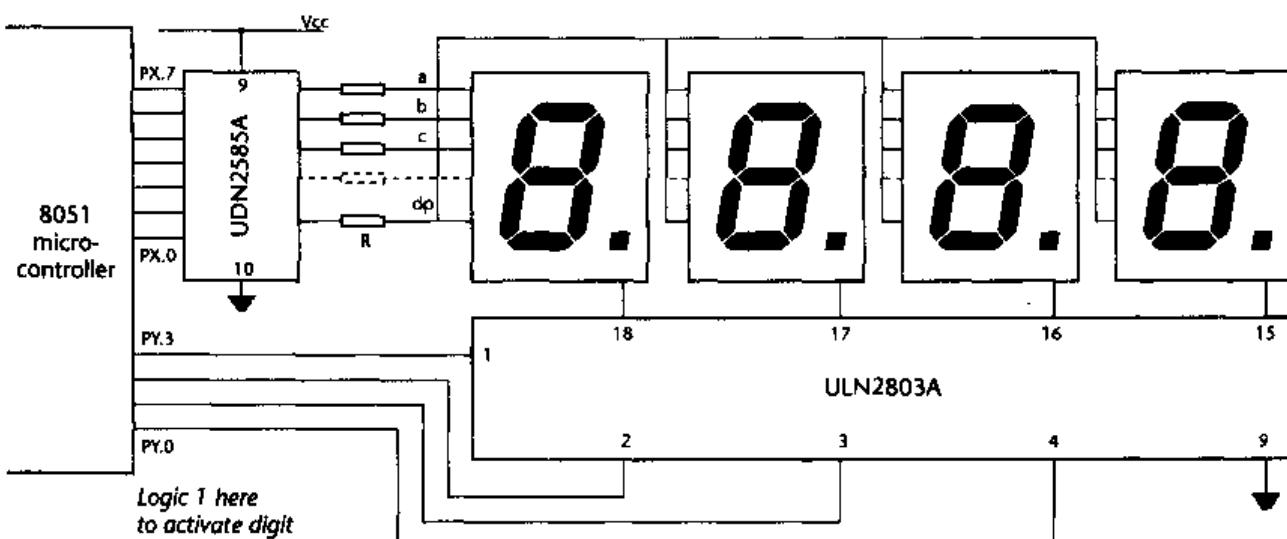


图 21.6 多路复用四（共阴极）LED 模块

注意，端口 Y 必须控制（灌入）来自八个单独 LED 的电流。即使小的显示也有可能高达 140mA，而对于大的（100 毫米）显示可能接近 500mA。因此，不能从端口直接驱动，必须使用 ULN2003 或 2803 等作为（灌入）缓冲器。注意，可以用四个 NPN 型晶体管（例如 2N2222）替换 2003/2803，但其性价比不高。

同样，端口 X 也需要由缓冲器提供足够的电流。在这里，UDN 258SA 是一种较好的选择。使用这样的电流源缓冲往往是必要的。因为为了获得高亮度的显示，通常在四分之一的时间里提供的电流高达正常显示电流的四倍（从而使平均电流达到要求值）。如果试图直接从微控制器芯片提供这样的电流将极大地降低芯片的寿命。

同样，图 21.6 中的电阻值的计算结果如图 21.6 中所示。在这个例子中，使用“排阻”来实现该电路是有益的。这种排阻在标准（双列直插或类似的）封装中包含七或八个电阻。

解决方案

这里讨论显示数字所需的软件代码以及实现无闪烁显示所需的刷新速率。

基本软件

编写软件来控制图 21.4 和图 21.5 所示排列的 LED 数字很容易。例如，把显示连接到端口 3 上，可以通过简单地写入以下代码来控制它：

```
P3 = code;
```

在这里，“code”是需要点亮的段的数值表示。显示附带小数点或无小数点的数字（0~9）所需的编码在源程序清单 21.1 中给出。

```
/*
Connections 连接
DP   G   F   E   D   C   B   A   =   LED 显示引脚
|   |   |   |   |   |   |   |   =
x.7 x.6 x.5 x.4 x.3 x.2 x.1 x.0   =   端口引脚
LED 编码 (NB-这里假定正逻辑)
0  = abcdef => 00111111 = 0x3F
1  = bc     => 00000110 = 0x06
2  = abdeg  => 01011011 = 0x5B
3  = abcdg  => 01001111 = 0x4F
4  = bcdgf  => 01100110 = 0x66
5  = acdfg  => 01101101 = 0x6D
6  = acdefg => 01111101 = 0x7D
7  = abc     => 00000111 = 0x07
8  = abcdefg => 01111111 = 0x7F
9  = abcdgf => 01101111 = 0x6F
以上数值加 10 (十进制) 将显示小数点
*/
// 查找表格-保存在代码区 (ROM)
tByte code LED_Table_G[20] =
// 0      1      2      3      4      5      6      7      8      9
{0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f,
 // 0      1      2      3      4      5      6      7      8      9
 0xBF, 0x86, 0xDB, 0xCF, 0xE6, 0xED, 0xFD, 0x87, 0xFF, 0xEF};
//
```

源程序清单 21.1 用来在多段 LED 上显示数据的编码

每个端口控制多个数字（软件问题）

如果可能的话，需要以大约 50Hz 的速度刷新所有显示。必要的话，低至 20Hz 的刷新速

度也可以工作。如果有 N 个 LED 模块，则每个 LED 将被激活大约 $1/N$ 时间。注意，如果决
~~定使用大电流驱动显示 定时必须相当精确 不则会损坏加士士的电源板可能降低且二和而此~~

的端口有限，这将是一种有用的解决方案。

另一种降低对端口要求的方法是使用 74x247 BCD 到 7 段译码器芯片来连接显示，如图 21.7 所示。这种方法主要的问题是 74x247 已不再被广泛使用。这种芯片很容易使用，但是必须能够保证未来的供货。详细资料请查阅其数据手册。

如果缺少端口是一个问题，那么另一个替代方案是在设计中使用另一个精简 8051。这个芯片将负责多路复用显示以及刷新。使用串行接口和共享时钟调度器可以很容易地向该芯片传送数据（如图 21.8 所示）。共享时钟调度器在第 6 篇中讨论。

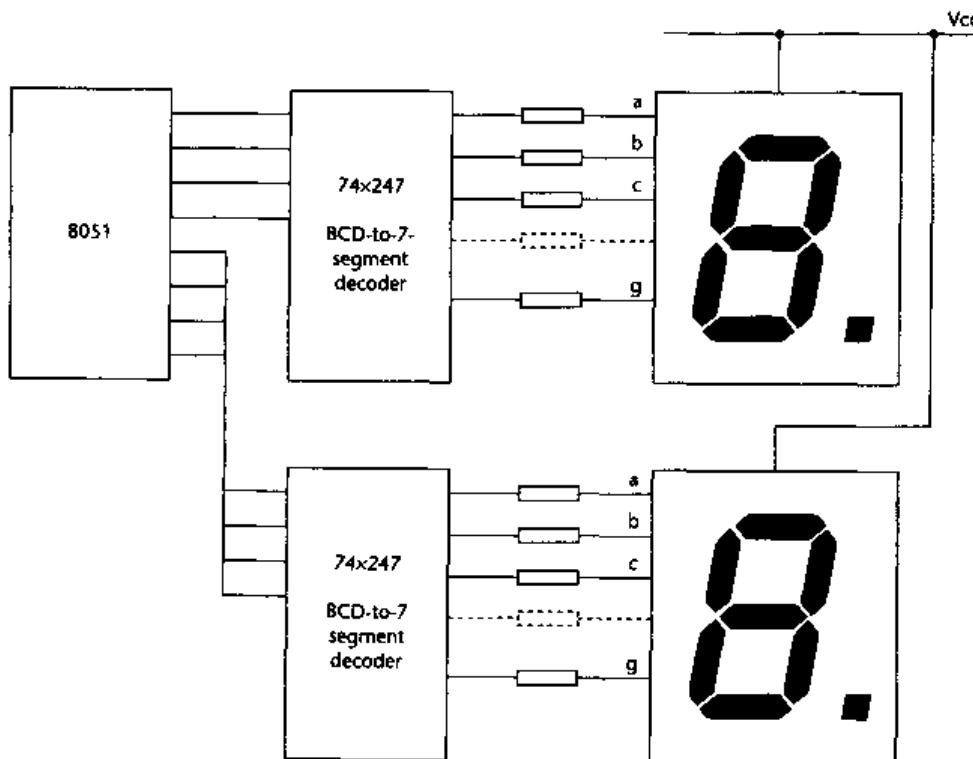


图 21.7 使用 74x247 BCD 译码器从一个端口驱动两个多段 LED 显示

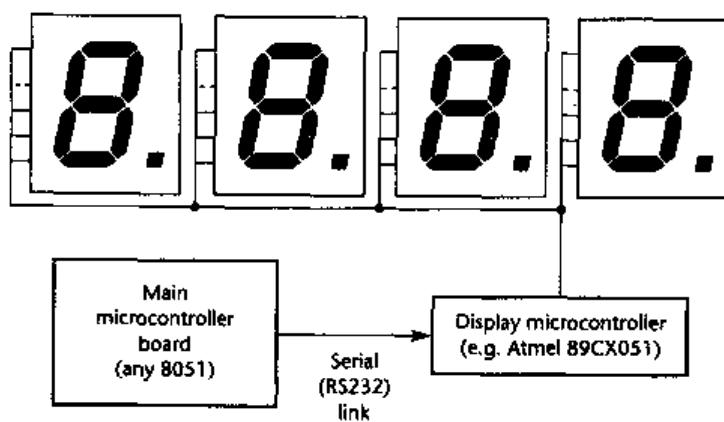


图 21.8 使用共享时钟调度器和两个 8051 微控制器来提供迅速的 LED 显示刷新

例子：在多路复用（四个数字）LED 上显示消逝的时间

LED 显示的一种常见用途是用来显示时间。这个简单的例子展示了在四个多路复用的 LED 上显示消逝的时间，该程序是针对标准 8051（8052）芯片编写的。

所需的硬件如图 21.6 所示。

这个例子的主要源文件如下（源程序清单 21.2~源程序清单 21.5）所示，所有的文件都包含在 CD 上。

```
/*-----*
Port.H (v1.00)

-----*
项目 LED_TIME 的“端口头文件”（参见第 10 章）
-----*/
// ----- Sch51.C -----
// 如果不需要错误报告，将这一行注释掉
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1
#endif
// ----- LED_MX4.C -----
// LED 连接需要 12 个端口引脚
#define LED_DATA_PORT (P2)
/* 连接到 LED_DATA_PORT-详细资料参见图 21.6
   DP   G   F   E   D   C   B   A   =   LED 显示引脚
   |   |   |   |   |   |   |   |   |
x.7 x.6 x.5 x.4 x.3 x.2 x.1 x.0   =   端口引脚
x.7 == LED_DATA_PORT^7, etc
LED 编码 (NB-这里假定正逻辑)
0  = abcdef  => 00111111 = 0x3F
1  = bc      => 00000110 = 0x06
2  = abdeg   => 01011011 = 0x5B
3  = abcdg   => 01001111 = 0x4F
4  = bcfg    => 01100110 = 0x66
5  = acdfg   => 01101101 = 0x6D
6  = acdefg  => 01111101 = 0x7D
7  = abc      => 00000111 = 0x07
8  = abcdefg => 01111111 = 0x7F
9  = abcdgf  => 01101111 = 0x6F
以上数值加 10 (十进制) 将显示小数点
// 这里可以使用任何端口的任何 4 个引脚组合
sbit LED_DIGIT_0 = P3^3;
sbit LED_DIGIT_1 = P3^4;
```

```
sbit LED_DIGIT_2 = P3^5;
sbit LED_DIGIT_3 = P3^6;
/*-----*
 *-----文件结束-----*
 */
```

源程序清单 21.2 在四个多路复用 LED 上显示所消逝时间的例子的一部分

```
/*-----*
Main.c (v1.00)

Demonstration program for:
用于驱动四个多路复用的多段 LED 来显示消逝时间的演示程序
要求的链接程序选项（详细资料参见第 13 章）：
OVERLAY

(main ~ (CLOCK_LED_Time_Update,LED_MX4_Display_Update),
SCH_dispatch_tasks !(CLOCK_LED_Time_Update,LED_MX4_Display_Update))
*-----*/
#include "Main.h"
#include "2_01_12g.h"
#include "LED_Mx4.h"
#include "Cloc_Mx4.h"
/* ..... */
/* ..... */
void main(void)
{
    // 设置调度器
    SCH_Init_T2();
    // 添加“Time Update”任务（每秒一次）
    // - 定时单位为时标 (1ms 间隔)
    // (最大的间隔 / 延迟是 65535 个 ticks 时标)
    SCH_Add_Task(CLOCK_LED_Time_Update,100,10);
    // 添加“Display Update”任务（每秒一次）
    // 必须每隔（大约）3ms 刷新 4-段显示
    // 必须每隔（大约）6ms 刷新 2-段显示
    SCH_Add_Task(LED_MX4_Display_Update,0,3);
    // 开始调度器
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

-----文件结束-----
```

源程序清单 21.3 在四个多路复用 LED 上显示所读取时间的例子的一部分

```
/*
Cloc_Mx4.C (v1.00)

用于跟踪消逝时间的简单的库函数
这个版本用于(多路复用) LED 显示
*/
#include "Main.h"
#include "Port.h"
#include "Cloc_Mx4.h"
// -----公有变量声明-----
extern tByte LED_Mx4_Data_G[4];
extern tByte code LED_Table_G[20];
// -----私有变量定义-----
// 时间变量
static tByte Hou_G, Min_G, Sec_G;
/*
CLOCK_LED_Time_Update()
刷新全局时间变量。
***必须每秒调度一次***
*/
void CLOCK_LED_Time_Update(void)
{
    bit Min_update = 0;
    bit Hou_update = 0;
    if (++Sec_G == 60)
    {
        Sec_G = 0;
        Min_update = 1;

        if (++Min_G == 60)
        {
            Min_G = 0;
            Hou_update = 1;

            if (++Hou_G == 24)
            {
                Hou_G = 0;
            }
        }
    }
    if (Min_update)
    {
        // 需要刷新分钟数据
        // (两个数字)
        LED_Mx4_Data_G[1] = LED_Table_G[Min_G / 10];
        LED_Mx4_Data_G[0] = LED_Table_G[Min_G % 10];
    }
    // 在这个版本中不显示秒
    // 使用秒数据来打开和关闭小时和分钟之间的小数点
```

第 21 章 多路复用 LED 显示

```
if ((Sec_G % 2) == 0)
{
    LED_Mx4_Data_G[2] = LED_Table_G[Hou_G % 10];
}
else
{
    LED_Mx4_Data_G[2] = LED_Table_G[(Hou_G % 10) + 10];
}

if (Hou_update)
{
    // 需要刷新“小时的高十位”数据
    LED_Mx4_Data_G[3] = LED_Table_G[Hou_G / 10];
}
}

/*
-----文件结束-----
*/

```

源程序清单 21.4 在四个多路复用 LED 上显示所流逝时间的例子的一部分

```
/*
-----*
LED_Mx4.C (v1.00)

-----*
用于在四个多路复用的八段 LED 上显示数据的简单的库函数
-----*/
#include "Main.h"
#include "Port.h"
#include "LED_Mx4.h"
// -----公有变量定义-----
// 查找表格-保存在代码区
// 连接和程序的详细资料参见 Port.H
tByte code LED_Table_G[20] =
// 0   1   2   3   4   5   6   7   8   9
{0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F,
// 0.   1.   2.   3.   4.   5.   6.   7.   8.   9.
0xBF, 0x86, 0xDB, 0xCF, 0xE6, 0xED, 0xFD, 0x87, 0xFF, 0xEF};
// 用于显示的格式化的全局数据，初始化为 0, 0, 0, 0)
tByte LED_Mx4_Data_G[4] = {0x3F, 0x3F, 0x3F, 0x3F};

// -----私有变量定义-----
static tByte Digit_G;
/*
-----*
LED_MX4_Display_Update()
刷新(四个)多路复用 8-段 LED 显示。
通常要求间隔大约 3ms 调度。在基本的 8051 上，一般占用大约 1% 的 CPU 开销
-----*/
void LED_MX4_Display_Update(void)
{
    // 增加待显示的数字
    if (++Digit_G == LED_NUM_DIGITS)
```

```

{
Digit_G = 0;
}
// 可以使用任何引脚
switch (Digit_G)
{
case 0:
{
LED_DIGIT_0 = 0;
LED_DIGIT_1 = 0;
LED_DIGIT_2 = 0;
LED_DIGIT_3 = 1;
break;
}
case 1:
{
LED_DIGIT_0 = 0;
LED_DIGIT_1 = 0;
LED_DIGIT_2 = 1;
LED_DIGIT_3 = 0;
break;
}
case 2:
{
LED_DIGIT_0 = 0;
LED_DIGIT_1 = 1;
LED_DIGIT_2 = 0;
LED_DIGIT_3 = 0;
break;
}
case 3:
{
LED_DIGIT_0 = 1;
LED_DIGIT_1 = 0;
LED_DIGIT_2 = 0;
LED_DIGIT_3 = 0;
}
}
LED_DATA_PORT = 255 - LED_Mx4_Data_G[Digit_G];
}
/*
----文件结束-----
*/

```

源程序清单 21.5 在四个多路复用 LED 上显示所消耗时间的一部分

从 1 到 3.2 秒之间

进阶阅读

控制 LCD 显示面板

引言

在第 21 章中讨论了创建基于 LED 的用户界面，这里讨论使用液晶显示（LCD）的用户界面。

与 LED 不同，LCD 基于无源显示技术。这指的是 LCD 控制光线的通道而不是发出光线。这使得这些芯片的功耗很低：大的（5V）面板需要最高 5mA 的电流。除背光以外，总功耗最多为 25 毫瓦。小的面板的功耗约为一半，与一个 LED 的功耗类似。所以在电池供电的嵌入式系统中使用 LCD 显示非常普遍。

LCD 显示有各种类型，它们大体上可以分为两类：图形显示和文本显示。笔记本计算机（以及越来越多的桌面计算机）使用由成千上万独立的像素组成的很复杂的图形显示。这样的显示非常昂贵，并且在运行时通常需要大量的存储器（一般为几兆字节）和强大的处理器。正如已经看到的，本书中涉及的嵌入式设备通常无法承受这种费用。这里将主要关注用来显示文字的小型显示。

有多种基于 LCD 的字符显示可供选择。一般排列为一、二或四行，每行 16~40 个字符。毫无疑问，大的显示更昂贵并且功耗更高。每个 LCD 字符通常为 5×8 的点阵，也有较少的一些 LCD 字符使用 5×11 的点阵。注意，在所有情况下，字符本身的大小为 5×7 或 5×10 像素，最下面一行留给光标使用。这样的显示应用在大多数嵌入式处理器上时，产生字符将耗费 CPU 大部分的时间（以及大部分端口或地址空间）。因此，大多数 LCD 显示面板包含片内控制器来处理这些，这通常是 Hitachi 的 HD44780 的一种变型。所有基于这种通用控制器的显示都有非常类似的硬件接口，并且将显示同样的英文（或日文）字符。

在本章中给出的 LCD 字符面板模式中，集中讨论基于这种“标准”控制器的 LCD 显示面板。

请注意，这里给出的大部分背景资料是由 Hitachi 的 HD44780 的数据手册改编而来的。

字符型 LCD 面板

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 正在创建该系统的用户界面。

问题

如何将基于 LCD 的字符面板显示连接到嵌入式处理器，并且使用高级编程语言有效地控制它？

背景知识

正如在本章开始时所指出的，这里关注基于 HD44780 微控制器的 LCD 字符面板。

HD44780 的主要模块

HD44780 包含其自己的复位电路、存储器等等。它包含了几个重要的模块（如图 22.1 所示），这些模块可以由相连的微控制器或微处理器访问控制。将在下述段落中讨论这些主要模块。

如图 22.1 中所示，微控制器和 HD44780 之间的接口由五组信号组成。在表 22.1 中给出了对这些信号的概要介绍。

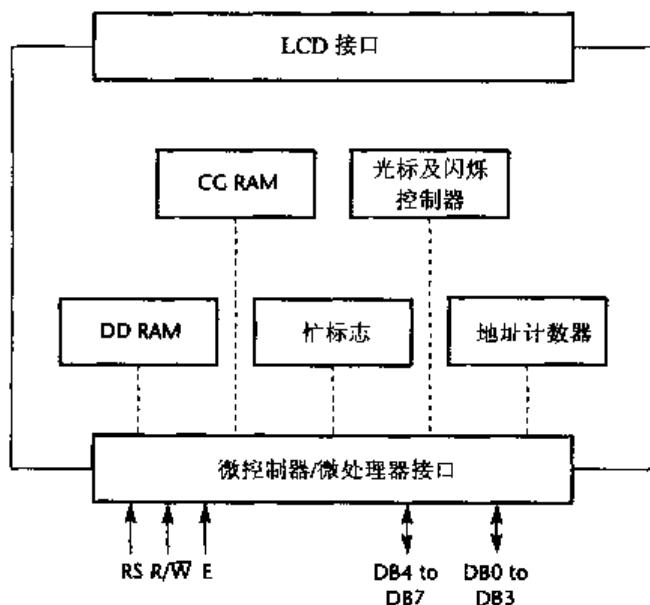


图 22.1 Hitachi 的 HD44780 的主要模块

DD RAM

HD44780 的核心是 DD RAM：“显示数据” RAM，用来以 8 位字符编码的形式存储显示数据，其容量是 80 个字符。因此最大的显示容量是 20 字符 x4 行或 40 字符 x2 行。

HD44780 的使用与（通过 4 位或 8 位数据总线）向 DD RAM 传送数据有关，HD44780 将使用这些数据按要求刷新显示。

通过 DD RAM 存储和显示的字符包含 ASCII 表中的大部分可显示字符，只做了很少的改变（如表 22.2 所示）。这意味着，可以向显示直接发送字符数据，并获得要求的输出。

注意，这些字符都不使用显示的最下面一行（光标线）。因此，带“尾巴”的小写字符（如，g、j、p、q 及 y）的显示将比正常的字符显得高一些。

表 22.1 HD44780 的接口

信号	引脚数	输入和/或输出？	功能
RS	1	I	选择指令寄存器 (0) 或数据寄存器 (1)
R/W	1	I	选择读取 (1) 或写入 (0)
E	1	I	开始数据读/写
DB4 to DB7	4	I/O	用来在 8051 和 HD44780 之间传送数据。DB7 可用作忙标志（在这里的程序例子中未使用）
DB0 to DB3	4	I/O	用来在 8051 和 HD44780 之间传送接收数据。在 4 位操作模式下不使用这些引脚（在这里的程序例子中未使用）

表 22.2 HD44780 中的基本字符集

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2X		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3X	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4X	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	B	W	X	Y	Z	{	¥	}	^	-
6X	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	l	}	→	←

注：大多数遵循 ASCII 码。然而，注意日元符号（字符 0x5C）代替了 “\”，而 0xE 和 0xF 分别是左右箭头。英国用户应该注意没有 “£” 符号（字符 0x23 是 “#”）。

字符编码很简单，然而向哪个地址写数据稍复杂一些。可以有许多不同的显示配置（包括 16 个字符 x1 行、16x2、20x2、20x4、24x1、24x2、40x1 及 40x2）。这些显示上的每一段的存储单元如表 22.3 所示。

CG RAM

CG RAM 为“字符产生” RAM，可用于创建和显示用户自定义字符。如果希望显示“标准”范围之外的字符，诸如“£”符号，这种功能将很有用。

表 22.3 各种 HD44780 LCD 显示的每一行的起始存储单元

字符数	行数	第1行	第2行	第3行	第4行
16(8)	1(2)	0x00-0x07	0x40-0x47	-	-
16	2	0x00-0x0F	0x40-0x4F	-	-
16	4	0x00-0x0F	0x40-0x4F	0x10-0x1F	0x50-0x5F
20	2	0x00-0x13	0x40-0x53	-	-
20	4	0x00-0x13	0x40-0x53	0x14-0x27	0x54-0x67
24	1	0x00-0x17	-	-	-
24	2	0x00-0x17	0x40-0x58	-	-
40(20)	1(2)	0x00-0x13	0x40-0x53	-	-
40	2	0x00-0x27	0x40-0x67	-	-

在随后的例子中将演示如何使用 CG RAM。

寄存器

HD44780 有两个 8 位寄存器：指令寄存器（IR）和数据寄存器（DR）。

- IR 存储指令代码，诸如清除显示、移动光标以及 DD RAM 和 CG RAM 的地址信息。
- DR 暂时存储写入到 DD RAM 或 CG RAM 中的数据。由 8051 写入到 DR 中的数据通过内部操作自动写入到 DD RAM 或 CG RAM 中。

忙标志 (BF)

当忙标志为 1 时，表示 HD44780 正在执行内部操作，不接收其他指令或数据。当 RS=0 并且 R/W=1 时（如表 22.4 所示），忙标志输出到 DB7。

表 22.4 寄存器选择

RS	R/W	操作
0	0	写入 IR 执行内部操作（清除显示等等）
0	1	读忙标志（DB7）以及地址计数器（DB0 到 DB6）
1	0	写入 DR 执行内部操作（DR 到 DD RAM 或 CG RAM）
1	1	读 DR 执行内部操作（DD RAM 或 CG RAM 到 DR）

光标/闪烁控制电路

正如名称所示，光标/闪烁控制电路产生光标或字符闪烁。光标或闪烁将会在地址计数器（AC）中设置的地址所指向的显示数据 RAM（DDRAM）中的数字的位置上显示。例如，当地址计数器为 0x08 时，光标在 DD RAM 地址 0x08 的位置上显示。

地址计数器 (AC)

地址计数器 (AC) 用于指定 DD RAM 和 CG RAM 的地址。当向 IR 中写入指令地址时，地址信息从 IR 发送到 AC。

选择DD RAM还是CG RAM也由指令决定。

在写入DDRAM或CGRAM之后，AC自动加1。当RS=0以及R/W=1时（表22.4），AC中的内容输出到DB0~DB6上。

解决方案

正如在“背景知识”中讨论的，这里只讨论基于Hitachi的HD44780控制器的LCD器件。

硬件

HD44780以4位操作或者8位操作来发送数据。这种特性可能是最初打算既允许连接4位微控制器又可以连接8位微控制器。然而，即使是8位微控制器（比如8051）的系统，4位接口也是最常用的，这是因为：

1. 对于大多数系统而言4位接口已足够快（传送一个字符小于0.1ms）。
2. 可以节省四个端口引脚。

将在例子中使用4位接口。

尽管其名称是“4位”接口，但它实际上需要六个端口引脚：四个用于数据总线，另外两个端口引脚通常用于控制线（如图22.2所示）。

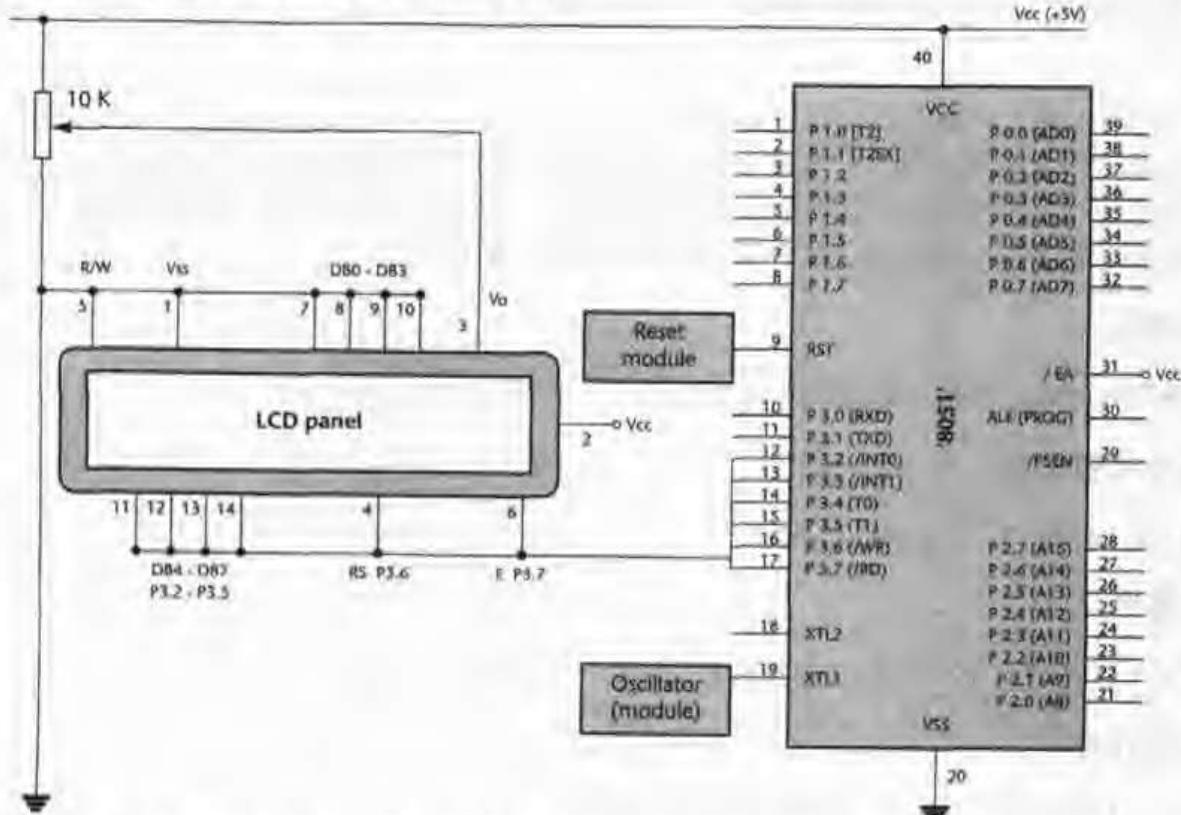


图22.2 8051微控制器和基于HD44780的LCD显示面板之间的4位接口

除地线和电源之外，还需要连接对比度调整引脚（Vo）。虽然许多LCD显示只需将这个

引脚接地，但推荐的对比度调整需要使用一个电位器，其连接如图 22.3 所示。

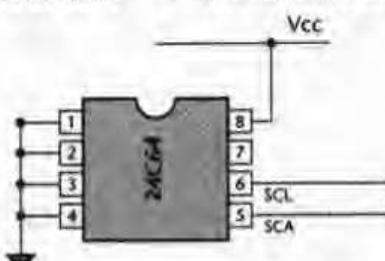


图 22.3 推荐的对比度调整连接

注意，在很多情况下，将 Vo 接地同样也可以工作。

背光

如果模块有背光，将极大地增加功耗，需要核对数据手册来获得详细资料。在确定需要背光之前，不妨试试一些现有的较新的显示模块。这些最新的器件极大地提高了对比度和可见度。

数据总线

数据总线（4 位）需要连接四个有内部上拉电阻的端口引脚。如果端口（例如端口 0）没有内部上拉电阻，则添加外部 10K 上拉电阻（到 Vcc）。

软件

在随后的例子中将给出完整的软件库。

注意，在这些例子中（正如在 RS-232 库中实现的），以多级任务的方式实现了该库。每一个字符被写入缓冲区，而刷新 LCD 显示是通过一个被调度的“刷新”函数来执行的。

同时还要注意，这些例子说明了用户自定义字符的使用。

硬件资源

使用许多端口引脚，以及一个定时器。

可靠性和安全性

LCD 显示可靠并且寿命更长。

可移植性

这个模式可用于任何微控制器系列。

优缺点小结

- ① LCD 提供了专业而且灵活的用户界面。
- ② 面板较昂贵。
- ③ 即使是“4 位”接口也需要六个引脚。

相关的模式和替代解决方案

- 参见多路复用 LED 显示
- 参见 PC 连接 (RS-232)

例子：在 LCD 上显示消逝的时间

这个例子在 LCD 字符面板上显示消逝的时间（源程序清单 22.1~源程序清单 22.3）。

硬件

所需硬件如图 22.2 所示。

软件

```
/*
  Port.H (v1.00)
  项目 LED_TIME 的“端口头文件”（参见第 10 章）
*/
// ----- Sch51.C -----
// 如果不需要错误报告，将这一行注释掉
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1
#endif
// ----- LCD_A.C -----
// 注意，可以使用任何 6 个引脚的组合（任何端口、任何顺序）
// 注意，[]中的数是 LCD 上的引脚号
sbit LCD_D4 = P3^2; // DB4 [11]
sbit LCD_D5 = P3^3; // DB5 [12]
sbit LCD_D6 = P3^4; // DB6 [13]
sbit LCD_D7 = P3^5; // DB7 [14]
sbit LCD_RS = P3^6; // 输出选择显示寄存器[4]
sbit LCD_EN = P3^7; // 输出显示使能[6]
// 连接 LCD 上的 Vss[1]到地
// 连接 LCD 上的 Vcc[2]到+5V
// 连接 LCD 上的 Vo[3]到地
// 连接 LCD 上的 RW[5]到地
/*
  -----文件结束-----
*/
```

源程序清单 22.1 在 LCD 字符面板上显示消逝时间的演示程序的一部分

```
/*
  Elap_LCD.C (v1.00)
*/
```

用于跟踪消逝时间的简单的库函数
这个版本用于 LCD 显示

```
-----*/
#include "Main.h"
#include "Port.h"
#include "Elap_LCD.h"
#include "LCD_A.h"
// -----公有变量定义-----
tByte Hou_G = 0;
tByte Min_G = 0;
tByte Sec_G = 0;
// -----公有变量声明-----
extern char LCD_data_G[LCD_LINES][LCD_CHARACTERS+1];
extern char code CHAR_MAP_G[10];
extern tByte Hou_G, Min_G;
/*-----*
Elapsed_Time_LCD_Init()
用于在 LCD 字符面板上显示消逝时间的简单的库的初始化函数
-----*/
void Elapsed_Time_LCD_Init(void)
{
    // 设置发给 LCD 的初始化数据
    // 假定 20 个字符显示
    char* pTime = "Elapsed time :-      ";
    tByte c;
    for (c = 0; c < LCD_CHARACTERS; c++)
    {
        LCD_data_G[0][c] = pTime[c];

        // 仅用于演示
        if (c % 2)
        {
            LCD_data_G[1][c] = LCD_UDEC_DEGREES_C;
        }
        else
        {
            LCD_data_G[1][c] = LCD_UDEC_POUNDS;
        }
    }
    LCD_data_G[0][15] = CHAR_MAP_G[Hou_G / 10];
    LCD_data_G[0][16] = CHAR_MAP_G[Hou_G % 10];
    LCD_data_G[0][18] = CHAR_MAP_G[Min_G / 10];
    LCD_data_G[0][19] = CHAR_MAP_G[Min_G % 10];
}
/*-----*
Elapsed_Time_LCD_Update()
用于在 LCD 字符面板上显示消逝时间的函数
***必须每秒调度一次***
-----*/
```

```
void Elapsed_Time_LCD_Update(void)
{
    bit Min_update = 0;
    bit Hou_update = 0;
    // 设置为（例如）2 来测试，否则为 60
    if (++Sec_G == 60)
    {
        Sec_G = 0;
        Min_update = 1;

        if (++Min_G == 60)
        {
            Min_G = 0;
            Hou_update = 1;

            if (++Hou_G == 24)
            {
                Hou_G = 0;
            }
        }
    }
    if (Hou_update)
    {
        // 需要刷新“小时”数据
        LCD_data_G[0][18] = CHAR_MAP_G[Hou_G / 10];
        LCD_data_G[0][19] = CHAR_MAP_G[Hou_G % 10];
        Hou_update = 0;
    }
    if Min_update = 0;
    }

    // 在这个版本中不显示秒
    // 使用秒数据来打开和关闭小时和分钟之间的小数点
    // (between hours and minutes)
    if ((Sec_G % 2) == 0)
    {
        LCD_data_G[0][17] = ' ';
    }
    else
    {
        LCD_data_G[0][13] = ':';
    }
}

/*
-----文件结束-----
*/
```

源程序清单 22.2 在 LCD 字符面板上显示流逝时间的演示程序的一部分

```

/*
LCD_A.C (v1.01)

LCD 库代码
为调度运行而设计
在这个例子中用于 2 行 x20 个字符显示
4 位接口 (使用 6 个引脚) 连接基于标准 HD44780 的 LCD
*/
// 硬件资源
// 使用 T0 (用于延迟) 加上六个输入/输出引脚
#include "Main.h"
#include "Port.h"
#include "LCD_A.h"
#include "Delay_T0.h"
// -----公有变量定义-----
// LCD 数据
char LCD_data_G[LCD_LINES][LCD_CHARACTERS+1]
= {" PLEASE WAIT      , " ...      "};

// -----私有函数原型-----
static void LCD_Send_Byte(const tByte, const bit);
static void LCD_Create_Character(const tByte, const tByte* const);
static void LCD_SetDDRAM(tByte);
static void LCD_Delay(void);

// -----私有的常数-----
// 用户自定义字符的位图 [仅用于演示]
// 这是一个英镑 (货币) 符号
// 765 43210
// ... .11 - 3 (Decimal)
// ... .1.. - 4
// ... .111. - 14
// ... .1.. - 4
// ... .1.. - 4
// ... .1.. - 4
// ... 11111 - 31
// ... ..... - 0
const tByte LCD_UDC_Pounds[8] = {3,4,14,4,4,31,0};
// #define LCD_UDC_POUNDS 1 (See LCD_A.H)
// 这是“摄氏温度” (沸水温度=100°C)
// 765 43210
// ... .11.. = 12 (Decimal)
// ... 1..1. = 18
// ... .11.. = 12
// ... ...11 = 3
// ... .1.. = 4
// ... .1.. = 4
// ... ...11 = 3
// ... ..... - 0
const tByte LCD_UDC_Degrees_C[8] = {12,18,12,3,4,4,3,0};
// #define LCD_UDC_DEGREES_C 2 (See LCD_A.H)

```

第 22 章 控制 LCD 显示面板

```
#define LCD_IND_ADDR_NO_SCROLL 0x06
#define LCD_CURSOR_OFF 0x08
#define LCD_DISPLAY_ON 0x04
#define LCD_CLEAR_DISPLAY 0x01
#define LCD_8BIT_2LINE_5x8FONT 0x38 // 0011 1000
#define LCD_4BIT_2LINE_5x8FONT 0x28 // 0010 1000
// 定义定时器0/定时器1的重装值为50μs 延迟
#define PRELOAD50micros (65536 - (tWord)((OSC_FREQ / 20000) / (OSC_PER_INST)))
#define PRELOAD50microsH (PRELOAD50micros / 256)
#define PRELOAD50microsL (PRELOAD50micros % 256)
/*-----*/
LCD_Init()
速度相当慢，但是可以初始化所有显示测试（并且只在该程序开始时调用）
注意，建议调用这个函数三次
/*-----*/
void LCD_Init(void)
{
    tByte loop;
    tByte l,c;
    // 设置LCD端口
    LCD_D4 = 1;
    LCD_D5 = 1;
    LCD_D6 = 1;
    LCD_D7 = 1;
    Hardware_Delay_T0(500);
    LCD_RS = 0;
    LCD_EN = 0;
    // 现在等待显示的初始化
    // 数据手册称至少40ms
    Hardware_Delay_T0(100);
    // 数据手册称发送这个指令3次...
    for (loop = 0; loop < 3; loop++)
    {
        // 使用4位总线，2行显示以及5x7点阵字体
        LCD_Send_Byte(LCD_4BIT_2LINE_5x8FONT, 0);
        Hardware_Delay_T0(100);
    }
    // 增加每个字符的显示地址但是不卷屏
    LCD_Send_Byte(LCD_INC_ADDR_NO_SCROLL, 0);
    Hardware_Delay_T0(50);
    // 打开显示并且关闭光标
    LCD_Send_Byte((LCD_CURSOR_OFF | LCD_DISPLAY_ON), 0);
    Hardware_Delay_T0(50);
    // 清除显示
    LCD_Send_Byte(LCD_CLEAR_DISPLAY, 0);
    Hardware_Delay_T0(50);
    // 隐去光标（调用空函数以避免库出错）
    LCD_Control_Cursor(0,0,0);
    Hardware_Delay_T0(200);
```

```

// 如果需要的话，设置用户自定义字符
LCD_Create_Character(LCD_UDC_DEGREES_C, LCD_UDC_Degrees_C);
Hardware_Delay_T0(200);
// 刷新显示中的一个字符
for (l = 0; l < LCD_LINES; l++)
{
    for (c = 0; c < LCD_CHARACTERS; c++)
    {
        LCD_data_G[l][c] = ' ';
        LCD_Update();
        Hardware_Delay_T0(10);
    }
}
/*-----*
LCD_Update()
这个函数刷新 LCD 显示面板中的一个字符（如果需要刷新的话）
持续时间：0.1ms
大约每隔 25ms 调度一次（2 行 x20 个字符显示），这样每隔一秒刷新整个显示
*-----*/
void LCD_Update(void)
{
    static tByte Line;
    static tByte Character;
    tByte Tests, Data;
    bit Update_required;
    // 查找下一个需刷新字符
    Tests = LCD_CHARACTERS * LCD_LINES;
    do {
        if (++Character == LCD_CHARACTERS)
        {
            Character = 0;
            if (++Line == LCD_LINES)
            {
                Line = 0;
            }
        }
        // 在数据写入 LCD 之后将队列内容设置为\0
        Update_required = (LCD_data_G[Line][Character] != '\0');
    } while ((Tests-- > 0) && (!Update_required));
    if (!Update_required)
    {
        return; // LCD 中无数据需要刷新
    }
    // 设置字符写入 DD RAM 的地址
    // -假定 2 行 x20 个字符显示
    // -用于其他显示尺寸的调整参见表 22-3
    if (Line == 0)
    {

```

```
LCD_SetDDRAM(0x00 + Character); // 第一行
}
else
{
    LCD_SetDDRAM(0x40 + Character); // 第二行
}
// 这是刷新数据
Data = LCD_data_G[Line][Character];
// 发送一个数据字节
LCD_Send_Byte(Data,1);
// 一旦数据已经写入 LCD
LCD_data_G[Line][Character] = '\0';
}

/*-----*
LCD_Send_Byte()
// 这个函数向 LCD 显示面板写入一个字节。
持续时间<0.1ms。
A 参数: DATA
    要写入显示的字节
    DATA_FLAG:
        如果 DATA_FLAG==1, 发送一个数据字节
        如果 DATA_FLAG==0, 发送一个命令字节
-----*/
void LCD_Send_Byte(const tByte DATA, const bit DATA_FLAG)
{
    // 需要延迟
    // [某些显示有可能减少延迟]
    LCD_D4 = 0;
    LCD_D5 = 0;
    LCD_D6 = 0;
    LCD_D7 = 0;
    LCD_RS = DATA_FLAG; // 数据寄存器
    LCD_EN = 0;
    LCD_Delay();
    // 写入数据 (高 4 位)
    LCD_D4 = ((DATA & 0x10) == 0x10);
    LCD_D5 = ((DATA & 0x20) == 0x20);
    LCD_D6 = ((DATA & 0x40) == 0x40);
    LCD_D7 = ((DATA & 0x80) == 0x80);
    LCD_Delay();
    LCD_EN = 1; // 锁存高 4 位
    LCD_Delay();
    LCD_EN = 0;
    LCD_Delay();
    LCD_D4 = 0;
    LCD_D5 = 0;
    LCD_D6 = 0;
    LCD_D7 = 0;
    LCD_RS = DATA_FLAG;
```

```

LCD_EN = 0;
LCD_Delay();
// 写入数据(低4位)
LCD_D4 = ((DATA & 0x01) == 0x01);
LCD_D5 = ((DATA & 0x02) == 0x02);
LCD_D6 = ((DATA & 0x04) == 0x04);
LCD_D7 = ((DATA & 0x08) == 0x08);
LCD_Delay();
LCD_EN = 1; // 锁存低4位
LCD_Delay();
LCD_EN = 0;
LCD_Delay();
}
*/
LCD_Control_Cursor()
这个函数使能或清除光标并且将光标移动到特定的位置
参数: Visible-如果需要光标可见时设置
      Blinking-如果需要字符闪烁时设置
      Address-想调整的(DD RAM)的地址
void LCD_Control_Cursor(const bit VISIBLE, const bit BLINKING,
                        const tByte ADDRESS)
{
// 光标/闪烁出现在当前的 DD RAM 地址
// - 使用 SetDDRAM() 来改变光标位置
tByte Command = 0x0C;
if (VISIBLE)
{
    Command |= 0x02;
}
if (BLINKING)
{
    Command |= 0x01;
}
LCD_Send_Byte(Command, 0);
LCD_SetDDRAM(ADDRESS);
}
*/
LCD_SetDDRAM()
将 DDRAM 设置到某个地址用来确定向 LCD RAM 的什么地方写入,
无论文本显示是在第 0 行还是第 1 行等等
详细资料参见正文
参数: 想要写入的 DDRAM 地址
void LCD_SetDDRAM(tByte ADDRESS)
{
ADDRESS &= 0x7F;
ADDRESS |= 0x80;
LCD_Send_Byte(Address, 0);
}

```

```

}

/*-----*
LCD_Create_Character()
在 CG RAM 中存储用户自定义字符。用这种方法可以最多保存 8 个字符
注意，假定字符大小为 5x8。如果需要 5x11 的字符，将需要修改这些代码
参数：字符数据（参见文件的开头）
*-----*/
void LCD_Create_Character(const tByte UDC_ID,
                           const tByte* const pUDC_PAT)
{
    tByte Row;
    tByte Address;
    // 选择 CG RAM，相应的地址
    Address = 0x40 + (UDC_ID << 3);
    LCD_Send_Byte(Address, 0);
    // 现在写入数据
    for (Row = 0; Row < 8; Row++)
    {
        LCD_Send_Byte(pUDC_PAT[Row], 1);
    }
    // 选择 DD RAM，地址 0
    Address = 0x80;
    // 保证下一个数据写入 DD RAM
    LCD_Send_Byte(Address, 0);
}

/*-----*
LCD_Delay()
这个函数为 LCD 库提供短延迟
*-----*/
void LCD_Delay(void)
{
    int x;
    x++;
    x++;
    x++;
}

/*-----*
-- 文件结束 --
*-----*/

```

源程序清单 22.3 在 LCD 字符面板上显示消逝时间的演示程序的一部分

例子：LCD 和键盘

在这个例子中展示一个 LCD 和键盘接口组合（图 22.4）。

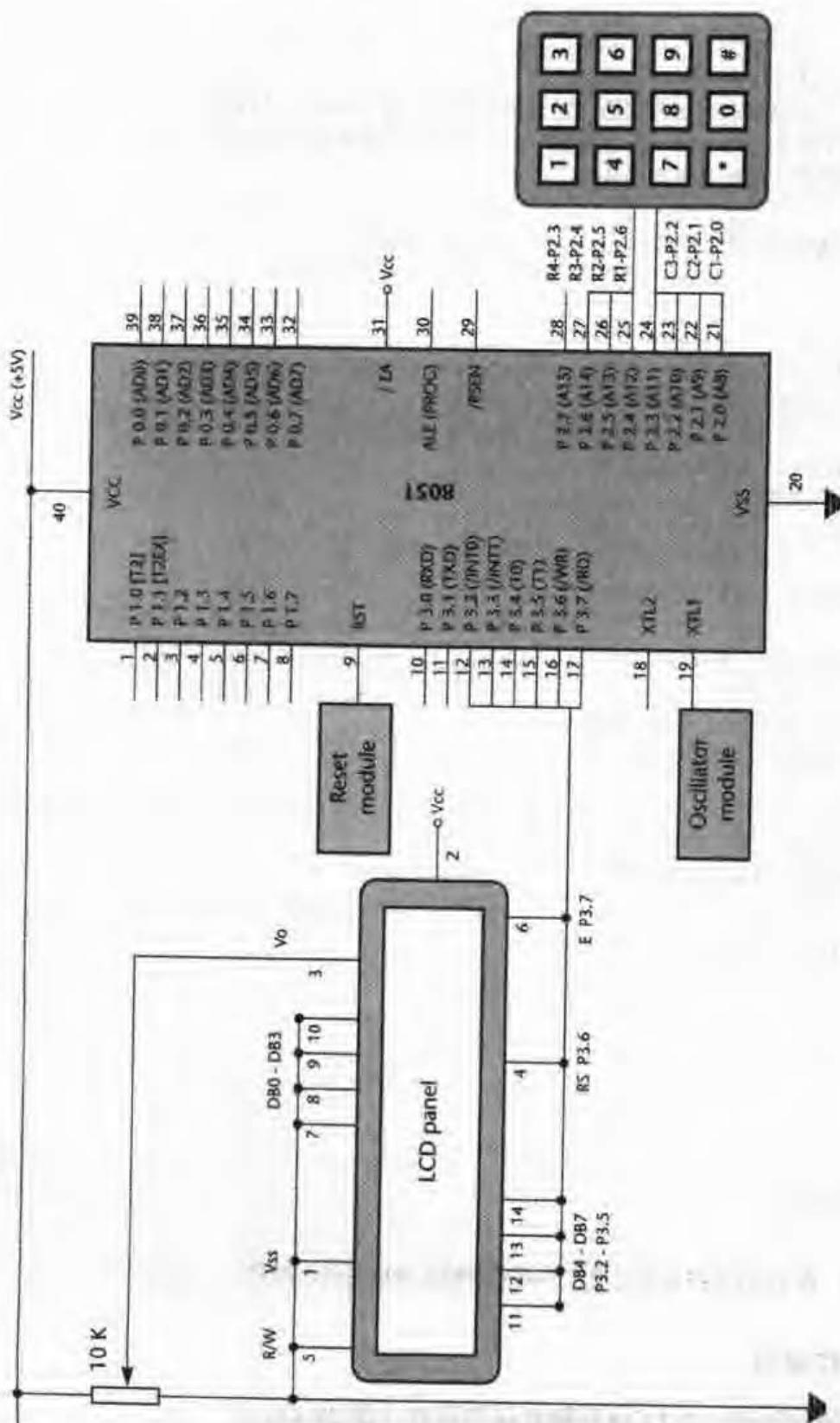


图 22.4 用来展示由 LCD 显示和小键盘组成的用户界面的硬件

这个例子的主要的源文件如下（源程序清单 22.4~源程序清单 22.6），所有文件都包含在 CD 上。

```
/*
Port.H (v1.00)

'Port file' (see Chap 10) for the project SCH_TEST.PRG
//项目 SCH_TEST.PRG 的“端口文件”（参见第 10 章）
*/
// ----- Sch51.C -----
// 如果不需要错误报告，将这一行注释掉
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1
#endif
// ----- Keypad.C -----
#define KEYPAD_PORT P2
sbit C1 = KEYPAD_PORT^0;
sbit C2 = KEYPAD_PORT^1;
sbit C3 = KEYPAD_PORT^2;
sbit R1 = KEYPAD_PORT^6;
sbit R2 = KEYPAD_PORT^5;
sbit R3 = KEYPAD_PORT^4;
sbit R4 = KEYPAD_PORT^3;
// ----- LCD_A.C -----
// 注意：可以使用任何 6 个引脚的组合（任何端口、任何顺序）
// 注意：[] 中的数是许多 LCD 上的引脚号
sbit LCD_D4 = P3^2; // DB4 [11]
sbit LCD_D5 = P3^3; // DB5 [12]
sbit LCD_D6 = P3^4; // DB6 [13]
sbit LCD_D7 = P3^5; // DB7 [14]
sbit LCD_RS = P3^6; // 输出选择显示寄存器 [4]
sbit LCD_EN = P3^7; // 输出显示使能 [6]
// 连接 LCD 上的 Vs[1] 到地
// 连接 LCD 上的 Vcc[2] 到 +5V
// 连接 LCD 上的 Vo[3] 到地
// 连接 LCD 上的 RW[5] 到地
/*
---文件结束---
*/

```

源程序清单 22.4 用来展示由 LCD 显示和小键盘组成的用户界面的软件

```
/*
Main.c (v1.00)

用于键盘-LCD 的演示程序。

```

这个版本用于'8052'/11.059MHz/5ms 时标
 要求的链接程序选项（详细资料参见第 14 章）

```

OVERLAY (main ~ (LCD_Update, LCD_KEY_Update),
SCH_dispatch_tasks ! (LCD_Update, LCD_KEY_Update)))
-----*/
#include "Main.h"
#include "2_05_11g.H"
#include "LCD_A.h"
#include "Keypad.h"
#include "LCD_Key.h"
/* ..... */
/* ..... */
void main(void)
{
    // 为任务作准备
    LCD_Init(); // 3x!!!
    LCD_Init();
    LCD_Init();
    KEYPAD_Init();
    LCD_KEY_Init();
    // 设置调度器
    SCH_Init_T2();
    // 根据 LCD 缓冲区刷新 LCD
    SCH_Add_Task(LCD_Update, 1000, 24);
    // 传送键盘数据到 LCD 缓冲区
    SCH_Add_Task(LCD_KEY_Update, 2000, 50);
    // 开始调度器
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
-----文件结束-----
-----*/

```

源程序清单 22.5 用来展示由 LCD 显示和小键盘组成的用户界面的软件

```

/*
LCD_Key.C (v1.00)
-----
用于传送键盘输入到 LCD 显示的简单的演示函数
-----
#include "Main.h"
#include "Keypad.h"

```

```

#include "LCD_Key.h"
#include "LCD_A.h"
// -----私有变量-----
static tByte Char_G;
// -----公有变量声明-----
extern char LCD_data_G[LCD_LINES][LCD_CHARACTERS+1];
/*-----*/
LCD_KEY_Init()
用于简单的键盘 LCD 库的初始化函数
*/
void LCD_KEY_Init(void)
{
    tByte c;
    char* pPrompt = "Type on the keypad:      ";
    // 清除键盘缓冲区
    KEYPAD_Clear_Buffer();
    // 准备显示
    for (c = 0; c < LCD_CHARACTERS; c++)
    {
        LCD_data_G[0][c] = pPrompt[c];
        LCD_data_G[1][c] = '*';
    }
    Char_G = 0;
}
/*-----*/
LCD_KEY_Update()
通过 LCD 显示键盘输入的函数
*/
void LCD_KEY_Update(void)
{
    char Key, FnKey;
    KEYPAD_Update();
    if (KEYPAD_Get_Data_From_Buffer(&Key, &FnKey) == 0)
    {
        // 缓冲区是空的
        return;
    }
    // 功能键与另一个按键被同时按下
    if (FnKey)
    {
        LCD_data_G[1][Char_G] = FnKey;
        if (++Char_G == LCD_CHARACTERS)
        {
            Char_G = 0;
        }
    }
}

```

```
        }
LCD_data_G[1][Char_G] = Key;

if (++Char_G == LCD_CHARACTERS)
{
    Char_G = 0;
}
}

/*-----文件结束-----*/

```

源程序清单 22.6 用来展示由 LCD 显示和小键盘组成的用户界面的软件

进阶阅读

Part 5

使用串行外围模块

在第 5 篇中，将讨论如何在时间触发结构的系统中使用两种功能强大而又有影响的串行通信协议（“I²C” 和 “SPI”）。使用这些协议有两个主要的好处：

- 使微控制器不需要使用大量端口引脚而连接各种不同的外围模块：存储器、显示、模数转换器以及类似芯片。
- 所有外围模块（例如 SPI）可以共用一组软件代码，减少了所需的研制工作。

在第 23 章中，将讨论由 Philips 开发的 I²C 总线。I²C 是一种简单的协议，可以很容易地用软件来产生。这使整个 8051 系列芯片（包括只有很少空闲端口引脚的精简 8051）都能够与各种外围设备通信。

在第 24 章中，将讨论由 Motorola 开发的 SPI 总线。越来越多的“标准”以及“扩展”8051 芯片对 SPI 有硬件支持，我们将在第 24 章使用这些功能。



Chapter 23

使用 I²C 外围模块

引言

I²C 总线是一种二线式的串行通信总线，最初由 Philips 公司在 1992 年推出。现在许多半导体厂商都支持 I²C，因此许多不同的外围设备（LCD、LED 显示、EEPROM、模数转换器以及数模转换器芯片等等）可以不使用大量端口引脚而连接到微控制器上。

本章中的模式解决了以下问题：

- 是否使用 I²C 协议来连接微控制器和外围设备，如果要使用，如何连接？

I²C 外围模块

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用基于调度器的时间触发结构。
- 系统中的微控制器需要连接多个外围模块，诸如键盘、EEPROM、数模转换器或类似芯片。

问题

是否使用 I²C 协议来连接微控制器和外围设备，如果使用，如何连接？

背景知识

就嵌入式系统开发人员而言，I²C 有五个主要特性：

- I²C 是一种用来将微控制器连接到各种外围模块的协议。这些外围模块包括存储器、显示、模数转换器，以及类似的芯片。I²C 只需要两个端口引脚，一般可以连接多达 20 个外围模块。
- 可以购买到许多现成的 I²C 外围设备。

- I²C 是一种简单的协议，可以很容易地用软件来产生。可以使所有 8051 芯片与各种外围设备进行通信。
- 所有 I²C 外围模块可以共用一组软件代码。
- 即使由软件产生，I²C 也足够快，可以与时间触发结构相兼容。典型的数据传输速率高达 1000 字节/秒（采用 1ms 调度器时标）。

本节中提供了解和使用 I²C 总线所需的一些背景知识。请注意，这里提供的许多材料都是由 Philips (1998) I²C 规约改编而来的。如需要详细了解，读者可以从 Philips 的网站获得这个文档。^①

硬件

首先考虑基本的 I²C 硬件功能。

总线

在二线式的 I²C 总线中，串行数据线 (SDA) 和串行时钟线 (SCL) 传送各种芯片之间的信息（图 23.1）。I²C 总线上连接的芯片所采用的不同技术（CMOS、NMOS、双极）决定了逻辑“0”（低）和“1”（高）的电平也不尽相同，它们也同时取决于系统的电源电压。

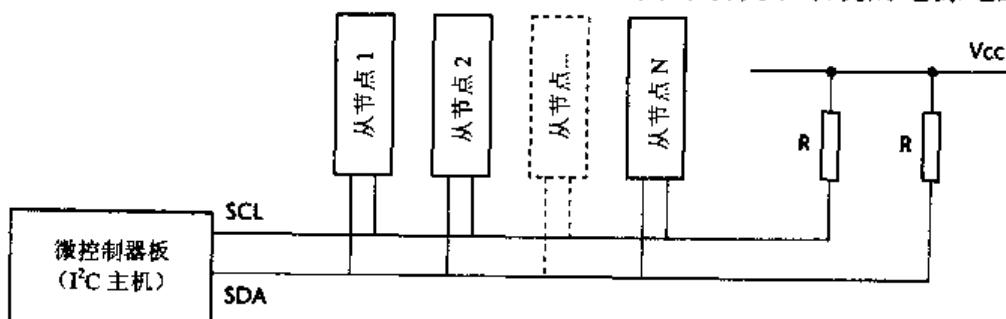


图 23.1 使用 I²C 总线

当总线空闲时，SCL 和 SDA 线都为高。

SDA 和 SCL 都为双向信号线。连接到总线上的芯片的输出级应该为集电极开路以满足协议的要求。这往往意味着使用 8051 芯片的端口 0 上的引脚（但是注意下面框中的内容）。

连接到 I²C 总线上的芯片数量仅仅受限于最高为 400pF 的总线负载电容。当缺乏相关信息时，可以假定每个外围设备和它的接线将带来总共 20pF 的电容。

每根线通过一个公用的上拉电阻连接到 Vcc。如果假定每个芯片都具有 20pF 电容，则所需的电阻值取决于 I²C 规约中的最大上升时间（标准 I²C 中为 1000ns）。如下所示：

$$R = \frac{50}{d}$$

其中：R 是所需的电阻值 (kΩ)

d 是总线上的芯片数量

^① www.philips.com

注意，在各种应用系统中，我们已经成功地使用 8051 芯片中的“普通的”（而不是集电极开路）引脚用于 I²C 通信，而不需使用外部上拉电阻。

如果需要同样的实现，建议只用于使用少量外围模块的场合，而且需要充分测试最后所得到的系统。

设备地址

在 I²C 中，通信是基于地址的。也就是说，总线上的每个芯片具有惟一的地址，信息可以从总线上的任何地方发送到具有某个地址的芯片。

在最新版本的标准中，芯片地址为十位（在最初的标准中为七位），允许使用 1000 个以上不同的地址。注意，7 位地址的芯片与 10 位地址的芯片可以在同一条总线上使用。

I²C 规约包含了 7 位或 10 位地址功能，然而在大多数的实际情况下，部分芯片地址由芯片的硬线连接决定。

例如，考虑一种有用的 I²C 外围设备：Atmel 的 24C64 串行 EEPROM^②。这个芯片提供了 8192×8 位非易失性数据存储空间，数据可以保持 100 年。因而，在监测系统中用它来存储少量的数据，或者在某些设备中用于存储系统口令都是很理想的。

24C64 的引脚在图 23.2 中给出。注意，图中只有三根地址线，通常接到 Vcc 或地来设置所需的设备地址。假设每个芯片给定一个惟一的地址，则 I²C 总线上最多可以连接 8 个 24C64，而不是 I²C 标准推荐的 128 (2^7) 或 1024 (2^{10})。

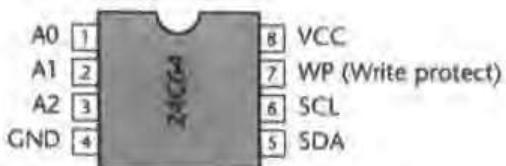


图 23.2 Atmel 的 24C64 串行 EEPROM 的引脚配置

对于总线而言，24C64 实际上具有 7 位地址。然而，最高的四位由芯片内部“硬线连接”决定。这些位始终为“1010”(0xA0)，许多串行 EEPROM 都通用这些编码。因此完整的设备地址为：“1 0 1 0 A2 A1 A0”。

设备地址的一部分由硬接线决定实际的意义。几乎没有哪个系统在单个总线上需要八个以上的串行 EEPROM，特别是因为现在可以选用具有相同接口的容量很大的存储器。如果需要由用户编码每个芯片的完整设备地址，那么芯片将需要增加至少四个引脚，因而增加了芯片的尺寸（如图 23.3 所示）。

其他 I²C 设备也使用非常类似的方案。例如，Dallas 的 1621 温度传感器^③是一种廉价而有用的元件，它能够通过 I²C 总线提供以摄氏度为单位的温度读数（如图 23.4 所示）。

② 8051 与 24C64 芯片的 I²C 连接的完整例子在第 510 页给出。

③ 8051 与 DS1621 温度传感器的 I²C 连接的完整例子在第 515 页给出。

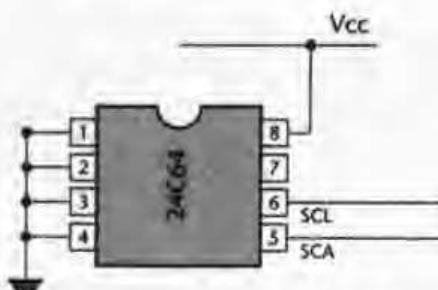
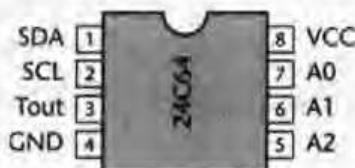


图 23.3 设备地址为“1010000”的 24C64

图 23.4 Dallas 的 DS1621 (I^2C) 温度传感器的引脚配置

这里，用户同样只能设置地址的三个最低位。在这个例子中，芯片的完整地址是“1001 A2 A1 A0”，地址的硬接线部分（0x90）由许多数模转换芯片共享。

I^2C 协议：主机和从机

I^2C 总线上的每个芯片都可以作为数据的发送器或接收器。例如，LCD 驱动器通常会作为接收器，而存储芯片既可以作为接收器又可以作为发送器。除发送器和接收器之外，芯片同样可以被看作是主机或者从机。主机启动总线数据传输并且产生允许传输的时钟信号。此时，任何设备地址都被认为是从机。

I^2C 总线具有多主机功能。这指的是总线上可以连接多个能够控制该总线的芯片，这将非常有用。例如，用 I^2C 在两个微控制器之间传输数据。

请注意：在这个模式中只关注单个微控制器（主机）和多个从机外设的 I^2C 应用。

I^2C 总线的时钟信号始终由主机产生。主机在总线上传输数据时产生自己的时钟信号。在单主机系统中，只有当一个低速从机通过拉低时钟线来延伸总线时钟信号时，才可以改变主机产生的时钟。

执行数据传输

现在考虑数据是如何在芯片之间传送的。例如，考虑希望从前面对 DS1621 温度传感器读取温度值，如下操作：

1. 产生 START 状态
2. 发送 DS1621 的设备地址（以及写访问请求）
3. 确信从机（DS1621）产生了 ACKNOWLEDGE 位
4. 发送命令“读取温度”（0xAA）



5. 确信从机 (DS1621) 产生了 ACKNOWLEDGE 位
6. 产生另一个 START 状态
7. 发送 DS1621 的设备地址 (以及读访问请求)
8. 确信从机 (DS1621) 产生了 ACKNOWLEDGE 位
9. 从 I²C 总线接收温度数据的第一个 (最高) 字节
10. 产生主机 ACKNOWLEDGE
11. I²C 总线接收温度数据的第二个字节
12. 产生主机 NOT ACKNOWLEDGE
13. 产生 STOP 状态

虽然这个过程初看起来相当复杂，但它包含了与其他 I²C 外设通信所需的所有 I²C 内核程序。因此，当讨论了这个过程之后，你将能够开发用于各类芯片的 I²C 接口。

参考图 23.5 将有助于理解下述讨论。

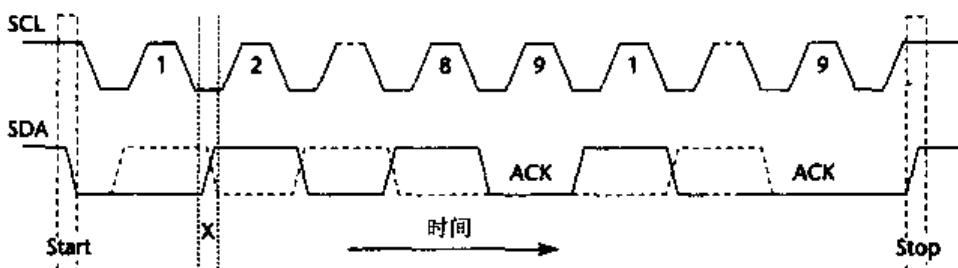


图 23.5 I²C 通信的基本格式

注意，对应传送的每个数据位都产生一个时钟脉冲。在时钟为高期间，SDA 线上的数据必须稳定。只有当 SCL 线上的时钟信号为低时才能改变数据线的状态。

首先，讨论 START 和 STOP 状态。

START 和 STOP 状态

I²C 通信从 START 状态开始，在消息内部可以产生更多的 START 状态，消息以 STOP 状态结束。

START 状态表示如下（参见图 23.5）：当 SCL 为高时，SDA 线上由高到低跳变。

STOP 状态表示如下（参见图 23.5）：当 SCL 为高时，SDA 线上由低到高跳变。

START 和 STOP 状态始终由主机产生。

字节格式

SDA 线上的每个字节都必须为八位。每个传输可以发送的字节数不受限制，每个字节后面必须有 ACKNOWLEDGE 位（参见下文）。

先传送数据的最高位。如果从机在完成一些内部操作（例如刷新存储器）以前不能接收或发送另一个完整的数据字节，它可以将 SCL 线拉低，使主机进入等待状态。当从机准备好接收或发送下一个数据字节时，释放 SCL，继续数据传输。

从机“ACKNOWLEDGE”和“NOT ACKNOWLEDGE”

在许多场合中，主机（在这里的例子中即微控制器）将发送数据到从机（例如，一个

EEPROM)。在这些例子中，当从机接收器接收了一个字节的数据时，必须产生从机接收器 ACKNOWLEDGE 信号。

ACKNOWLEDGE 相关的时钟脉冲始终由主机产生。在 ACKNOWLEDGE 时钟脉冲期间，发送器释放 SDA 线(高)。从机接收器必须在 ACKNOWLEDGE 时钟脉冲期间拉低 SDA 线，使得在时钟脉冲为高时保持为低。

通常，活动的从机接收器必须在接收到每个字节后产生一个 ACKNOWLEDGE。

当从机不应答从机地址(例如，因为执行一些内部操作不能接收或者发送)时，从机必须使数据线保持为高。然后主机或者产生 STOP 状态来中止传输，或者重复 START 状态来启动新的传输。

如果从机接收器对从机地址做了应答，但是在后面的传输过程中无法接收更多的数据字节，主机则必须再次中止传输。从机通过对接下来的第一个字节产生一个“NOT ACKNOWLEDGE”响应来通知主机这种情况。这意味着从机在主机产生的 ACKNOWLEDGE 时钟脉冲期间使数据线保持为高。主机将产生 STOP(或者重复 START)状态，在一段延迟之后将再次尝试发送。

主机接收器“ACKNOWLEDGE”和“NOT ACKNOWLEDGE”

在一些场合中，主机(在这里的例子中即微控制器)将接收来自从机(例如温度传感器)的数据。在一些例子中，当主机接收器接收了一个字节的数据时，必须产生主机接收器 ACKNOWLEDGE 信号。然而，在数据传输的最后(当主机接收器已经收到它所需的最后一个字节的数据时)，必须通过在从机输出的最后一个字节上不产生 ACKNOWLEDGE 来通知从机发送器已到数据的末尾。从机发送器必须释放数据线，从而允许主机产生 STOP 或者重复 START 状态。

同步

所有主机都在 SCL 线上产生自己的时钟，以便在 I²C 总线上传送信息。由于数据仅在时钟为高期间有效，因此需要定义一个时钟来完成逐位的判断操作。

I²C 到 SCL 线的接口使用线与方式来完成时钟同步。这意味着 SCL 线上由高到低的跳变将导致相关的芯片开始对时钟低周期计数，一旦芯片的时钟变低，它将把 SCL 线拉低，直到时钟变高。然而，如果另一个芯片的时钟仍然在低周期之内，则时钟由低到高的跳变可能不会改变 SCL 线的状态。因此 SCL 线保持为低的周期将取决于具有最长低周期的芯片。在这段时间内，具有较短低周期的芯片将进入高等待状态。

当所有相关的芯片都结束低周期时，时钟线释放为高。此时，芯片时钟和 SCL 线的状态之间将没有差异，所有芯片将开始对高周期计数。首先完成高周期的芯片会再次将 SCL 线拉低。这样，将产生一个同步的 SCL 时钟，其低周期取决于具有最长时钟低周期的芯片，而高周期取决于具有最短时钟高周期的芯片。

更详细的资料

I²C 总线的完整技术规范请参见 Philips 的 WWW 网站。

解决方案

是否应使用 I²C?

为了确定在时间触发系统中使用 I²C 总线是否合适，讨论当考虑使用任何通信协议或者相关技术时应该关心的一些主要问题。

主要应用领域

I²C 总线设计为主要用于单个系统内部的模块互连。虽然可以使用 I²C 总线来将处理器（通常为微控制器）连接到另一个处理器或者其他计算机系统，然而其主要用途是将标准外围设备连接到微控制器。这些标准外围设备包括 LCD 显示面板、EEPROM 或者消费应用中的“智能”模块（例如，电视调节器、PLL 合成器及视频处理器）。

易于开发

I²C 可用来与大量的外设通信。通过使用同样的协议与一系列的芯片通信，可以减少研制工作量。

可伸缩性

连接到 I²C 总线上的芯片数量仅仅受限于最高为 400pF 的电容。为了估计网络的最大容量，可以将所有想要连接到总线上的芯片的输入电容累加起来。

例如，SGS-Thomson 与 I²C 兼容的 ST24C16 16K 的 EEPROM，其输入电容为大约 8pF，可以在同一个总线上连接大约 50 个这样的芯片。注意，这种粗略估计忽略了电缆线路本身的影响。如果假定每个芯片的“标准”电容为 20pF（包括寄生电容在内），则仍然可以在一条总线上将大约 20 个外设连接到微控制器上。

总的说来，I²C 总线支持足够多的外设以满足其主要应用领域的需要。

灵活性

I²C 很灵活。可以实现单主机多从机的网络，也可以实现多个主机的网络。注意，在本模式中仅讨论单主机系统。

执行速度和代码长度

正如已经提到的，I²C 可以运行在很宽的速度范围内：从“标准”版（1992）的 100 kB/s，到“快速”版的 400 kB/s，而在 1998 年发布的最新的“高速”版中可以达到 3.4 MB/s。

然而，在本模式中我们只关注全部由软件创建的 I²C 接口，这样的接口只适用于低速系统。

生成 I²C 协议不可避免地将增加代码的长度，详细资料参见 I²C 内核库（在下面的例子中）。

成本

I²C 总线的许可证费用包含在购买外设模块的费用中。在大多数情况下，不需要其他的费用。

注意，如果要设计连接到 I²C 总线上的 I²C 外设（用于出售），则可能需要额外的费用。如有疑问，请联系 Philips 公司。

实现芯片和供应厂商的选择

这里给出的 I²C 库可以用于任何 8051 芯片。

适用于时间触发系统

正如在第 18 章看到的，RS-232 通信协议适用于时间触发系统。这是因为在 RS-232 网络上发送（以及接收）数据的相关任务的运行时间非常短。注意，传输时间与网络的波特率没有直接关系，这主要是因为几乎所有的 8051 系列芯片都有对 RS-232 的硬件支持，因此信息“在后台”发送与接收。

I²C 的情况却有很大的不同。具体来说，在这个模式中，我们将关注基于软件的 I²C 协议，这无疑将增加软件开销。例如，考虑发送一个字节数据到基于 I²C 的 ROM 芯片的过程（在后面给出完整的例子），该任务总的运行时间为 0.5ms 左右。

如果最高数据速率（1000 字节/秒）能够满足系统的需要，那么即使采用 1ms 定时器时标，时间触发系统也可以支持这样的任务运行时间。

如何在时间触发系统中使用 I²C？

下面的例子包含了一个完整的 I²C 库，可以用于 8051 系列的任何芯片。

硬件资源

I²C 需要使用两个端口引脚，这比创建一个连接大多数外围设备的并行接口所需的端口引脚要少得多。

然而，I²C 需要很大的 CPU 开销，详细资料参见“解决方案”一节。

可靠性和安全性

I²C 协议中只加入了很少的错误检测机制。如果需要的话，检测外围设备传输的数据是否损坏就必须在软件中执行。大多数情况下，将迅速地检测到电缆线路的损坏。

可移植性

I²C 是一种简单的协议，可以很容易地用软件来产生。这使得此处给出的技术可以用于所有 8051 芯片。

优缺点小结

- ◎ 有大量的外围设备支持 I²C。
- ◎ I²C 只需要两个端口引脚，一般可以连接最多 20 个外设。
- ◎ I²C 是一种简单的协议，可以很容易地用软件来产生，实现了所有 8051 芯片与各种外围设备间的通信。
- ◎ 所有 I²C 外围模块可以共用同一组软件代码。
- ◎ 虽然 I²C 足够快（即使由软件产生），可以与时间触发结构兼容，但典型的数据传输

速率相对较低（采用 1ms 调度器时标时，最多为 1000 字节/秒）。

相关的模式和替代解决方案

参见 SPI 外设模式（第 24 章）。

同时记住，如果可能的话，最好完全避免使用外围模块，而使用微控制器片内功能。例如，考虑使用具有 2KB 片内 EEPROM 的 Atmel 的 AT89LS8252，而不是使用一个基本的 8051 加上串行 EEPROM。

例子：I²C 内核库

本节给出了 I²C 内核程序库（源程序清单 23.1~源程序清单 23.3）。在大多数情况下，当把这些文件用于某个 I²C 外围模块时，需要增加一些函数。下面的例子说明了这种方法。

请注意，并不是所有的 I²C 芯片都使用 ACKNOWLEDGE/NOT ACKNOWLEDGE 信号（在前面讨论过）。与之类似，并不是所有芯片都需要“读取字节”功能（例如，脉宽调制和数模转换器输出）。可以通过 I²C_Core.C 文件的最前面几行省略这些库功能：

```
// 如果不需要这些函数，那么将这几行注释掉
//#define I2C_ACK_NACK
#define I2C_READ_BYTE
```

和往常一样，CD 上包含了这个项目的所有源文件。

```
-----*
Port.H (v1.00)
-----
I2C 内核库的“端口头文件”（参见第 10 章）
-----*/
// ----- I2C_Core.C -----
// 二线式的 I2C 总线
// 注意：如果使用外部上拉电阻，则通常将使用端口 0
sbit I2C_SCL = P1^7;
sbit I2C_SDA = P1^6;
/*----- 程序清单 -----*/
-----*
```

源程序清单 23.1 I²C 内核库的一部分

```
-----*
I2C_Core.H (v1.00)
-----
详细资料参见 I2C_Core.C
-----*/
#include "Main.h"
// --- 公用的函数原型 ---
void I2C_Send_Start(void);
void I2C_Send_Stop(void);
```

```
tByte I2C_Write_Byte(const tByte);
tByte I2C_Read_Byte(void);
void I2C_Send_Master_Ack(void);
void I2C_Send_Master_NAck(void);
// -----公用的常数-----
#define I2C_READ      0x01      // 读命令
#define I2C_WRITE     0x00      // 写命令
/*-----程序清单-----*/
/*-----*/
```

源程序清单 23.2 I²C 内核库的一部分

```
/*-----*
 * I2C_CORE.C (v1.00)
 *
 * I2C 库的内核
 * 通常需要其他模块来创建一个完整的库（例如，参见 I2C_ROM.C）
 *-----*/
#include "Main.h"
#include "Port.H"
#include "I2C_Core.h"
#include "TimeoutH.H"
// -----私有函数原型-----
static tByte I2C_Get_Ack_From_Slave(void);
static bit I2C_Sync_The_Clock_T0(void);
static void I2C_Delay(void);
// 如果不需要这些函数，那么将这几行注释掉
// #define I2C_ACK_NACK
/*-----*
 * I2C_Send_Start()
 * 产生 START 状态
 *-----*/
void I2C_Send_Start(void)
{
    // 准备总线
    I2C_SCL = 1;
    I2C_SDA = 1;
    I2C_Delay();
    // 产生 START 状态
    I2C_SDA = 0;
    I2C_Delay();
    I2C_SCL = 0;
}
/*-----*
 * I2C_Send_Stop()
 * 产生 STOP 状态
 *-----*/
void I2C_Send_Stop(void)
{
```



```

I2C_SDA = 0;
I2C_Delay();
I2C_SCL = 1;
I2C_Delay();
I2C_SDA = 1;
}
/*-----*/
I2C_Get_Ack_From_Slave()
这里实现“主从”通信协议，微控制器作为主机。这个函数等待从机的应答（含超时功能）
/*-----*/
tByte I2C_Get_Ack_From_Slave(void)
{
// 准备总线
I2C_SDA = 1;
I2C_SCL = 1;
if(I2C_Sync_The_Clock_T0())
{
    return 1;// 出错-同步失败
}
// 完成时钟同步
I2C_Delay();
if (I2C_SDA)
{
    // 产生时钟周期
    I2C_SCL = 0;
    return 1; // 出错-从机无应答
}
I2C_SCL = 0; // 产生时钟周期
return 0; // OK-从机有应答
}
/*-----*/
I2C_Write_Byte()
发送一个字节的数据到从机。通过允许“时钟延伸”来支持低速从机
运行时间~100 微秒（除非发生总线错误）
/*-----*/
tByte I2C_Write_Byte(tByte Data)
{
tByte Bit = 0;
// 每次发送一位数据（最高位优先）
for (Bit = 0; Bit < 8; Bit++)
{
    I2C_SDA = (bit)((Data & 0x80) >> 7);
    I2C_SCL = 1;

    if (I2C_Sync_The_Clock_T0())
    {
        return 1; // 出错-同步失败
    }
    I2C_Delay();
}
}

```

```

// 产生时钟周期
I2C_SCL = 0;
// 准备发送下一位
Data <= 1;
}
// 确信从机有应答
return(I2C_Get_Ack_From_Slave());
}
/*-----*/
I2C_Read_Byte()
从从机读取一个字节的数据。通过允许“时钟延伸”来支持低速从机
*/
tByte I2C_Read_Byte(void)
{
tByte result = 0; // 返回读取的 I2C 字节
tByte Bit = 0; // 位计数器
for (Bit = 0; Bit < 8; Bit++)
{
I2C_SDA = 1; // 释放 SDA
I2C_SCL = 1; // 释放 SCL
if (I2C_Sync_The_Clock_T0())
{
return 1; // 出错-同步失败
}
I2C_Delay();
result <= 1; // 结果左移
if (I2C_SDA)
{
result |= 0x01; // 最低位设置为实际的 SOA 状态
}
I2C_SCL = 0; // 强制时钟周期
I2C_Delay();
}
return(result);
}
/*-----*/
I2C_Sync_The_Clock_T0()
I2C 数据传输使用的底层函数
***有 1ms 硬件超时（定时器 0） ***
返回：1-出错（没有同步）
0-OK（时钟同步）
*/
bit I2C_Sync_The_Clock_T0(void)
{
// 将定时器 0 配置为 16 位定时器
TMOD &= 0xF0; // 清除所有有关 T0 的位 (T1 不变)
TMOD |= 0x01; // 设置所需的 T0 的位 (T1 不变)
ET0 = 0; // 不使用中断
// 简单的超时特性——大约 1ms
}

```

```

TH0 = T_01ms_H; // T_的详细资料参见 TimeoutH.H
TL0 = T_01ms_L;
TF0 = 0; // 清除标志
TR0 = 1; // 启动定时器
// 试图同步时钟
while ((I2C_SCL == 0) && (TF0 != 1));
TR0 = 0; // 停止定时器
if (TF0 == 1)
{
    return 1; // 错误——超时条件不满足
}
return 0; // OK——时钟同步
}

/*-----*
I2C_Delay()
较短的软件延迟(大约10微秒)
调整为最少5.425微秒,以连接标准的I2C设备。比这更长的延迟将同样可以正常工作。
现在的设备也可以使用较短的延迟
注意: 使用硬件延迟通常无法做到!
-----*/
void I2C_Delay(void)
{
    int x;
    x++;
    x++;
}
#endif I2C_ACK_NACK
/*-----*
I2C_Send_Master_Ack()
产生“ACKNOWLEDGE”状态
-----*/
void I2C_Send_Master_Ack(void)
{
    I2C_SDA = 0;
    I2C_SCL = 1;
    I2C_Sync_The_Clock_T0();
    I2C_Delay();
    I2C_SCL = 0;
}
/*-----*
I2C_Send_Master_NAck()
产生“NOT ACKNOWLEDGE”状态
-----*/
void I2C_Send_Master_NAck(void)
{
    I2C_SDA = 1; I2C_SCL = 1;
    I2C_Sync_The_Clock_T0();
    I2C_Delay();
    I2C_SCL = 0;
}

```

```

}
#endif
/*
-----程序清单-----
*/

```

源程序清单 23.3 I²C 内核库的一部分

例子：I²C EEPROM 接口

在这个例子中，考虑如何使用 I²C 总线连接 8051 微控制器和串行 EEPROM。注意，使用这样的 EEPROM 是在掉电之后保持系统设置的一种常用方法。EEPROM 的存储是非易失性的，可以保存 100 年左右。

硬件

所需的硬件如图 23.6 所示。

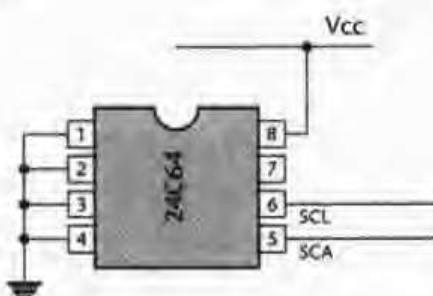


图 23.6 设备地址为“1010000”的 24C64

注意，在这个例子中，不使用集电极开路的端口并且不需要上拉电阻。如果使用多个 I²C 芯片，最好将代码修改为使用端口 0（很小的改变）并使用上拉电阻，电阻值的计算参见“背景知识”一节。

软件

所需的软件在源程序清单 23.4~源程序清单 23.6 中给出。注意，还需要用到“解决方案”一节中的 I²C 内核库。

```

/*
Main.C (v1.00)

I2C 库 (24C64) 的简单测试程序
按库文件 (I2C_Core.C) 中的说明将 24C64 连接到 SDA 和 SCL 引脚上通常总线上不需要终端电阻
将 EEPROM 上的所有三根地址线接地
*/
#include "Main.h"
#include "I2C_ROM.h"
#include "Delay_T0.h"

```

第 23 章 使用 I²C 外围模块

```
// 在这个测试程序中，在这里定义错误代码变量  
// (通常在调度器库中)  
tByte Error_code_G = 0;  
void main( void )  
{  
    tByte data1 = 0;  
    tByte data2 = 0;  
    tWord Data_address = 0;  
    while (1)  
    {  
        Data_address = data1;  
        // 写 EEPROM  
        I2C_Write_Byte_AT24C64(Data_address, data1);  
        // 从 EEPROM 读回  
        data2 = I2C_Read_Byte_AT24C64(Data_address);  
        // 显示 EEPROM 中的值  
        P1 = 255 - data2;  
        // 显示错误代码  
        P2 = 259 - Error_code_G;  
        if (++data1 == 255)  
        {  
            data1 = 0;  
        }  
        Hardware_Delay_T0(1000);  
    }  
}/*-----  
---程序清单-----  
-----*/
```

源程序清单 23.4 I²C EEPROM 库的一部分

```
/*-----  
I2C_ROM.H (v1.00)  
-----  
- 详细资料参见 I2C_ROM.C  
-*-----  
#include "Main.h"  
// ---公用的函数原型---  
// 从 EEPROM 读字节  
tByte I2C_Read_Byte_AT24C64(TWord);  
// 从 EEPROM 写字节  
void I2C_Write_Byte_AT24C64(tWord, tByte);  
/*-----  
---程序清单-----  
-----*/
```

源程序清单 23.5 I²C EEPROM 库的一部分

```
/*
I2C_ROM.C (v1.00)

用于二线式串行 EEPROM (AT24C64) 的 I2C 库函数
很容易扩展/修改以适用于其他二线式 EEPROM
*/
#include "Main.h"
#include "Delay_T0.h"
#include "I2C_Core.h"
#include "I2C_ROM.h"
// -----公有变量声明-----
extern tByte Error_code_G;
// -----私有常数-----
// 这个例子中使用的 EEPROM 的设备标识符
// 其他芯片使用的设备标识符参见正文或 Philips 文档
#define I2C_EEPROM_ID 0xA0
/*
I2C_Write_Byte_AT24C64()
发送一个字节的数据到 EEPROM
*/
void I2C_Write_Byte_AT24C64(const tWord address, tByte content)
{
    tByte MSByte; // 数据高位字节的地址
    tByte LSByte; // 数据低位字节的地址
    I2C_Send_Start(); // 产生 START 状态
    // 发送从机地址和写请求
    if (I2C_Write_Byte(I2C_EEPROM_ID | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
        return;
    }
    // 地址的高位字节
    MSByte = (address >> 8) & 0x00FF;

    // 地址的低位字节
    LSByte = address & 0x00FF;
    // 发送存储器地址
    if (I2C_Write_Byte(MSByte))
    {
        Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
        return;
    }
    // 发送存储器地址
    if (I2C_Write_Byte(LSByte))
    {
        Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
        return;
    }
    // 向存储器地址发送数据
```

```
if (I2C_Write_Byte(content))
{
    Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
    return;
}
I2C_Send_Stop();
return;
}

/*-----*
 * I2C_Read_Byte_AT24C64()
 * Read a byte of data from the EEPROM.
 * 从 EEPROM 读取一个字节的数据
 *-----*/
tByte I2C_Read_Byte_AT24C64(tWord address)
{
    tByte MSByte; // 数据高位字节的地址
    tByte LSByte; // 数据低位字节的地址
    tByte Result = 0;
    I2C_Send_Start(); // 产生 START 状态
    // 发送从机地址和空写请求
    if (I2C_Write_Byte(I2C_EEPROM_ID | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_READ_BYTE_AT24C64;
        return 0;
    }
    // 地址的高位字节
    MSByte = (address >> 8) & 0x00FF;
    // 地址的低位字节
    LSByte = address & 0x00FF;
    // 发送存储器地址
    if (I2C_Write_Byte(MSByte))
    {
        Error_code_G = ERROR_I2C_READ_BYTE_AT24C64;
        return 0;
    }
    // 发送存储器地址
    if (I2C_Write_Byte(LSByte))
    {
        Error_code_G = ERROR_I2C_READ_BYTE_AT24C64;
        return 0;
    }
    I2C_Send_Start(); // 产生 START 状态
    // 发送从机地址和读请求
    if (I2C_Write_Byte(I2C_EEPROM_ID | I2C_READ))
    {
        Error_code_G = ERROR_I2C_READ_BYTE_AT24C64;
        return 0;
    }
    Result = I2C_Read_Byte(); // 读存储器内容
```

```

    // 不产生主机应答
    I2C_Send_Stop();
    return(Result);
}
/*
-----文件结束-----
*/

```

源程序清单 23.6 I²C EEPROM 库的一部分

例子：I²C 温度传感器接口

在这个例子中，讨论如何连接 8051 微控制器和 Dallas 的 DS1621 (I²C) 温度传感器。

硬件

参见图 23.4。

软件

主要的软件文档在源程序清单 23.7~源程序清单 23.9 中给出。这个项目所需的全部文件都包含在 CD 上与本章有关的目录中。

```

/*
-----*
Main.C (v1.00)

-----*
I2C 库 (DS1621) 的简单的测试程序
按库文件 (I2C_Core.C) 中的说明将 DS1621 连接到 SDA 和 SCL 引脚上
通常总线上不需要终端电阻
-----*/
#include "Main.h"
#include "I2C_1621.h"
#include "Delay_T0.h"
tByte Temperature_G
// 在这个测试程序中，在这里定义错误代码变量
//(通常在调度器库中)
tByte Error_code_G = 0;
void main( void )
{
    while (1)
    {
        I2C_Write_Byte_AT24C64(Data_address, data1);
        // 从 EEPROM 读取
        data2 = I2C_Read_Byte_AT24C64(Data_address);
        // 显示从 EEPROM 读取的值
        P1 = 255 - Temperature_G;
        P2 = 255 - Error_code_G;
        Hardware_Delay_T0(1000);
    }
}

```

```
/*
---文件结束---
*/
```

源程序清单 23.7 I²C 温度传感器例子的一部分

```
/*
I2C_1621.H (v1.00)

-- 详细资料参见 I2C_1621.H
*/
#include "Main.h"
// -----公用的函数原型-----
void I2C_Read_Temperature_DS1621(void);
void I2C_Init_Temperature_DS1621(void);
/*
---文件结束---
*/
```

源程序清单 23.8 I²C 温度传感器例子的一部分

```
/*
I2C_1621.C (v1.00)

-- DS1621 温度传感器的基于 I2C 的库
*/
#include "Main.h"
#include "I2C_core.h"
#include "I2C_1621.h"
#include "Delay_T0.h"
// -----公有变量声明-----
extern tByte Temperature_G;
extern tByte Error_code_G;
// -----私有的常数-----
#define I2C_DS1621_ID 0x90
/*
I2C_Init_Temperature_DS1621()
设置传感器为“连续转换”模式，以便后面得到温度读数
*/
void I2C_Init_Temperature_DS1621(void)
{
    I2C_Send_Start(); // 产生 START 状态
    // 发送从机地址和写请求
    if (I2C_Write_Byte(I2C_DS1621_ID | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_DS1621;
        return;
    }
    // 发送控制字节：
    // 配置命令
}
```

```

if (I2C_Write_Byte(0xAC))
{
    Error_code_G = ERROR_I2C_DS1621;
    return;
}
// 发送配置数据-连续模式
if (I2C_Write_Byte(0x00))
{
    Error_code_G = ERROR_I2C_DS1621;
    return;
}
I2C_Send_Stop(); // 产生 STOP 状态
// 这里必须延迟以使传感器中的 EEPROM 保存这个数据，手册称需要 10ms
Hardware_Delay_T0(100);
// 现在开始温度转换
I2C_Send_Start(); // 产生 START 状态
// 发送从机地址和写请求
if (I2C_Write_Byte(I2C_DS1621_ID | I2C_WRITE))
{
    Error_code_G = ERROR_I2C_DS1621;
    return;
}
// 发送命令数据-开始转换
if (I2C_Write_Byte(0xEE))
{
    Error_code_G = ERROR_I2C_DS1621;
    return;
}
I2C_Send_Stop(); // 产生 STOP 状态
}

/*-----*
 * I2C_Read_Temperature_DS1621()
 * 传感器连续采样（大约每秒 1 个新数值）
 * 获得最新的值
 *-----*/
void I2C_Read_Temperature_DS1621(void)
{
    tByte result = 0;
    I2C_Send_Start(); // 产生 START 状态
    // 发送 DS1621 的设备地址（以及写访问请求）
    if (I2C_Write_Byte(I2C_DS1621_ID | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_DS1621;
        return;
    }
    // 发送命令-读取温度 (0xAA)
    if (I2C_Write_Byte(0xAA))
    {
        Error_code_G = ERROR_I2C_DS1621;
    }
}

```

```
    return;
}
I2C_Send_Start(); // 再次产生 START 状态
// 发送 DS1621 的设备地址 (以及读访问请求)
if (I2C_Write_Byte(I2C_DS1621_ID | I2C_READ))
{
    Error_code_G = ERROR_I2C_DS1621;
    return;
}
// 从 I2C 总线接收第一个 (高位) 字节
Temperature_G = I2C_Read_Byte();
I2C_Send_Master_Ack(); // 产生主机 ACK
// 这里只需要精确到 1 度的温度
// 去弃低位字节 (执行一次空读取)
I2C_Read_Byte();
I2C_Send_Master_NAck(); // 产生主机 NACK
I2C_Send_Stop(); // 产生 STOP 状态
}

/*-----*
 *-----文件结束-----*
 */
```

源程序清单 23.9 I²C 温度传感器例子的一部分

例子：I²C 模数转换器

I²C 接口的模数转换器的例子参见第 32 章。

进阶阅读

相关的详细资料参见 Philips 公司的 I²C 规约 (www.philips.com)。

Chapter 24

使用 SPI 外围模块

引言

使用诸如“RS-232”之类的异步串行协议时，互连的两个芯片必须使用相同的通信频率（波特率），每个芯片使用独立的基于晶振的时钟来保证以所需的频率运行。这种方法带来的后果是：如果发送接收芯片的时钟频率相差大于百分之几，接收设备将不能正确地对输入数据解码。

相比而言，串行外围接口（SPI）使用同步通信协议。这指的是发送和接收芯片共用一个公共时钟。这个公共时钟的跳变决定了什么时候发送和接收各个位。例如，在一个简单的同步协议中，发送装置在时钟的上升沿写入一位，而接收装置在时钟的下降沿读取这个位。注意，时钟频率不需要保持不变。

这样的同步接口主要设计在以厘米为单位的距离内使用，而不是以米为单位的距离。这里的讨论将限制为使用 SPI 连接 8051 和兼容 SPI 的外围模块。

SPI 外围模块

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用基于调度器的时间触发结构。
- 系统中的微控制器将连接多个外围模块，诸如键盘、EEPROM、数模转换器或者类似芯片。
- 微控制器硬件支持 SPI 协议。

问题

是否使用 SPI 总线来连接微控制器和外围设备，如果要使用，如何连接？

背景知识

就嵌入式系统开发人员而言，SPI 有五个主要特性：

- SPI 协议设计用来将微控制器连接到各种外围模块：存储器、显示、模数转换器，以及类似芯片。总线一般需要三个端口引脚，再加上每个外设需要一个片选引脚。
- 可以购买到许多现成的 SPI 兼容的外围模块。
- 越来越多的“标准”和“扩展”8051 芯片对 SPI 有硬件支持，将在这个模式中使用这样的功能。
- 所有 SPI 外围模块可以共用一组软件代码。
- SPI 和时间触发结构兼容，而且正如在本书中实现的，比 I²C 更快（主要由于使用了片内硬件支持）。典型的数据传输速率最高为 5000~10 000 字节/秒（采用 1ms 调度器时标时）。

本节提供了有关 SPI 的一些背景资料。

历史

串行外围接口（SPI）由 Motorola 开发，包含在 68HC11 和其他微控制器上。最近，这个接口标准已经被其他微控制器厂商采用。越来越多的“标准”和“扩展”8051 芯片（参见第 3 章）对 SPI 有硬件支持，将在这个模式中使用这样的功能。

基本的 SPI 操作

SPI 往往被称为三线式接口。实际上，几乎所有的实现都需要两根数据线、一根时钟线、一根片选线（通常每个外围设备一根）和一根公共接地线。这样至少为四根线，再加上地线。

数据线被称为 MOSI（主出从入）和 MISO（主入从出）。

SPI 的整个操作很容易理解，只要记住该协议基于使用两个 8 位移位寄存器，一个在主机中，另一个在从机中（如图 24.1 所示）。

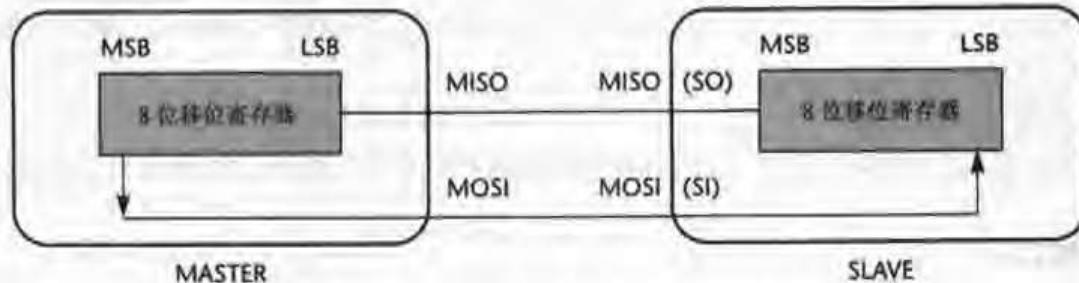


图 24.1 SPI 协议的核心是两个 8 位移位寄存器

注意，一些从机（例如 EEPROM）的数据线标记为 SI（从入）和 SO（从出）。从机上的 SI 线连接到主机上的 MOSI 线，而从机上的 SO 线连接到主机上的 MISO 线。

SPI 的关键操作是在主机和当前选定的从机之间传输一个字节的数据。与此同时，一个字节的数据将由从机回传到主机。

单主机、多从机

SPI 是一种单主机、多从机接口，主机产生时钟信号。就这里所讨论的而言，微控制器将作为主机而外围设备将作为从机。

选择时钟的极性

SPI 支持两种时钟极性。极性为 0 时，时钟线在静态时为低。在使用过程中，发送的数据在时钟的上升沿写入，而在时钟的下降沿读出数据；极性为 1 时，时钟线在静态时为高。在使用过程中，发送的数据在时钟的下降沿写入，而在时钟的上升沿读出数据。

极性 0 已被更为广泛使用。

最高时钟频率

目前，SPI 的最高时钟频率为 2.1MHz。考虑到需要花费八个时钟周期来传送一个字节的数据，而且同时有其他开销（例如指令和地址），最高数据传输速率大约为 130 000 字节/秒。

Microwire

注意，Microwire 接口标准（由 National Semiconductor 开发）和 SPI 类似，但在接线的名称、极性及其他详细规范上有所不同。

本书中没有对 Microwire 做更进一步的讨论。

解决方案

是否应使用 SPI？

为了确定在时间触发系统中使用 SPI 总线是否合适，下面讨论一下当考虑使用任何通信协议或者相关技术时应该关心的一些主要问题。

主要应用领域

虽然 SPI 总线可以用来连接处理器（通常为微控制器）到另一个处理器或者其他计算机系统，然而和 I²C 一样，其主要应用领域是将微控制器连接到标准外围设备，诸如 LCD 显示面板或者 EEPROM。

易于开发

SPI 可用来与大量的外设通信。通过使用同样的协议与一系列芯片通信，可以减少研制的工作量。

可伸缩性

每个 SPI 从机需要主机节点提供一个单独的/CS（片选）线。如果使用大量外围模块，这将增加微控制器所需的引脚数。

灵活性

每个 SPI 兼容的微控制器都既可以作为主机也可以作为从机。在这个模式中，只考虑使用微控制器作为主机节点。

运行速度和代码长度

目前，SPI 的最高时钟频率为 2.1MHz。

因为在这个模式中将使用基于硬件的 SPI，所以代码开销很小。

成本

SPI 总线的许可证费用包含在购买的外设模块的费用中。在大多数情况下，不再需要其他费用。

注意，如果要设计连接到 SPI 总线上的 SPI 外设（用于出售），可能需要额外的费用。如有疑问，请联系 Motorola。

实现芯片和供应厂商的选择

这里给出的 SPI 库只能用于具有 SPI 硬件支持的 8051 芯片。

适用于时间触发系统

正如在第 18 章看到的，RS-232 通信协议适用于时间触发系统。这是因为和在 RS-232 网络上发送（以及接收）数据的有关任务的运行时间非常短。注意，传输时间与网络的波特率没有直接关系，这主要是因为几乎所有的 8051 系列芯片都有对 RS-232 的硬件支持，因此信息“在后台”发送与接收。

SPI 的情况与之类似。具体地说，在这个模式中，将关注基于硬件的 SPI 协议。一般将占用很少的软件开销，并且任务的运行时间很短。例如，考虑发送一个字节数据到基于 SPI 的 ROM 芯片的过程（在后面将给出完整的例子），任务的总运行时间为 0.1ms 左右。注意，这比使用本书给出的 I²C 库相对应的操作快得多。

即使是采用 1ms 的定时器时标，时间触发系统也可以很容易地支持这样的任务运行时间。

如何在时间触发系统中使用 SPI？

此处将集中讨论 Atmel 的 AT89S53——一种有片内 SPI 支持的标准 8051 芯片。注意，其他厂商提供的硬件支持与之非常类似。

AT89S53 的 SPI 包含以下特性：

- 全双工、三线式同步数据传送
- 主机或从机操作
- (最高) 1.5MHz 波特率
- 最低位优先或最高位优先数据传输
- 四个可编程波特率
- 传输结束中断标志
- 写冲突标志保护
- 从空闲模式唤醒（只用于从机方式）

主机和从机 CPU 之间的 SPI 互连如图 24.1 所示。

在主机方式下，SCK 引脚是时钟输出（在这里将只使用主机方式）。向主机 CPU 的 SPI

数据寄存器写入将启动 SPI 时钟发生器，写入的数据将移出 MOSI 引脚并且进入从机 CPU 的 MOSI 引脚。在移出一个字节后，SPI 时钟发生器将停止，并设置传输结束标志（SPIF）。如果将 SPI 的中断允许位（SPIE）和串行端口中断允许位（ES）都设置为允许，则将产生中断请求。本书没有使用这样的中断功能。

在从机方式下，从机选择输入 SS/P1.4 为低时，选择该 SPI 芯片。当 SS/P1.4 为高时，将停止使用 SPI 端口，MOSI/P1.5 引脚可用作输入。

相对于串行数据，SCK 的相位和极性有四种组合，它们取决于控制位 CPHA 和 CPOL（参见表 24.1）。

表 24.1 Atmel 的 89S53 的 SPI 控制寄存器（SPCR）

位	名称	概述
7	SPIE	SPI 中断允许。在这里未使用 通常设置 SPIE=0
6	SPE	SPE=1 使能该 SPI 通道。注意，这意味着端口 1 的高四位不能用作通用的输入/输出 通常设置 SPE=1
5	DORD	数据顺序。DORD=0 选择“最高位优先”数据传输； DORD=1 选择“最低位优先”数据传输； 通常设置 DORD=0
4	MSTR	主机/从机选择。MSTR=1 选择主机 SPI 模式 这里将只使用主机方式：设置 MSTR=1
3	CPOL	时钟极性。当 CPOL=0 时，主机的 SCK 在无数据发送时为低；当 CPOL=1 时，SCK 在这种情况下为高。 通常设置 CPOL=0
2	CPHA	时钟相位。详细资料参考 Atmel 的文档。 通常设置 CPHA=0
1	SPR1	时钟频率选择。当芯片配置为主机时，这两个位控制 SCK 的频率
0	SPR0	SCK 和振荡器/谐振器频率（“Osc”）之间的关系如下：
		SPR1 SPR0 SCK
		0 0 Osc/4
		0 1 Osc/16
		1 0 Osc/64
		1 1 Osc/128
注意 AT89S53 具有最高为 1.5MHz 的（SPI）波特率		

AT89S53 的 SPI 的大多数特性将在下面的例子中说明。

硬件资源

SPI 模块有片内硬件支持，只占用极少的软件开销。

可靠性和安全性

SPI 协议只加入了很多的错误检测机制。如果需要的话，检测外围设备传输的数据是否损坏必须在软件中执行。

可移植性

这个模式需要对 SPI 的硬件支持，它不能用于不提供这种支持的微控制器。

这里的讨论基于 Atmel 的 89S53，使用其他 8051 微控制器（包括许多 Infineon 的 8051）也很简单。

优缺点小结

- ⊕ 有大量的外围设备支持 SPI。
- ⊕ SPI 总线一般需要三个端口引脚，且每个外设需要一个片选引脚。
- ⊕ 使用基于硬件的 SPI（正如这里讨论的）将使任务具有较短的运行时间。因此，该协议很适合时间触发系统的需要。典型的数据传输速率最高为 5000~10 000 字节/秒（采用 1ms 调度器时标时）。
- ⊕ 所有 SPI 外围模块均可以共用一组软件代码。
- ⊕ 这个模式的使用限制于有 SPI 硬件支持的微控制器。

相关的模式和替代解决方案

这个模式的使用限制于有 SPI 硬件支持的微控制器。

不需要硬件支持而提供非常类似功能的替代方案参见 I²C 外设。

例子：SPI 内核库

SPI 的实现由 SPI “内核” 和为满足某个外设（诸如某种 EEPROM）的需要而添加的附加库组成，内核在源程序清单 24.1~源程序清单 24.3 中给出。

```
/*
Port.H (v1.00)

-----*
----- SPI 内核库的“端口头文件”（参见第 10 章）
-----*/
// ----- Sch51.C -----
// 如果不需要错误报告，将这一行注释掉
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1
```

```
#endif
// ----- SPI_Core.C -----
// 这里创建 sbits 用于所有需要的芯片选择
sbit SPI_CS = P1^4;
// 注意：还需要使用引脚 P1.4、P1.5、P1.6 和 P1.7-参见正文
/*
-----文件结束-----
*/

```

源程序清单 24.1 Atmel 的 AT89S53 的 SPI 内核库的一部分

```
/*
  SPI_Core.H (v1.00)
  -----
  详细资料参见 SPI_Core.C
*/
#include "Main.h"
// -----公用的函数原型-----
void SPI_Init_AT89S53(const tByte);
tByte SPI_Exchange_Bytes(const tByte);
/*
-----文件结束-----
*/

```

源程序清单 24.2 Atmel 的 AT89S53 的 SPI 内核库的一部分

```
/*
  SPI_Core.C (v1.00)
  -----
  Atmel 的 AT89S53 的 SPI 内核库
*/
#include "Main.h"
#include "Port.h"
#include "SPI_Core.h"
#include "TimeoutH.H"
// -----公有变量声明-----
// 错误代码变量
//
// 用来显示错误代码的端口以及错误代码的详细说明参见 Main.H
extern tByte Error_code_G;
/*
  SPI_Init_AT89S53()
  设置片内 SPI 模块
*/
void SPI_Init_AT89S53(const tByte SPI_MODE)
{
    // SPI 控制寄存器 (SPCR)
    // 位 7=SPIE (如果 ES 也为 1, 使能 SPI 中断)
    // 位 6=SPE (使能 SPI)
    // 位 5=DORD (数据顺序, 1 为最低位优先, 0 为最高位优先)
}
```

```

// 位 4=MSTR (1 为主机, 0 为从机)
// 位 3=CPOL (时钟极性, 1=当空闲时为高, 0=当空闲时为低)
// 位 2=CPHA (传输格式)
// 位 1=SPR1 (SPR0、SPR1 控制时钟频率)
// 位 0 = SPR0
SPCR = SPI_MODE;
}

/*-----*
 * SPI_Exchange_Bytes()
 * 与从机交换一个字节的数据
-*-----*/
tByte SPI_Exchange_Bytes(const tByte OUT)
{
    // 向 SPI 寄存器写入字节 (启动时钟)
    // 这些数据将被传送到从机
    SPDR = OUT;
    // 等待直到字节传输完成, 设置 5ms 超时-开始
    // 将定时器 0 配置为 16 位定时器用于超时
    TMOD &= 0xF0; // 清除所有有关 T0 的位 (T1 不变)
    TMOD |= 0x01; // 设置需要的 T0 位 (T1 不变)
    ET0 = 0; // 不使用中断
    // 简单的超时特性——大约 5ms
    TH0 = T_05ms_H; // T_ 的详细资料参见 TimeoutH.H
    TL0 = T_05ms_L;
    TF0 = 0; // 清除标志
    TR0 = 1; // 启动定时器
    while (((SPSR & SPIF) == 0) && (!TF0));
    TR0 = 0;
    if (TF0 == 1)
    {
        // SPI 芯片超时
        Error_code_G = ERROR_SPI_EXCHANGE_BYTES_TIMEOUT;
    }
    // 清除 SPIF 和 MCOL
    SPSR &= 0x3F;
    // 返回 SPI 寄存器中的内容
    // 这些是从从机收到的数据
    return SPDR;
}
/*-----*
 * -----文件结束-----
-*-----*/

```

源程序清单 24.3 Atmel 的 AT89S53 的 SPI 内核库的一部分

例子：使用 SPI 接口 EEPROM (X25320 或者类似芯片)

在这个例子中，给出一个和外部 EEPROM 通信的 SPI 库。在这个例子中，使用 X25320

(4k x8位)芯片，任何类似的SPI接口EEPROM都可以很容易地使用。这样的芯片用作存储诸如口令等类似的非易失性信息时非常有用。

硬件基于Atmel的89S53(参见图24.2)。

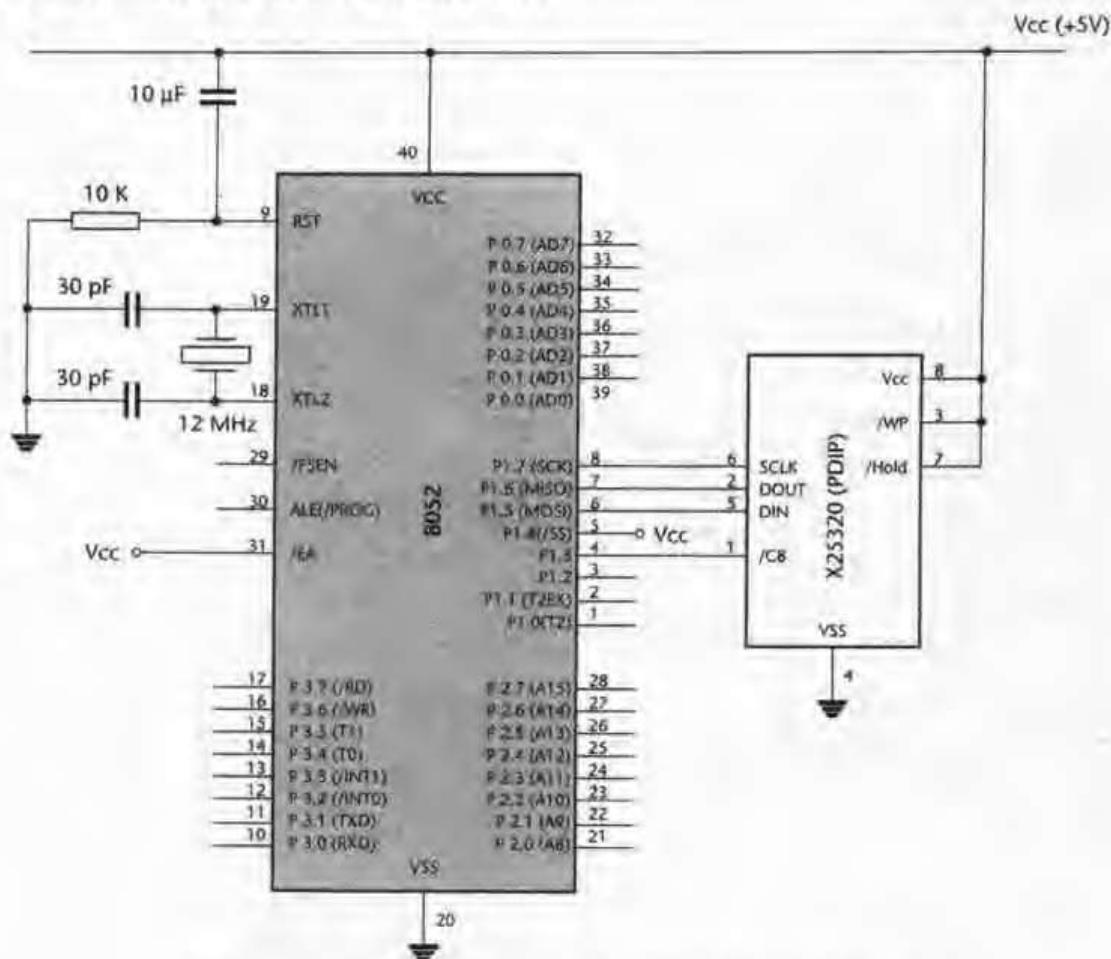


图24.2 将SPI接口EEPROM连接到Atmel的AT89S53上

注意，在main()中执行以下操作：

```
SPI_Init_AT89S53(0x51);
```

在这个例子中，电路中使用12MHz的晶振，SPI时钟频率设置为750 000位/秒。这大致上是100字节/毫秒，意味着基本的数据传输任务的运行时间为0.01ms。

主要的文件在源程序清单24.4~源程序清单24.6中给出，此外，还需要前面给出的SPI内核文件。和前面一样，CD上包含了这个项目的所有源文件。

```
/*
Main.C (v1.00)

SPI代码库的简单测试程序，读写xicor的X25320(4k x8位)EEPROM
*/

```

```

#include "Main.h"
#include "SPI_Core.h"
#include "SPI_X25.h"
#include "Delay_T0.h"
// 在这个测试程序中，在这里定义错误代码变量
tByte Error_code_G = 0;
void main(void)
{
    tByte Data1 = 0;
    tByte Data2 = 0;
    tWord Data_address = 0;
    // 详细资料参见正文
    SPI_Init_AT89S53(0x51);
    while (1)
    {
        // 写 EEPROM
        SPI_X25_Write_Byte(Data_address, Data1++);
        // 从 EEPROM 回读
        Data2 = SPI_X25_Read_Byte(Data_address);
        // 显示 EEPROM 中的值
        P2 = 255 - Data2;
        // 显示错误代码（如果有）
        P3 = 255 - Error_code_G;
        if (++Data_address == 4095)
        {
            Data_address = 0;
        }
        Hardware_Delay_T0(1000);
    }
}
/*-----*
---文件结束-----
*/

```

源程序清单 24.4 将 SPI 接口 EEPROM 连接到 Atmel 的 AT89S53 上的例子的一部分

```

/*-----*
  SPI_x25.H (v1.00)
-----*/
// 详细资料参见 SPI_x25.C
/*-----* */
#include "Main.h"
// -----公用的函数原型-----
void SPI_X25_Write_Byte(const tWord, const tByte);
tByte SPI_X25_Read_Byte(const tWord);
/*-----*
---文件结束-----
*/

```

源程序清单 24.5 将 SPI 接口 EEPROM 连接到 Atmel 的 AT89S53 上的例子的一部分

用于 Atmel 的 AT89S53 的简单的 SPI 库，在 Xicor 的 X25138 EEPROM（或类似芯片）上存储数据。

```
/*
  SPI_X25.C (v1.00)

  Atmel 的 AT89S53 的简单的 SPI 库
  -使得数据存储在 Xicor 的 X25138 EEPROM(或类似芯片)上
*/
#include "Main.H"
#include "Port.h"
#include "SPI_Core.h"
#include "SPI_X25.h"
#include "TimeoutH.h"
// -----公有变量声明-----
// 错误代码变量
//
// 用来显示错误代码的端口以及错误代码的详细说明参见 Main.H
extern tByte Error_code_G;
// -----私有函数原型-----
void SPI_Delay_T0(void);
void SPI_X25_Read_Status_Register(void);
/*-----*
  SPI_X25_Write_Byte()
  在 EEPROM 上存储一个字节的数据
-*-----*/
void SPI_X25_Write_Byte(const tWord ADDRESS, const tByte DATA)
{
    // 0. 检查状态寄存器
    SPI_X25_Read_Status_Register();
    // 1. 拉低/CS 引脚来选择该芯片
    SPI_CS = 0;
    // 2. 发送“写允许”指令 (0x06)
    SPI_Exchange_Bytes(0x06);
    // 3. 现在/CS 必须拉高
    SPI_CS = 1;
    // 4. (短暂的) 等待
    SPI_Delay_T0();
    // 5. 拉低/CS 引脚来选择该芯片
    SPI_CS = 0;
    // 6. 发送“写允许”指令 (0x02)
    SPI_Exchange_Bytes(0x02);
    // 7. 发送想要读取的地址
    // 注意, 发送一个 16 位地址:
    // 取决于芯片的字长, 可以忽略一些位
    SPI_Exchange_Bytes((ADDRESS >> 8) & 0x00FF);      // 发送最高位
    SPI_Exchange_Bytes(ADDRESS & 0x00FF);                // 发送最低位
    // 8. 写入的数据从 MOSI 引脚上移出
    SPI_Exchange_Bytes(DATA);
```

```

// 9. 拉高/CS 引脚来完成操作
SPI_CS = 1;
}

/*-----*/
SPI_X25_Read_Byte()
从 EEPROM 读取一个字节的数据
/*-----*/
tByte SPI_X25_Read_Byte(const tWord ADDRESS)
{
    tByte Data;
    // 0. 检查状态寄存器
    SPI_X25_Read_Status_Register();
    // 1. 拉低/CS 引脚来选择该芯片
    SPI_CS = 0;
    // 2. 发送“读”指令 (0x03)
    SPI_Exchange_Bytes(0x03);
    // 3. 发送想要读取的地址
    // 注意，发送一个 16 位地址：
    // 取决于芯片的字长，可以忽略一些位
    SPI_Exchange_Bytes((ADDRESS >> 8) & 0x00FF);
    SPI_Exchange_Bytes(ADDRESS & 0x00FF);
    // 4. 通过发送一个填充字节将请求的数据移出 SO
    Data = SPI_Exchange_Bytes(0x00);
    // 5. 拉高/CS 引脚来完成操作
    SPI_CS = 1;
    return Data; // 返回 SPI 数据字节
}
/*-----*/
SPI_X25_Read_Status_Register()
读取状态寄存器，检查 WIP（“正在写入”）标志，确信前面的“写”操作（如果有的话）已经完成
***NB：内部 EEPROM 写入操作需要 10ms ***
***以适当的间隔调度写入（以及写入后的读取）以避免任务抖动 ***
在~15ms 之后超时。使用 T0 来产生（硬件）超时——参见第 15 章
/*-----*/
void SPI_X25_Read_Status_Register()
{
    tByte Data;
    bit Wip;
    // 将定时器 0 配置为 16 位定时器
    TMOD &= 0xF0; // 清除所有有关 T0 的位 (T1 不变)
    TMOD |= 0x01; // 设置所需的 T0 的位 (T1 不变)
    ET0 = 0; // 不使用中断
    // 简单的超时特性——大约 15ms
    TH0 = T_15ms_H; // T_ 的详细资料参见 TimeoutH.H
    TL0 = T_15ms_L;
    TF0 = 0; // 清除标志
    TR0 = 1; // 启动定时器
    do {
        // 0. 拉低/CS 引脚来选择该芯片

```

```

SPI_CS = 0;
// 1.发送“RDSR”指令(0x05)
SPI_Exchange_Bytes(0x05);
// 2.读取ROM的状态寄存器的内容
Data = SPI_Exchange_Bytes(0x00);
// 3.拉高/CS引脚来完成操作
SPI_CS = 1;
// 检查WIP标志
Wip = (Data & 0x01);
} while ((Wip != 0) && (TF0 != 1));
TR0 = 0;
if (TF0 == 1)
{
    // ROM超时
    Error_code_G = ERROR_SPI_X25_TIMEOUT;
}
}

/*-----*
 * SPI_Delay_T0()
 * 使用“超时”代码产生大约50微秒的延迟
-*-----*/
void SPI_Delay_T0(void)
{
    // 将定时器0配置为16位定时器
    TMOD &= 0xF0; // 清除所有有关T0的位(T1不变)
    TMOD |= 0x01; // T0需要的位置1(T1不变)
    ET0 = 0; // 不使用中断
    // 简单的超时特性——大约50微秒
    TH0 = T_50micros_H; // T_的详细资料参见TimeoutH.H
    TL0 = T_50micros_L;
    TF0 = 0; // 清除标志
    TR0 = 1; // 启动定时器
    while (!TF0);
    TR0 = 0;
}

*/

```

源程序清单 24.6 将 SPI 接口 EEPROM 连接到 Atmel 的 AT89S53 上的例子的一部分

例子：使用 SPI 接口模数转换器

使用基于 SPI 的模数转换器参见第 32 章。

进阶阅读

Part 6

多处理器系统的 时间触发体体系结构

在第 6 篇中，将主要讨论多处理器系统。时间触发（合作式）调度体系结构的一个重要优点是具备固有的可伸缩性而且可以自然地推广应用至多处理器环境。

在第 25 章中，讨论了使用多处理器的一些优点和缺点，然后介绍了共享时钟调度器，并说明了怎样用共享时钟调度器建立包含两个或更多微控制器的时间触发系统。

在第 26 章，讨论了通过从节点微控制器的外部中断保持同步的共享时钟调度器。这个简单的调度器只有很少的存储器、CPU 或者硬件资源开销。然而，这一体系通常只适用于系统样机或者主节点微控制器，以及从节点微控制器位于同一电路板的情况。

在第 27 章，将详细讲述建立共享时钟调度器的方法，这一技术可用于远程连接多个控制器，使用 RS - 232、RS - 485 协议和相应的收发器。另外，还示范了这一方法在短距离内的应用，此时可不用任何收发器元件。

最后，在第 28 章，讨论了使用强大的 CAN 总线（控制器区域网总线）技术的共享时钟调度器。CAN 总线广泛地应用于汽车业、工业和其他领域。对于需要高可靠性的多处理器系统，特别是需要在网络上传输较多数据的应用，CAN 总线是一个优秀的平台。和基于 UART（通用异步收发器）的技术一样，CAN 协议同时适用于本地和分布式系统。



共享时钟调度器的介绍

本章将讨论嵌入式系统的另外一个重要的特征：多处理器体系结构的应用。在后文中将看到，前几章引入的调度器体系结构可以轻易地应用到多处理器体系结构上。

25.1 引言

前面的章节讲述过的嵌入式系统具有丰富的多样性，但是都只包含一个微控制器。对比之下，许多现代的嵌入式系统包含多个处理器，例如，现代的客车可能包含 40 个微处理器 (Leen 等, 1999)，控制着车闸、车门窗、后视镜、转向系统和安全气囊等等。类似地，工业火警探测系统一般可能包含 200 个以上连接在一起的处理器，例如，多种不同的传感器和传动装置。

本章将从多处理器系统的两个关键优点说起，接着介绍一种合作式调度器——共享时钟调度器，这种调度器能够帮助开发人员最大程度地利用多处理器设计的优势。

最后，本章将以关于多处理器可靠性要求的讨论结束。

25.2 额外的CPU性能和外围硬件

假设需要以下规格的微控制器：

- 超过 60 根的端口引脚
- 6 个定时器
- 2 个 USART（通用同步/异步收发器）
- 128KB 的 ROM（只读存储器）
- 512 字节的 RAM（随机存取存储器）
- 成本大约为 1 美元

用扩展 8051 微控制器（参见第 3 章的介绍）可以满足这些要求。然而，成本一般至少是所要求的 (\$1.00) 5~10 倍，图 25.1 中的“微控制器”差不多能满足这些要求。

图 25.1 中，两个标准的 8051 微控制器通过一个端口引脚连接起来：正如在 SCI 调度器（时标）模式（第 26 章）中所演示的，建立这种调度器只需要非常小的软件和硬件开销。这种方案建立了一个灵活的硬件平台，有 62 个可用的端口引脚，5 个可用的定时器，2 个 USART 等等。注意，可以很容易地加入更多的微控制器，而且单线通信（再加上地线）能保证各个处理器上的任务完美地同步在一起。

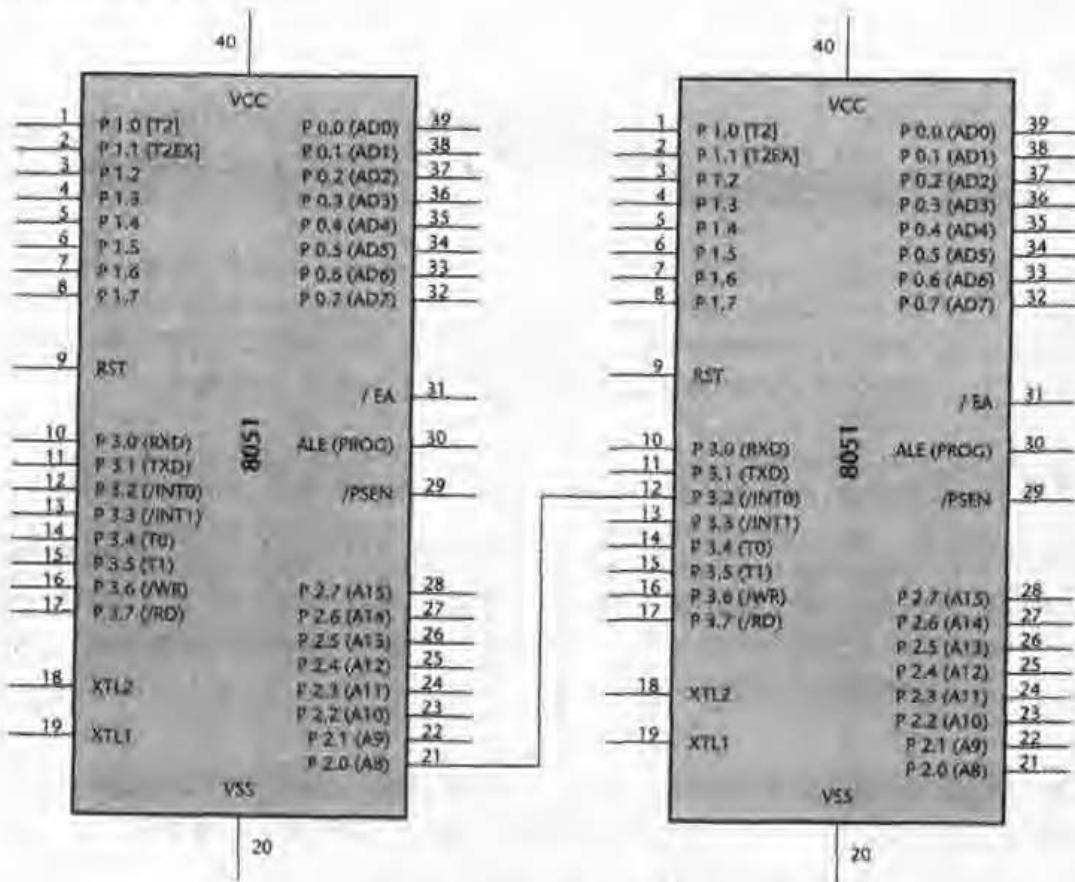


图 25.1 用简单的共享时钟调度器连接两个 8051 微控制器

当然，除了列出的这些特点外，双微控制器设计的另一个特点是有两个 CPU。在很多（但不是所有的）情况下，这样可以更快地执行任务，或者在给定的时间间隔内执行更多的任务。

长任务模式（第 29 章）和多米诺骨牌任务模式有时与数据联合模式（第 29 章）一起使用，有效的软件封装体系可以使多处理器设计达到最佳的性能。

25.3 模块化设计的优点

假设需要生产多种不同的时钟，这些时钟具有各种类型的显示（如图 25.2 所示）。

一些时钟可能具有不同的特色（例如，能够设置闹铃），但是无论哪种情况，最关键的任务都是保持精确的计时和显示计时信息。



图 25.2 各种时钟产品的不同显示

在某些情况下，最好将功能分布在两个模块上，每个功能使用一个单独的微控制器实现。第一个模块处理基本的计时和时间调整功能；第二个模块为不同形式的显示提供支持，诸如 LCD（液晶）驱动器或者步进电机。这种方法能够提供经济上的好处，因为能够以低廉的成本大规模生产成千上万的基本计时模块。另外再为满足不同顾客的需要，生产不同的显示部件。

这种模块化方法普遍应用于汽车工业，在新车设计中，越来越多地使用基于微控制器的模块。

再看另一个例子。假设有一个由单个处理器和许多分布式（简单的）传感器组成的数据采集系统（如图 25.3 所示）。

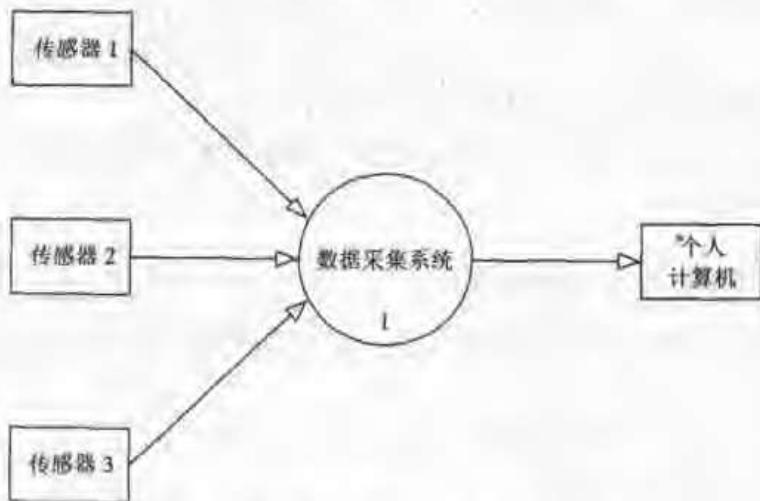


图 25.3 一个简单的数据采集系统的概略设计

在这个方案中，如果到传感器 1 的电缆坏了，不修复连接就无法从传感器获得任何数据；更糟糕的是，如果使用了不恰当的数据表示方法，数据采集系统甚至无法发现连接已经损坏了。

现在看一下使用智能传感器的替代方案（如图 25.4 所示）。

在这个替代方案中，（假定）传感器 1 很靠近微控制器（微控制器 A）；这两个器件一起组成了智能传感器。智能传感器和主数据采集系统间的通信系统可以很容易地检测到链路的断开。有了这个设计，当通信链路断开时传感器可以继续搜集数据，修好通信连接后，主数据采集板便可以恢复“丢失”的数据。

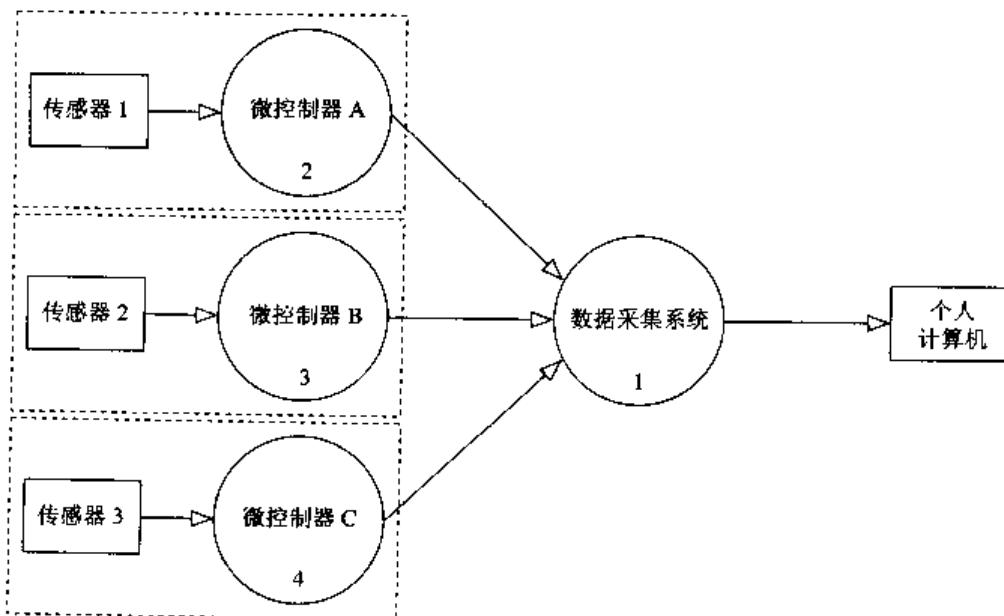


图 25.4 智能数据采集系统的概略设计

这种智能节点在许多系统中都很有用。例如，在 A310 空中客车飞机中，前缘缝翼和襟翼操纵计算机构成了一个智能传动子系统。如果在着陆期间发现了错误，机翼将被置于安全状态，而传动子系统会自动关闭 (Burns 和 Wellings, 1997, p.102)。

在第 6 篇剩下的章节中将看到，大多数共享时钟调度器都支持后备节点，如果需要的话，后备节点也可以是智能的。

25.4 怎样连接多个处理器

下面开始讨论在设计多处理器系统时将面对的一些挑战，从一个基本的问题出发：

- 如何保持各节点的时钟同步？

接着是两个许多这类系统都会面临的问题：

- 如何在各节点间传输数据？
- 一个节点如何检测其他节点是否发生了错误？

在后面将看到，使用共享时钟 (S-C) 调度器可以解决上述三个问题。此外，其中使用的时分多路存取 (TDMA) 协议是本书前面章节中讨论的单处理器系统时间触发体系的自然延伸 (Burns 和 Wellings, 1997, p.484)。

时钟的同步

为什么需要同步在多处理器系统不同部分运行的任务？

看一个简单的例子。假设需要设计一个移动式交通信号灯系统，控制一条正在修理的狭窄道路的交通。交通信号灯系统将用在道路的两端，在任何时刻，都只允许单方向通行（如图

25.5 所示)。



图 25.5 移动式交通信号灯控制系统

传统的红 (R) 黄 (A) 绿 (G) 灯将以通常的排列顺序用于每个节点 (如图 25.6 所示)。

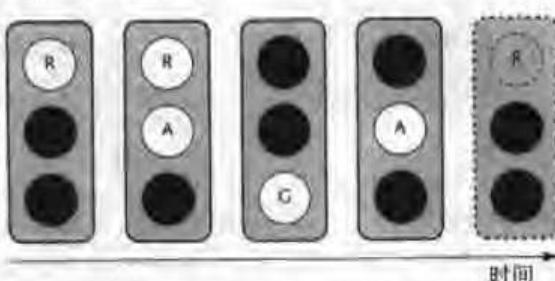


图 25.6 交通信号灯例子中的信号灯顺序

假定在道路的两端各有一个微控制器控制两组交通信号灯，并假定每个微控制器运行一个由相互独立的晶体振荡器电路驱动的调度器——这个方案的问题是两个微控制器的调度器可能很快就失去同步。主要原因是两块电路板不可能在完全相同的温度下运行，因此晶体振荡器的振荡频率会略有不同。

这在实际应用中会造成问题。在这个例子中，会有两套交通信号灯同时为绿的风险，结果很可能是一起交通事故。

共享时钟调度器通过在不同处理器板之间共享单个时钟来解决这个问题，其原理如图 25.7 所示。

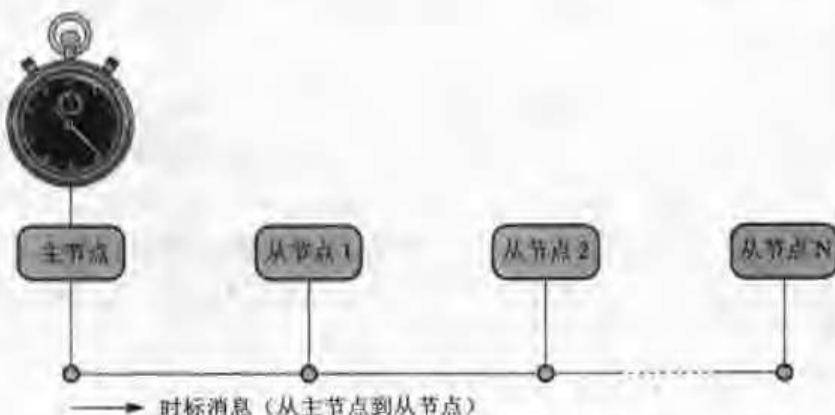


图 25.7 共享时钟调度器的最简单形式

注意，其中包含每隔一定间隔从主节点发送到从节点的时标消息，所有共享时钟调度器都有此部分。

这样，在网络上，主节点处有一个精确的时钟。这个时钟按第三篇中讲述的方式驱动主节点的调度器^①。

从节点也有调度器。然而，用于驱动这个调度器的中断来源于主节点产生的时标消息（图 25.8）。这样，在基于 CAN 的网络中（例如），从节点的共享时钟调度器将由接收到主节点数据而产生的接收中断驱动。

就交通信号灯的例子来说，在最坏的情况下，温度的变化或快或慢总会改变信号灯的周期，最终使两组信号灯失去同步。

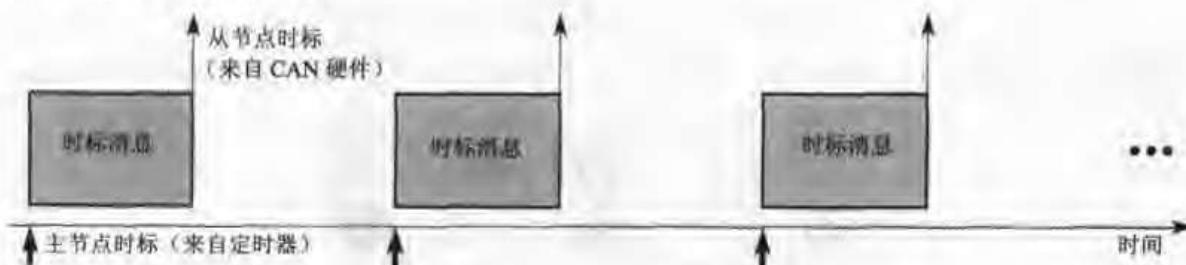


图 25.8 最简单的共享时钟网络（基于 CAN）中主节点和从节点间的通信

注意，时标消息的传输由主节点的定时器触发。从节点收到的规则的时标消息是驱动其调度器的中断的来源，从节点并不另外再用定时器产生中断。在某些网络中，主节点和从节点间的时标存在轻微的延迟（特别是基于 CAN 和 UART 的网络）。但是这个延迟很短（大多数情况下不到 1ms）。重要的是，延迟是可预测和固定不变的。第 27 章和第 28 章中将讨论如何计算和在必要时消除这个延迟。

传输数据

迄今为止，本章的讨论都集中在各个节点的调度器的同步。在许多系统中，还需要在运行于不同处理器节点上的任务间传输数据。

为了说明这一点，再看一下交通信号灯控制器。假设其中一组信号灯的一个灯泡烧坏了。当有一个信号灯不能亮时，交通信号是含糊不清的。因此，需要检测每个节点以确定是否有灯泡发生了故障，如果检测出了故障，则通知另一个节点有故障发生，从而能够采取一些应对措施，例如，熄灭两个节点所有的灯泡或者闪烁所有的灯泡，两种情况都能通知路人信号灯出了问题，必须小心通过该道路。

如果灯泡的故障是在主节点一侧检测到的，处理就十分简单。正如以前所讨论过的，主节点规则地向从节点发送时标消息，一般每毫秒一次。在大多数共享时钟调度器中，这个时标消息可以传输数据，所以可以简单地发送一个消息到从节点通知灯泡发生了故障。

为了从从节点传输数据到主节点，需要另外的机制：通过使用确认消息实现（如图 25.9 所示）。最终的结果是一个简单且可预测的时分多路存取(TDMA)协议（参见 Burns 和 Wellings, 1997），确认消息和时标消息在网络上交错传输。例如，在图 25.10 中，时标消息和确认消息在一个典型的两个从节点的(CAN)网络上混合传输。

^① 必要时，温度补偿或者卫星时钟可用于主节点以确保整个网络的精确定时。参见第 4 章。

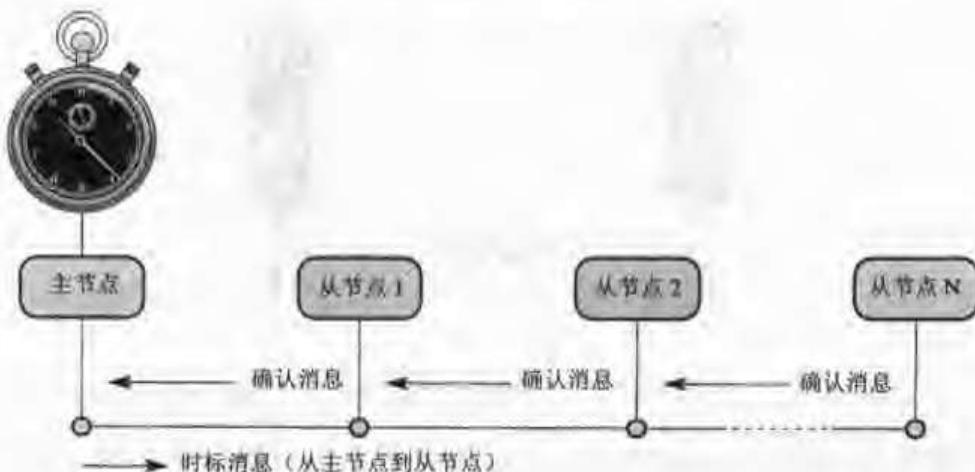


图 25.9 大多数共享时钟调度器同时支持时标消息（从主节点发送至从节点）和确认消息（从从节点发送至主节点）



图 25.10 一主二从三个节点的共享时钟 (CAN) 调度器的主从节点间通信

注意，时标消息更新所有从节点的调度器时标。然而，每个时标消息只需要一个确认消息，这个消息将在下一次时标消息前发送。这样便满足了简单且可预测的 (TDMA) 网络协议要求。

注意，对于共享时钟调度器，所有的数据传送都是通过交错的时标和确认消息进行的，不允许其他消息在总线上传送。因此，网络带宽要求可以事先确定，以保证所有的消息能准时传送。

检测网络和节点错误

最后再看一下交通信号灯控制系统。可用于两个节点间同步和数据传输的机制已经讨论过了，尚未讨论的是由网络硬件（电缆、收发器、连接器等等）故障或节点故障引起的问题。

例如，一个可能出现的简单问题是连接两组信号灯的电缆损坏或者完全被切断。这可能导致从节点无法收到主节点发出的时标消息而使调度器停止调度。如果主节点不知道从节点收不到消息，则将存在两组信号灯同时为绿的危险，这可能导致严重的交通事故（参见图 25.11）。

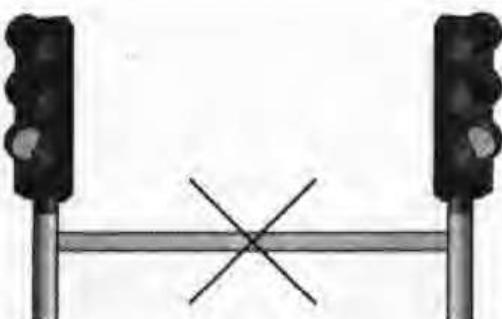


图 25.11 因为电缆故障而处在危险状态的移动式交通信号灯系统

共享时钟调度器用错误检测和恢复机制处理这个潜在的问题，下一节将讨论这一机制。

在从节点侧检测错误

共享时钟调度器让从节点能快速而简单地检测出错误。具体地说，因为设计规范规定从节点将每间隔 1ms（比如说）接收到一个时标消息，所以只需要简单地测量时标消息间的时间间隔即可。如果时标消息间的时间间隔大于 1ms，则可以判断为发生了错误。

在很多情况下，设置一个溢出时间比时标间隔稍长的看门狗计时器^②可以有效地实现这个错误检测方法。正常情况下，每个收到的时标消息都会调用从节点的更新函数，该更新函数将更新看门狗计时器。如果时标消息没有收到，看门狗计时器会溢出，而错误处理例程将被调用。

后面会进一步讨论错误处理例程。

在主节点侧检测错误

在主节点侧检测错误需要每个从节点每隔一定间隔发送一条确认消息至主节点（参见图 25.10），下面以一个一主十从网络说明一个实现主节点侧检测的简单方法。

- 主节点每毫秒同时将时标消息发送至所有的节点，这个消息用于调用所有从节点的更新函数。
- 对于大多数的调度器而言，每个时标消息都含有发送给某个特定节点的数据。这里，假定主节点在时标消息中轮流给每个从节点发送数据，这样每个从节点每隔 10ms 便接收到一次数据。
- 当从节点收到包含其 ID（标识符）的时标消息时，发送确认消息至主节点；对其他时标消息，则不返回确认消息。

正如前面提到过的，这个方案提供了可预测的总线负载和与每个从节点单独通信的机制。另外还意味着主节点能检测到某个从节点是否没有回应时标消息。

处理从节点检测的错误

假定从节点通过看门狗计时器检测错误，书中的共享时钟调度器将以下面的方式处理这种

^② 参见硬件看门狗模式 [第 176 页]。

错误：

- 每当从节点复位时（上电或者因为看门狗溢出而复位），节点便进入“安全状态”。
- 节点保持在这个状态中，直到从主节点接收到一系列正确的启动命令。

这种方式的错误处理易于实现而且在大多数情况下都是有效的。

另一个重要的替代方式是在检测到主节点故障时，将一个从节点转换为主节点。这个方式可能是很有效的，特别是在可以按优先级传输信息的网络上（例如 CAN）。本书对此不做详细讨论。

处理主节点检测的错误

处理从节点检测的错误对于共享时钟网络而言很简单，处理从主节点检测的错误则要复杂得多。看一下本书中的三个主要方法：

- 进入安全状态然后关闭网络。
- 重启动网络。
- 启用后备从节点。

下面依次讨论这三种方法。

进入安全状态并关闭网络

从主节点检测到错误后关闭网络很容易实现——只需要简单地停止发送时标消息。由于时标消息停止发送，从节点将复位，从而进入安全状态等待启动命令。整个网络将停止，直到主节点复位。

如果能够确定什么状态是安全的，这个方法对许多系统的网络错误就都是适用的。这当然是高度依赖于不同的应用的。

例如，前文提到过 A310 空中客车大型客机的前缘缝翼和襟翼操纵计算机，会在降落期间检测到错误时将机翼系统恢复到安全状态并关闭。在这种情况下，安全状态指将两侧的机翼设为一样，在降落期间，只有不平衡的设定是危险的（Burns 和 Wellings, 1997, p.102）。

这个方法的优点和缺点如下：

- ◎ 实现简单。
- ◎ 对许多系统有效。
- ◎ 如果高级恢复方案失败了，这个方法能够提供最后一道防线。
- ◎ 这个方法不试图恢复正常网络操作或者启用后备节点。

这个方法可以用于书中讨论的任何网络（基于中断、基于 UART 或者基于 CAN），第 26 章将详细说明这个方法。

复位网络

另一个处理错误的简单方法是复位主节点和整个网络。当主节点复位后，将试图和每个从节点依次重建通信：如果没能与某个从节点建立通信，则将试着连接该从节点的备用节点。

这个方法有效且易于实现。例如，许多设计使用“N-版本”编程，为关键组件准备备份

版本^③。执行复位后，网络上的所有节点重新同步，然后启用后备从节点（如果有后备节点可用的话）。

这个方法的优点和缺点如下：

- ⑤ 可以充分利用后备节点。
- ⑥ 重启网络费时（可能要半秒或更长）：虽然网络完全恢复正常运作，但延迟可能太长了（例如，汽车制动器或者航空和航天飞行控制）。
- ⑦ 如果设计或者实现有问题，那么错误可能导致网络不停地复位。这可能比简单地“进入安全状态并关闭”方法更不安全。

这个方法可用于书中讨论的任何基于 UART 或者 CAN 的网络。第 27 章将详细说明这个方法。

启用后备从节点

第三种错误恢复方法如下：如果一个从节点失效，与其重启整个网络，不如启动相应的备用单元。

这个方法的优点和缺点如下：

- ⑧ 可以充分利用后备节点。
- ⑨ 在大多数情况下，启用备用单元只需花费相对较短的时间。
- ⑩ 所需的编程比其他两个方案复杂。
- ⑪ 这个方法可用于书中讨论的任何基于 UART 或者 CAN 的网络，第 28 章将详细说明这个方法。

25.5 为什么增加处理器并不一定能改善可靠性

理解下面这个结论非常重要：如果没有认真地设计，那么在一个网络中增加处理器的数目可能对整个系统的可靠性产生有害的影响。

这一结论并不难理解。例如，忽略处理器之间连接可能发生的故障和多处理器体统更复杂的（软件）操作环境可能导致的故障，假设一个网络有 100 个微控制器并且每个微控制器的可靠性是 99.99%。因此，需要所有 100 个节点同时正确工作的多处理器系统的总可靠性是 $99.99\% \times 99.99\% \times 99.99\% \times \dots$ 。最终的结果是 0.9999^{100} ，约为 37%。可靠性大大降低了！一个 99.99% 可靠的器件可以被认为 10000 年才会失效一次，而 37% 可靠的器件则大约每 18 个月就会失效一次^④。

只有当共享时钟设计提供的可靠性增长超过了系统复杂性增加导致的可靠性降低时，才可能保证总系统可靠性的增长。不幸的是，预测任何复杂的设计特性的成本和效益（对可靠性而言）多少都还只是一种巫术（除了对于非常简单的系统）。

例如，考虑前面讨论的使用冗余节点的方法。具体地说，假设我们正在研发汽车的巡航控

③ N-版本编程技术优点和缺点的详细讨论参见 Leveson, 1995。

④ 更精密的分析参见 Storey, 1996。

制系统（如图 25.12 所示）。

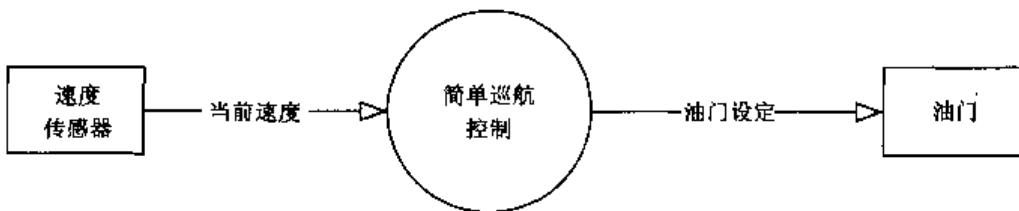


图 25.12 汽车巡航控制系统的概略设计

这个巡航控制系统有明确的安全要求。系统突然出现故障并将汽车设置为最大油门，可能导致致命的结果。因此，希望使用两个微控制器节点，万一第一个节点失效，还有一个备用节点可用。

如果满足以下的要求，这将是一个有效的设计方案：网络上有两个基本上相同的节点，当第一个节点失效时能激活第二个节点。这样似乎可以改善整个系统的可靠性。实际上，这是许多飞机的飞行控制系统所使用的方法，由驾驶员或者副驾驶员根据需要切换“主”、“备”和“滑翔”控制器（例如，Storey, 1996）。

然而，冗余网络的存在并不能保证增加可靠性。例如，1974 年土耳其航空公司的 DC-10 飞机的货舱门突然在高空打开。这个故障导致了货舱减压，接着引起客舱地板塌陷。除主控系统之外，飞机还包含两条（冗余的）控制线，但是这三条线路都位于客舱地板的下面。飞机因此失控，坠毁在巴黎市郊，346 人丧生（Bignell 和 Fortune, 1984, pp. 143~144; Leveson, 1995, p.50 和 p.434）。

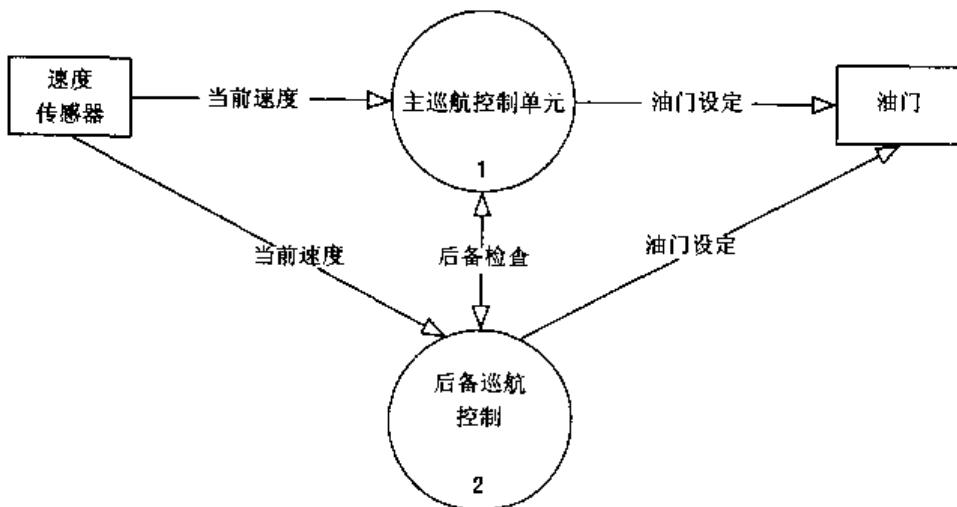


图 25.13 具有后备控制单元的汽车巡航控制系统的概略设计

另外，在许多嵌入式系统中没有操作员或者允许切换到后备节点（或者网络）的时间过短，不可能进行人工干预。在这种情况下，如果检测主节点故障并切换至后备节点的组件比较复杂（通常也是如此），则这个切换组件本身可能成为严重的可靠性问题的根源（参见 Leveson, 1995）。

注意，这些解释并不意味着多处理器设计不适用于高可靠性系统。多处理器可以安全地用于这些环境。然而，所有的多处理器设计都必须小心进行，必须特别严格地进行设计、评审和测试。

25.6 小结

本章讨论了使用多处理器可能带来的一些利弊，同时介绍了共享时钟调度器，并举例说明了共享时钟调度器可被用于建立高效的包含两个或多个微控制器的时间触发系统。

在随后的几章中，将对共享时钟调度器进行详细的说明。

使用外部中断的共享时钟调度器

引言

本章将介绍两个简单的共享时钟调度器的实现。在这两种实现方式中，主从节点都通过脉冲来同步，即，由主节点产生脉冲来触发从节点的中断，从而驱动从节点调度器中的“刷新”函数。

这种简单的调度器不仅功能强大，而且只占用很少的存储器容量、CPU 和硬件开销也很小。

注意：在使用外部中断传递时钟标记时，系统特别容易受到电磁干扰的影响。因此，这种调度器只适用于“本地”网络，即，微控制器间的距离不超过几厘米的情况。大多数情况下，所有微控制器都应安装在同一个屏蔽机箱的同一块印刷电路板上。

共享时钟中断调度器（时标）

适用场合

- 用多个 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何在多个通过中断连接的 8051 构成的局域网络中调度任务？

背景知识

总结第 25 章讲述的内容，共享时钟调度器的主要特性如下：

- 任务调度过程是合作式的。
- 网络由 1 个主节点和 N 个从节点构成， $N \geq 1$ 。

- 主节点外接 1 个精确的振荡器，用来驱动微控制器的内部时钟。时钟到时后运行主节点调度器，进而产生时标消息发送至从节点。
- 当从节点通过中断接收到主节点发送来的时标消息时，运行从节点调度器，即，从节点不使用内部定时器来驱动调度器。
- 每个从节点运行 1 个低精度的看门狗定时器，用于检测主节点是否发生了故障。

此外，大多数共享时钟调度器还有以下特性：

- 每发送 N 个消息后，主节点从每个从节点收到应答，即，各从节点每收到 N 个消息时发送 1 个应答。
- 从节点检测到主节点丢失的最大延迟可以小到 2 个时标周期，具体取决于从节点内部看门狗定时器的设置。
- 主节点检测到所有从节点丢失（例如，当整个网络发生故障时）的最大延迟一般为 1 个时标周期。
- 主节点检测到某个从节点丢失（例如，当某个节点发生故障时）的最大延迟一般为 N 个时标周期。

解决方案

在这一节中将讨论简单共享时钟调度器的 2 个可行实现。

基本技术

正如在第 25 章中讨论的，在多个微控制器之间进行任务调度时，面临的首要问题是确保不同电路板上的任务保持同步。共享时钟调度器通过在不同处理器板之间共享单一时钟解决了该问题，其原理如图 26.1 所示。

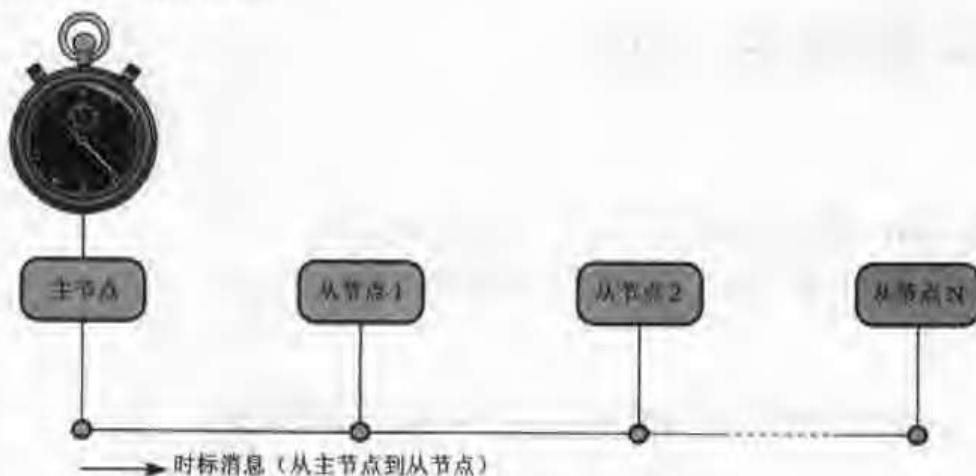


图 26.1 共享时钟调度器的最简单形式

注意，这包括主节点向从节点定时发送时标消息。所有共享时钟调度器都提供此功能。

在网络的主节点上有一个精确时钟，以通常的方式驱动主节点上的调度器，即，通过 1

一个片内定时器溢出产生的中断来触发。这样，一个典型的主节点“刷新”函数看起来如下：

```
void SCI_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    ...
}
```

从节点同样有调度器，然而用来驱动该调度器的中断不是由定时器产生，而是由主节点发送的“时标消息”产生。主节点消息产生中断可以有多种方式，在第六篇中将看到这些方式。对于本章中的所有调度器，中断源是外部中断输入引脚上的电平变化，例如 Pin3.2（中断 0）。

```
void SCI_SLAVE_Update(void) interrupt INTERRUPT_EXTERNAL_0
{
    ...
}
```

在指定时刻产生这样的电平变化很简单：

- 在主节点的“刷新”函数中，将某个端口引脚（如 Pin2.0）由逻辑 1 变为逻辑 0。
- 在从节点中，将调度器的“刷新”函数配置为由某个外部中断触发，如 Pin3.2。
- 将主节点和从节点相连，如图 26.2 所示。

现在，这两个微控制器就可以精确地同步运行了。

注意，这种方法完全可以用在多个从节点的情况下（如图 26.3 所示）。

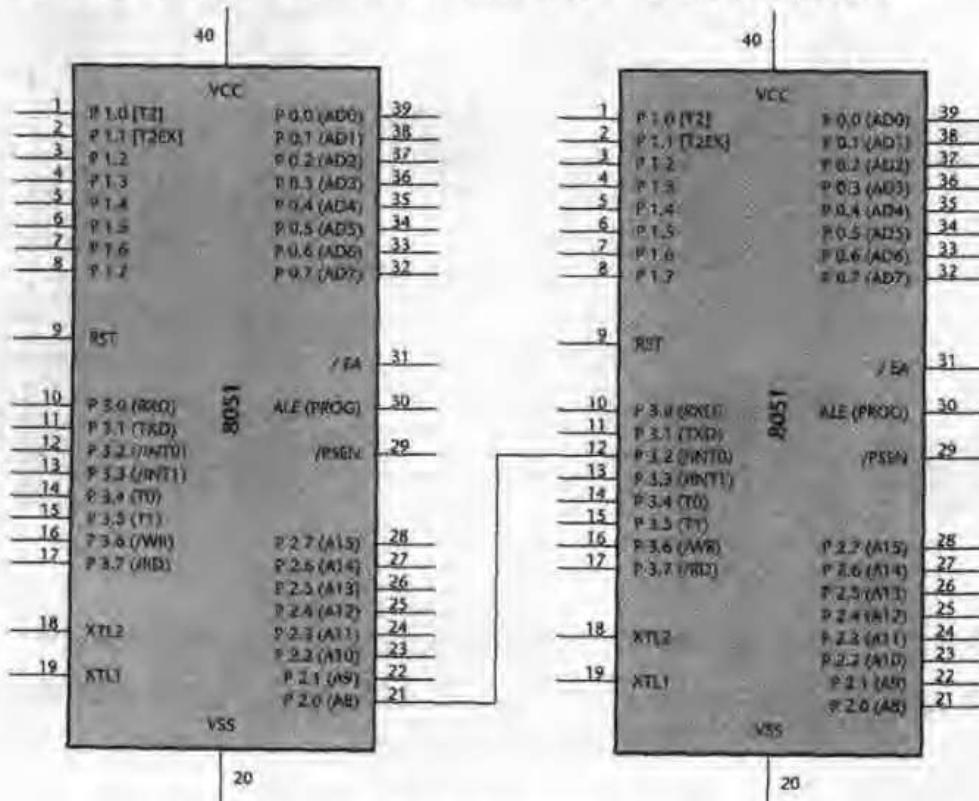


图 26.2 一个非常简单的共享时钟（中断）调度器

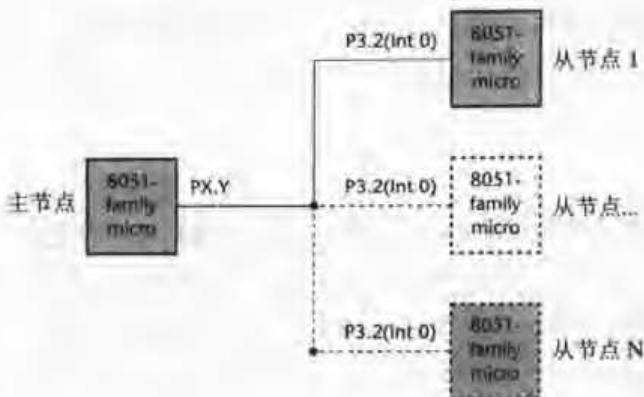


图 26.3 简单的共享时钟（中断）调度器可以包含多个从节点

实现细节

现在考虑这种调度器的实现细节，图 26.4 给出了一个实用的硬件框架。

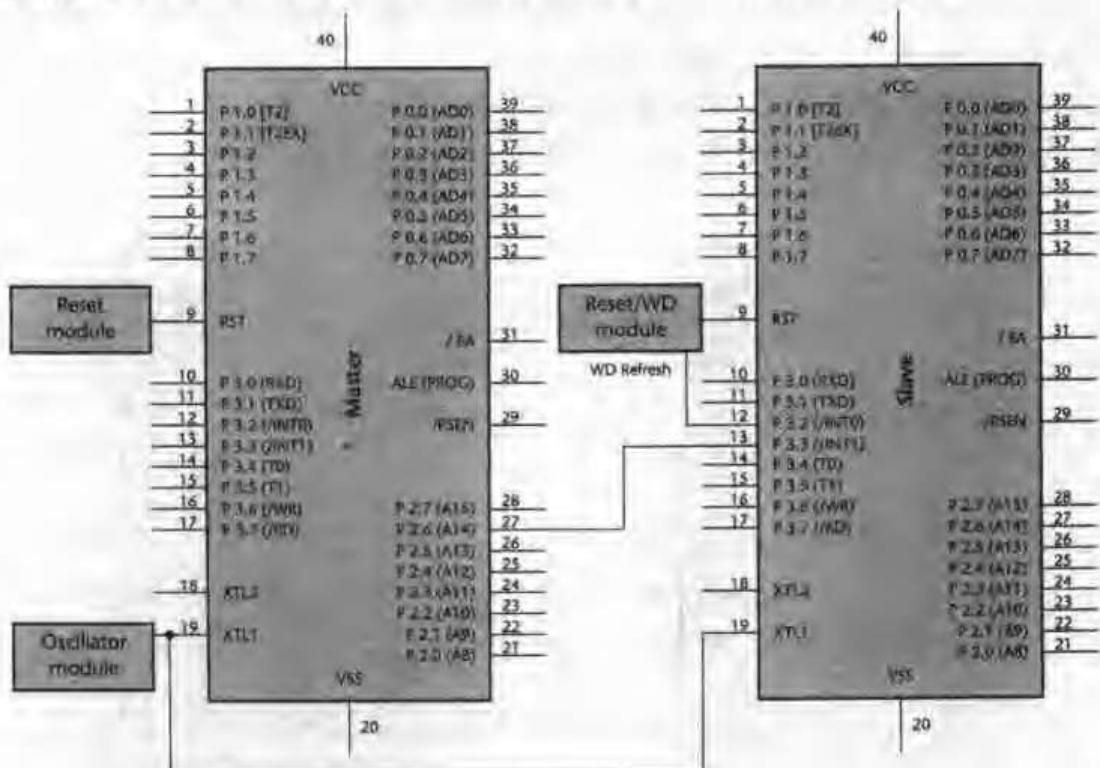


图 26.4 共享时钟中断调度器的一种可行硬件平台（从节点有外部看门狗定时器）

正如在第 25 章中讨论的，可以通过在从节点上增加一个看门狗定时器来增强共享时钟网络的可靠性。一旦有了看门狗定时器，再加上正确的编程，从节点就能够检测到主节点或网络其他部分发生的故障。

当然，并不一定采用外部看门狗定时器。图 26.5 给出了一个采用内部看门狗定时器的替

代硬件平台。

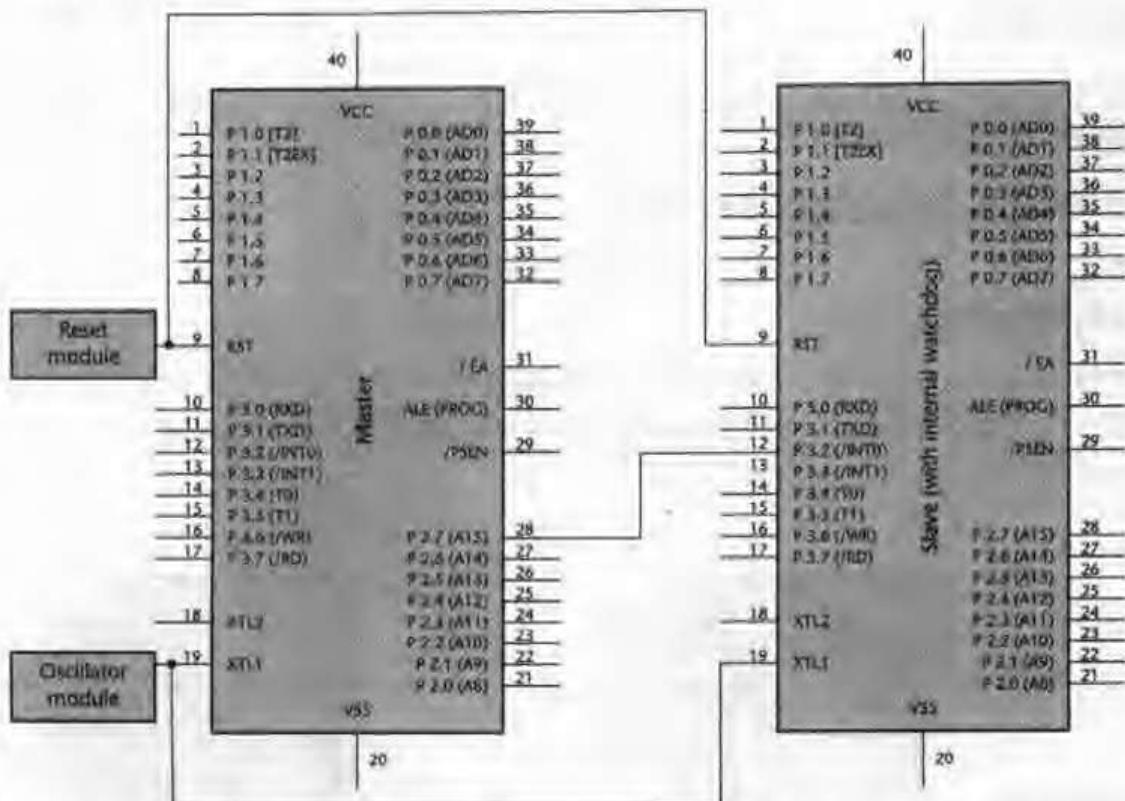


图 26.5 共享时钟中断调度器的一种可行硬件平台（从节点有内部看门狗定时器）

这种调度器的完整实现代码将在后面的例子中给出。

提高系统的可靠性

上述的共享时钟（中断）调度器既简单而又功能强大。在这种系统中，主节点的故障可以被从节点迅速检测到。然而，如果从节点中的某一个发生故障，则无法被主节点和其他从节点检测到。

因此，在要求必须确保所有节点都运行正常的应用场合中，这种调度器就无法胜任了。图 26.6 给出了一个解决该问题的方法。注意，这种方案只适用于两个节点的网络。

如图 26.6 所示，仍然是由两个基于 8051 的节点构成网络。从节点可以通过使用看门狗来检测主节点（或通信连接）的故障，而且从节点用来刷新外部看门狗的信号也被反馈到主节点。通过监视看门狗引脚上的信号变化，主节点可以了解从节点是否仍在运行。

这种调度器的完整代码例子将在本章后面部分中给出。

硬件资源

这是一种高效的调度器。与标准的合作式调度器相比，主节点上所需的额外软件开销通常可以忽略不计。硬件上只要求在每个主节点、从节点上提供 2 个端口引脚。

此外，从节点中的定时器0、定时器1和定时器2（如果有的话）都未被占用。

可靠性和安全性

使用共享时钟中断调度器（时标）存在一些重要的安全性和可靠性问题。在决定采用这种结构之前请仔细阅读本节内容。

为什么增加处理器后并不一定能提高可靠性

第434页中的讨论给出了使用多处理器可能会降低系统可靠性的原因。

振荡器频率漂移的影响

整个网络（例如10个微控制器）的时间精度取决于主节点时钟的稳定性。如果必要，可以在主节点中采用温度补偿时钟或基于卫星时钟来保证整个网络的时间精度。

相关技术将在“稳定的调度器”中讨论。

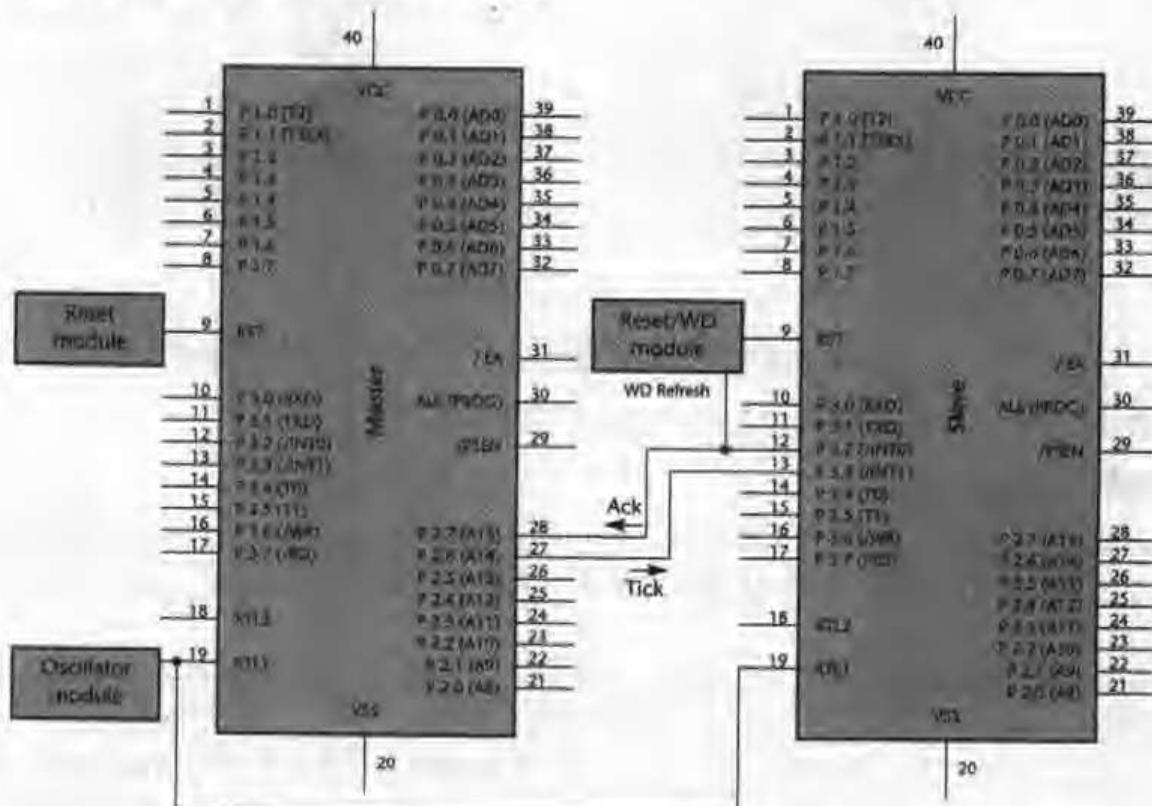


图26.6 共享时钟中断调度器的另一种形式

注意，这种方式下，主节点能够了解从节点是否在运行。

振荡器故障的影响

共享时钟调度器的一个潜在问题是主节点的“单点失效”，即，如果主节点上的振荡器发生故障或者主节点发生软、硬件故障，整个网络将停止工作。

这种担心是合理的，但是可以通过建立一个或多个“备用主节点”来解决。运行方式如下：

- 正常情况下，主节点的时标将备用主节点置为睡眠。
- 一旦时标停止（说明主节点发生故障），备用主节点将主节点从总线上断开，并取而代之。
- 通过改变对主节点故障的响应时间，同一个网络上可以使用多个备用主节点。这样，某个备用主节点将在 1 秒后切入，第二个备用主节点在 2 秒后切入，依此类推。

使用外部中断的危险之处

采用外部中断来传递时标信息意味着系统特别容易受到电磁干扰的影响，除非采取了适当的预防措施。

如果中断引脚上的引线过长（相当于天线），将使微控制器对电磁干扰更为敏感。因此，这种调度器只适用于“本地”网络，即，多个微控制器的间距不超过几厘米的应用场合。大多数情况下，所有微控制器都应安装在屏蔽机箱的同一块印刷电路板上。

检测和处理故障

在第 25 章中讨论了 3 种检测和处理共享时钟网络故障的技术。在本章中，假定采用“进入安全状态然后关闭”的故障处理技术，详见第 25 章。

可移植性

几乎所有的微控制器都有至少 1 个外部中断引脚。因此，采用这种方式毫无问题。

优缺点小结

- ⊕ 简单，有效，而且开销极小。
- ⊖ 主从节点之间提供了单向通信机制，主节点无法检测到从节点上的故障。
- ⊖ 缺乏主从节点之间的数据传输机制。
- ⊖ 除非采取适当的预防措施，否则系统极易受电磁干扰的影响。

相关的模式和替代解决方案

本章和第 6 篇中的其他模式提供了其他接多个微控制器的可选技术。

此外，图 26.7 给出了另一种选择。如图 26.7 所示，主从节点之间的惟一连接是复位和时钟电路。每个器件都有自己的调度器，并由定时器驱动。然而由于以下原因，一般不推荐使用这种方法：

- 每个节点上都必须占用一个重要的硬件资源（定时器）。
- 两个（或多个）节点上的任务通常无法同步，这是因为每个节点开始调度的时刻取决于初始化该节点（如果有的话）所需的时间。
- 主从节点完全独立，任一节点都不知道其他节点的状态。

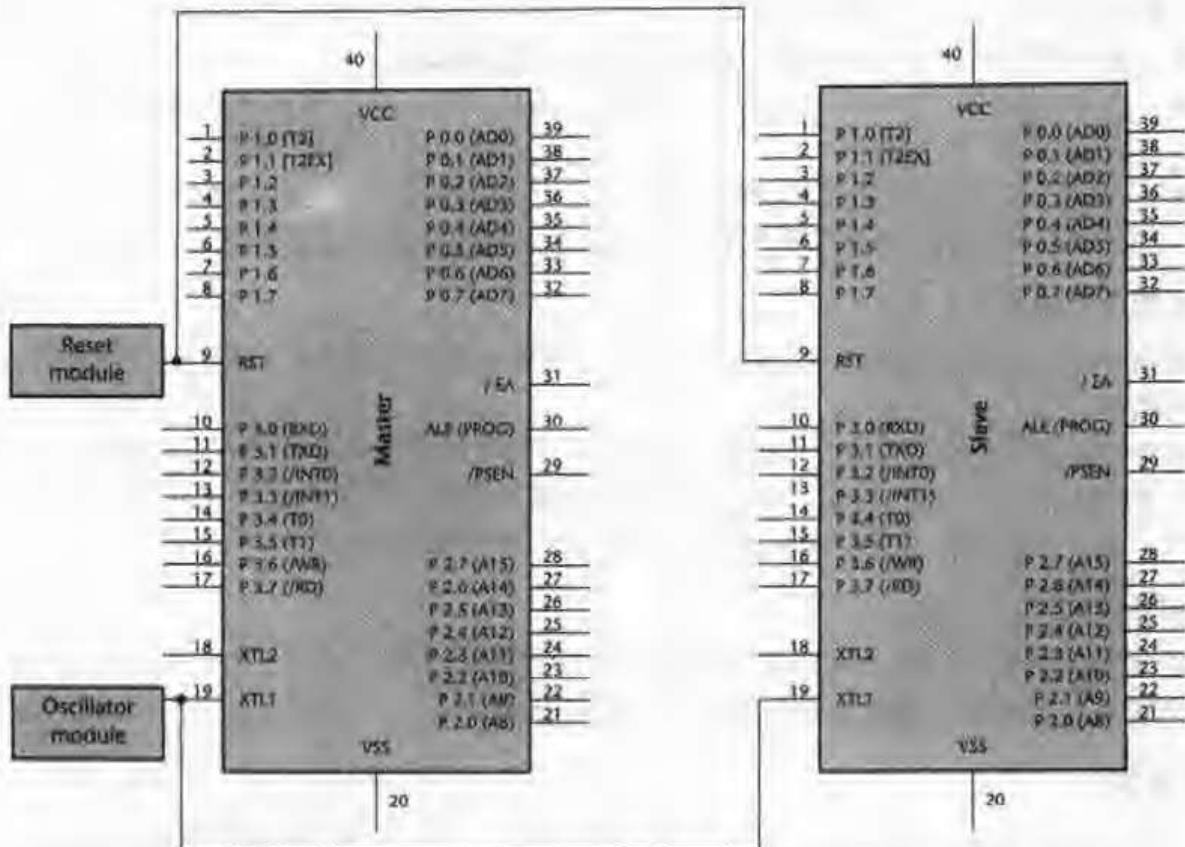


图 26.7 一种可选的共享时钟调度器。主从节点之间为松散耦合

举例：精确的定时器时标和标准波特率

如图 26.6 所示，共享时钟中断调度器通常采用 1 个振荡电路来驱动两个（或多个）处理器，这样可以降低成本和电路板的尺寸。

然而，这并不是最佳选择。例如，如果采用 12（或 24）MHz 的振荡器来驱动主节点，则所有微控制器都可以工作在精确的 1ms 时标下。假如某个从节点有 1 个独立的 11.095MHz 的振荡电路，它将可以工作在 1ms 时标下，同时能够产生标准（如 9600）波特率（如图 26.8 所示）。

这种组合方式在某些应用场合中非常有用。

举例：交通信号灯（版本 1）

本章将通过在第 25 章介绍的“交通信号灯”例子的不同版本来阐述调度器的使用。

图 26.9 给出了该系统的第一个例子所采用的硬件。引脚 2.0、2.1 和 2.2 上的 LED 灯表示交通信号灯（分别对应红、黄、绿）。引脚 0.7 上的 LED 灯是一个“闪烁 LED”，用来显示样机的运行状态。

源程序清单 26.1~源程序清单 26.7 给出了基于共享时钟中断调度器（时标）的主从节点

软件。

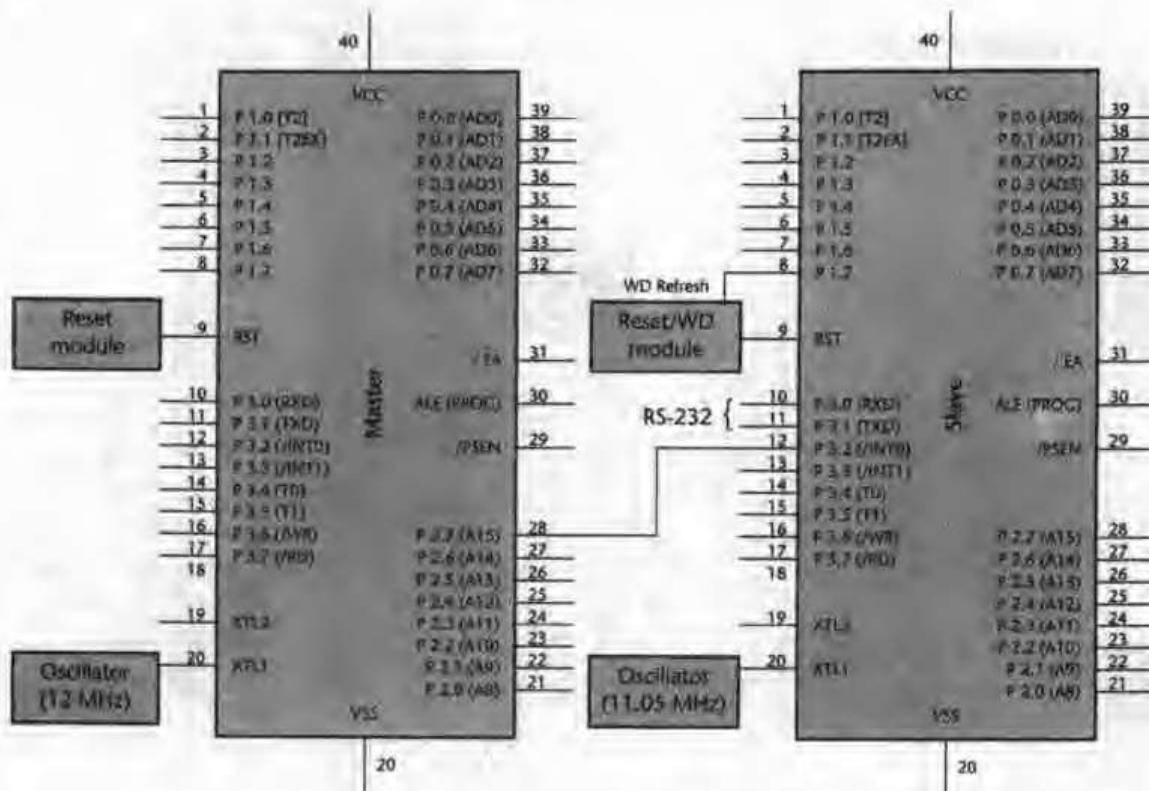


图 26.8 主从节点采用不同振荡频率的共享时钟调度器

软件——主节点

```
/*
Port.H (v1.00)
-----
'Port Header' (see Chap 10) for the project Sci_Tilm (Chap 26)
*/
// ----- Sch51.C -----
// 如果不需要错误报告，注释掉此行
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P1
#endif
// ----- SCI_Tilm.C -----
// 这个引脚连接至从节点微控制器的中断输入引脚（通常是 P3.2 或 P3.3）
sbit Interrupt_output_pin = P2^5;
// ----- TLight_A.C -----
sbit Red_light = (P2^0);
```

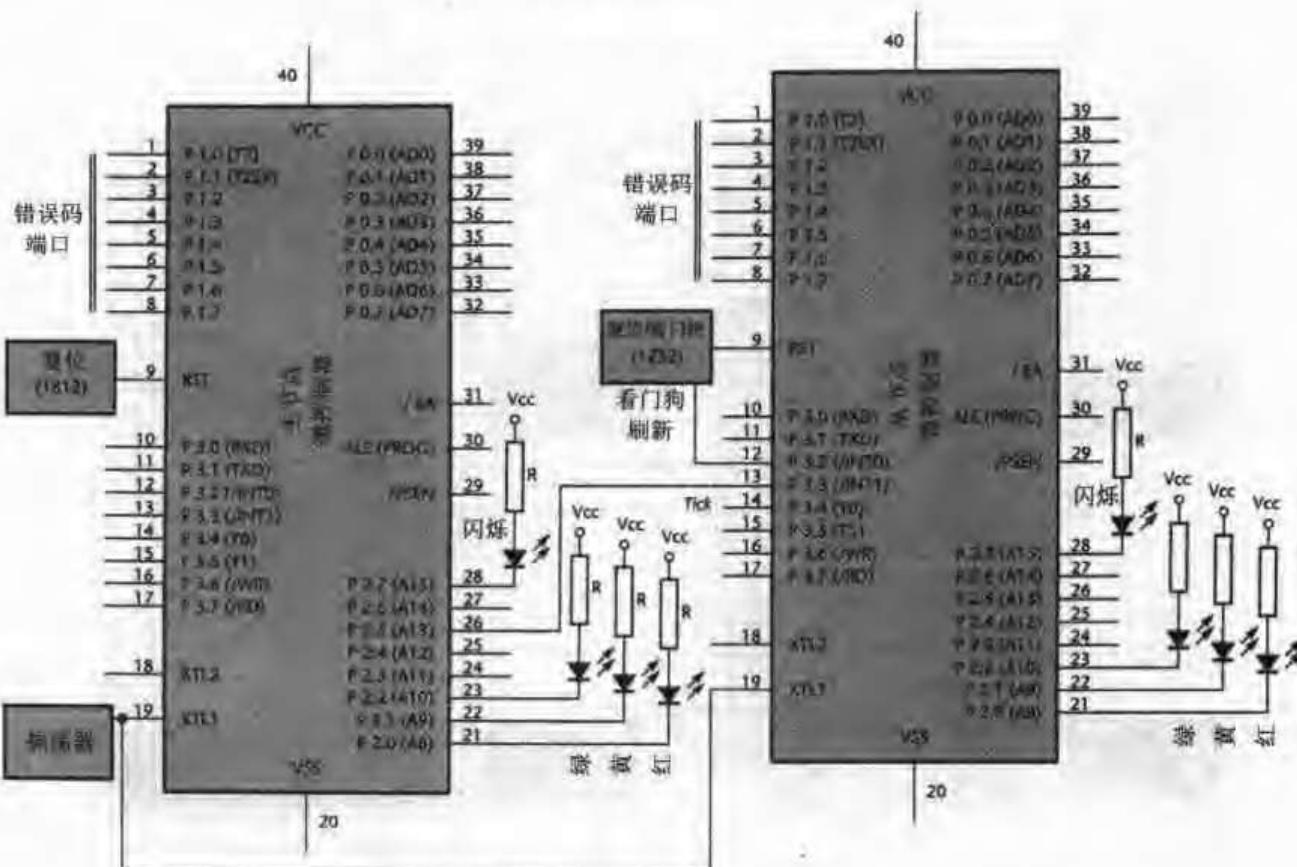


图 26.9 例子“简单交通信号灯”所用的硬件

注意，在这个简单的硬件版本中，主节点不能检测到从节点的故障。另外，主节点和从节点之间无法相互传输数据。这个例子只是用于说明共享时钟调度器的体系结构。

```
sbit Amber_light = (P2^1);
sbit Green_light = (P2^2);
// ----- LED_Flas.C -----
// 用于让 LED 闪烁
sbit LED_pin = P2^7;
/*
----- 文件结束 -----
*/

```

源程序清单 26.1 简单交通信号灯系统的部分软件（主节点）

[注意：在这个简单的硬件版本中，主节点不能检测到从节点的故障。另外，主节点和从节点之间无法相互传输数据。]

```
/*
Main.c (v1.00)

// 89C52 共享时钟调度器的测试程序。
***主节点和从节点的时标间隔都是 1ms ***
所需的链接程序选项（详情参见正文）：
OVERLAY (main - (LED_Flash_Update,TRAFFIC_LIGHTS_Update),

```

```

SCH_Dispatch_Tasks ! (LED_Flash_Update,TRAFFIC_LIGHTS_Update))
-----*/
#include "Main.h"
#include "LED_flas.h"
#include "SCI_Ti1m.H"
#include "TLight_A.h"
/* ..... */
/* ..... */
void main(void)
{
    // 设置调度器
    SCI_TICK1_MASTER_Init_T2();
    // 准备交通信号灯任务
    TRAFFIC_LIGHTS_Init();
    // 准备 LED 闪烁任务 (仅用于演示)
    LED_Flash_Init();
    // 增加一个 LED 闪烁任务 (亮 1000ms, 灭 1000ms)
    SCH_Add_Task(LED_Flash_Update, 0, 1000);
    // 增加一个“交通信号灯”任务
    SCH_Add_Task(TRAFFIC_LIGHTS_Update, 30, 1000);
    // 启动调度器
    SCI_TICK1_MASTER_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
-----文件结束-----
-----*/

```

源程序清单 26.2 简单交通信号灯系统的部分软件（主节点）

注意，在这个简单的硬件设计版本中，主节点不能检测到从节点的故障。另外，主节点和从节点之间无法相互传输数据。

```

-----*
  SCI_Ti1m.c (v1.00)
-----
这是 8051/52 的共享时钟中断调度器
***主节点：唯一的时标信号（双工） ***
***T2 用于定时，16 位自动重新装入 ***
***12 MHz 振荡器->精确的 1ms 时标间隔 ***
---假设从节点上有 1232 看门狗---
-----
#include "Main.h"
#include "Port.h"
#include "SCI_Ti1m.H"
#include "Delay_T0.h"
#include "TLight_A.h"
// ----- 公有变量声明 -----
// 任务数组 (参见 Sch51.c)
-----*
```

```

extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量(参见 Sch51.c)
extern tByte Error_code_G;
/*-----*
SCI_TICK1_MASTER_Init_T2()
调度器初始化函数, 准备调度器数据结构并将定时器中断设置为指定时标间隔,
在使用调度器前, 必须调用此函数。
*-----*/
void SCI_TICK1_MASTER_Init_T2(void)
{
    tByte i;
    // 尚无中断
    EA = 0;
    // -----设置调度器-----
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 清除全局错误码变量
    // SCH_Delete_Task()将产生一错误代码
    // (因为任务数组是空的)
    Error_code_G = 0;
    // -----设置定时器2(开始)-----
    // 现在设置定时器2
    // 16位定时器函数, 自动重新装入
    // 晶振频率为12MHz
    // 定时器2的分辨率是0.000001秒(1微秒)
    // 所需的定时器2溢出时间是0.001秒(1ms)
    // 需要1000个时标
    // 重装的值为65536 - 1000 = 64536(十进制) = 0xFC18

    T2CON     = 0x04;    // 装入定时器2控制寄存器
    T2MOD     = 0x00;    // 装入定时器2方式寄存器
    TH2       = 0xFC;    // 装入定时器2高字节
    RCAP2H    = 0xFC;    // 装入定时器重装捕捉寄存器, 高字节
    TL2       = 0x18;    // 装入定时器2低字节
    RCAP2L    = 0x18;    // 装入定时器重装捕捉寄存器, 低字节
    ET2       = 1;       // 使能定时器2中断
    TR2       = 1;       // 启动定时器2
    // -----设置定时器2(结束)-----
}

/*-----*
SCI_TICK1_MASTER_Start()
使能中断, 启动调度器。
注意: 通常在所有的常规任务加入后调用, 以保持任务同步。
注意: 只能使能调度器中断!!!
*-----*/
void SCI_TICK1_MASTER_Start(void)

```

10

```

{
// 在启动或发生错误后，将系统置于安全状态。
SCI_TICK1_MASTER_Enter_Safe_State();
// 在这儿等待，以便从节点超时复位
// 调节等待时间，和从节点的超时时间匹配
Hardware_Delay_T0(500);
// 现在发送第一个定时时标（启动从节点）
Interrupt_output_pin = 1;
Hardware_Delay_T0(5);
Interrupt_output_pin = 0;
Hardware_Delay_T0(5);
Interrupt_output_pin = 1; // 在下降沿启动
// 启动调度器
EA = 1;
}
/*-----*/
SCI_TICK1_MASTER_Update_T2
这是调度器中断服务程序。该程序的调用频率由 SCI_TICK1_MASTER_Init_T2()
函数中的定时器设定决定。本版本中由定时器 2 中断触发：定时器会自动重装
*-----*/
void SCI_TICK1_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
tByte Index;
TF2 = 0; // 必须人工清除。
// 现在发送定时时标至从节点
Interrupt_output_pin = 0;
// 注意：计算单位是定时时标数，而非毫秒
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
// 检查是否有任务
if (SCH_tasks_G[Index].pTask)
{
if (SCH_tasks_G[Index].Delay == 0)
{
// 任务将开始运行
SCH_tasks_G[Index].RunMe += 1; // 运行标志加 1
if (SCH_tasks_G[Index].Period)
{
// 再次调度周期性的任务运行
SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
}
}
else
{
// 还没准备好运行：延迟减 1
SCH_tasks_G[Index].Delay -= 1;
}
}
}
}

```

```

// 准备下次的定时时标到来
Interrupt_output_pin = 1;
}

/*
-----*
SCI_TICK1_MASTER_Enter_Safe_State()

当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位
(2) 从节点失效
(3) 网络出错
(4) 因为任何其他的原因，确认消息没有按时收到
在这些情况下，把系统置于安全状态
-----*/
void SCI_TICK1_MASTER_Enter_Safe_State(void) reentrant
{
    // 由用户根据自己的需要编辑
    // 显示安全输出
    TRAFFIC_LIGHTS_Display_Safe_Output();
}
/*
-----文件结束-----
-----*/

```

程序清单 26.3 简单交通信号灯系统的部分软件（主节点）

注意，在这个简单的硬件设计版本中，主节点不能检测到从节点的故障。另外，主节点和从节点间无法相互传输数据。

```

/*
-----*
TLight_A.C (v1.00)

-----*
交通信号灯控制程序
私有常量
单向通信版本，没有其他的节点灯泡状态的信息。
-----*/
#include "Main.h"
#include "Port.h"
#include "TLight_A.h"
// ----- Private constants -----
// 很容易在这里改变逻辑
#define ON 0
#define OFF 1
// 四种可能的灯光状态的显示时间
// (时间的单位是秒，必须每秒调用更新任务一次)
//
#define RED_DURATION (10)
#define RED_AND_AMBER_DURATION (10)
// 注意：
// 必须指定这是个主节点还是个从节点
// GREEN_DURATION 必须等于 RED_DURATION

```

```
#define GREEN_DURATION RED_DURATION
#define AMBER_DURATION RED_AND_AMBER_DURATION
// 必须指定这是一个主节点单元还是从节点单元
#define MASTER_SLAVE MASTER
// -----私有变量-----
// 系统状态
static eLight_State Light_state_G;
// -----私有函数原型 -----
bit TRAFFIC_LIGHTS_Check_Local_Bulb(void);
/*-----*/
TRAFFIC_LIGHTS_Init()
准备调度
-----*/
void TRAFFIC_LIGHTS_Init(void)
{
    // 主节点和从节点必须在相反的状态下启动
    if (MASTER_SLAVE == MASTER)
    {
        Light_state_G = RED;
    }
    else
    {
        Light_state_G = GREEN;
    }
    // 显示安全的信号，直到调度器启动
    TRAFFIC_LIGHTS_Display_Safe_Output();
}
/*-----*/
TRAFFIC_LIGHTS_Update()
必须每秒被调度一次
-----*/
void TRAFFIC_LIGHTS_Update(void)
{
    static tWord Time_in_state;
    // 检查该节点的灯泡是否烧掉了
    TRAFFIC_LIGHTS_Check_Local_Bulb();
    // 这是主更新代码
    switch (Light_state_G)
    {
        case RED:
        {
            Red_light = ON;
            Amber_light = OFF;
            Green_light = OFF;
            if (++Time_in_state == RED_DURATION)
            {
                Light_state_G = RED_AMBER;
                Time_in_state = 0;
            }
        }
    }
}
```

```
break;
}
case RED_AMBER:
{
    Red_light = ON;
    Amber_light = ON;
    Green_light = OFF;
    if (++Time_in_state == RED_AND_AMBER_DURATION)
    {
        Light_state_G = GREEN;
        Time_in_state = 0;
    }
    break;
}
case GREEN:
{
    Red_light = OFF;
    Amber_light = OFF;
    Green_light = ON;
    if (++Time_in_state == GREEN_DURATION)
    {
        Light_state_G = AMBER;
        Time_in_state = 0;
    }
    break;
}
case AMBER:
{
    Red_light = OFF;
    Amber_light = ON;
    Green_light = OFF;
    if (++Time_in_state == AMBER_DURATION)
    {
        Light_state_G = RED;
        Time_in_state = 0;
    }
    break;
}
case BULB_BLOWN:
{
    // 检测烧掉的灯泡
    // 关掉所有的灯泡
    // (司机们不会高兴, 但会清楚有地方出错了)
    Red_light = OFF;
    Amber_light = OFF;
    Green_light = OFF;
    // 保持这个状态, 直到人工地改变了状态或者系统被复位
    break;
}
```

```

        }
    }

/*-----*
 * TRAFFIC_LIGHTS_Check_Local_Bulb()
 * 检查本节点的灯泡状态（此处的源码为空函数）
 *-----*/
bit TRAFFIC_LIGHTS_Check_Local_Bulb(void)
{
    // 该空函数用于确认灯泡是好的
    //
    // 该函数的完整版本见第 32 章
    return RETURN_NORMAL;
}

/*-----*
 * TRAFFIC_LIGHTS_Display_Safe_Output()
 * 用于系统万一故障时
 *-----*/
void TRAFFIC_LIGHTS_Display_Safe_Output(void)
{
    if (TRAFFIC_LIGHTS_Check_Local_Bulb() == RETURN_NORMAL)
    {
        // 灯泡正常
        // 显示停止
        Red_light = ON;
        Amber_light = OFF;
        Green_light = OFF;
    }
    else
    {
        // 至少一个灯泡烧掉了
        // 关掉所有的灯泡
        Red_light = OFF;
        Amber_light = OFF;
        Green_light = OFF;
    }
}

/*-----*
 * ---文件结束---
 *-----*/

```

源程序清单 26.4 简单交通信号灯系统的部分软件（主节点）

注意，在这个简单的硬件设计版本中，主节点不能检测到从节点的故障。另外，主节点和从节点之间无法相互传输数据。

软件——从节点

```

/*-----*
 * Port.H (v1.00)
 *-----*
 * SCI_Tick 项目（参见第 26 章）的端口头文件（参见第 10 章）
 *-----*/

```

```

// ----- Sch51.C -----
// 如果不需要报告错误，注释掉此行
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P1
#endif
// ----- SCI_Tils.C -----
// P3.3 用于中断输入
// 1232 的/ST 引脚连接到 WATCHDOG_pin
sbit WATCHDOG_pin = P3^2;
// ----- TLight_A.C -----
sbit Red_light = (P2^0);
sbit Amber_light = (P2^1);
sbit Green_light = (P2^2);
// ----- LED_Flas.C -----
// LED 闪烁
sbit LED_pin = P2^7;
/*
----- 文件结束 -----
*/

```

源程序清单 26.5 简单交通信号灯系统的部分软件（从节点）

注意，在这个简单的硬件版本中，主节点不能检测到从节点的故障。另外，主节点和从节点间无法相互传输数据。

```

*-----*
Main.c (v1.00)

共享时钟中断调度器的测试程序
***定时时标 2 -从节点代码***
--- 假设硬件为： ---
--- 89C52 (或者任何有定时器 2 的 8051/52 器件)
--- DS1232 (或者类似的) 外部看门狗
*** 主节点和从节点的时标间隔都是 1ms
*** 详情参见主节点的代码 ***
所需的链接程序选项 (详情参见正文):
OVERLAY (main ~ (LED_Flash_Update,TRAFFIC_LIGHTS_Update),
SCH_Dispatch_Tasks ! (LED_Flash_Update,TRAFFIC_LIGHTS_Update))
*-----*/
#include "Main.h"
#include "LED_Flas.h"
#include "SCI_Tils.h"
#include "TLight_A.h"
/* ..... */ /* ..... */
void main(void)
{
    // 设置调度器

```

```

SCI_TICK1_SLAVE_Init();
// 建立 LED 闪烁任务（仅用于演示）
LED_Flash_Init();
// 准备交通信号灯任务
TRAFFIC_LIGHTS_Init();
// 增加一个 LED 闪烁任务（亮 1000ms，灭 1000ms）
SCH_Add_Task(LED_Flash_Update, 0, 1000);
// 增加一个“交通信号灯”任务
SCH_Add_Task(TRAFFIC_LIGHTS_Update, 10, 1000);
// 启动调度器
SCI_TICK1_SLAVE_Start();
while(1)
{
    SCH_Dispatch_Tasks();
}
/*
-----文件结束-----
*/

```

源程序清单 26.6 简单交通信号灯系统的部分软件（从节点）

注意，在这个简单的硬件版本中，主节点不能检测到从节点的故障。另外，主节点和从节点间无法相互传输数据。

```

/*-----*
 * SCI_Ti1s.c (v1.00)
 *
 * 这是基于中断的 8051/52 共享时钟调度器程序。
 * ***定时时标 1 -从节点***
 * ***使用 1232 看门狗计时器***
 * ***主节点和从节点的时标间隔都是 1ms ***
 * ***详情参见主节点的代码***
 *
 *-----*/
#include "Main.h"
#include "Port.h"
#include "SCI_Ti1s.h"
#include "TLight_A.h"
// -----公有变量声明-----
// 任务数组（参见 Sch51.c）
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量数组（参见 Sch51.c）
extern tByte Error_code_G;
// -----私有函数原型-----
static void SCI_TICK1_SLAVE_Enter_Safe_State(void);
static void SCI_TICK1_SLAVE_Watchdog_Init(void);
static void SCI_TICK1_SLAVE_Watchdog_Refresh(void) reentrant;
/*
 *-----*
 * SCI_TICK1_SLAVE_Init()
 * 调度器初始化函数。准备调度器数据结构并设置定时器中断频率。
 * 使用调度器前必须调用此函数。
 *
 *-----*/

```

```

void SCI_TICK1_SLAVE_Init(void)
{
    tByte i;
    // 清除所有的任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // SCH_Delete_Task()将生成一错误代码
    // (因为任务数组是空的)
    Error_code_G = 0;
    // -----外部中断0 -----
    // 从节点是由中断输入驱动的
    // 使能中断
    // 由引脚 P3.2 上的下降沿触发
    ITO = 1;
    EX0 = 1;
    // 启动看门狗
    SCI_TICK1_SLAVE_Watchdog_Init();
}

/*-----*
SCI_TICK1_SLAVE_Start()
使能中断，启动从节点调度器
注意：通常在所有的常规任务加入后调用，以保持任务间的同步。
注意：只能使能调度器中断！！！
-*-----*/
void SCI_TICK1_SLAVE_Start(void)
{
    // 将系统置于安全状态
    // 如果主节点没有启动或者启动失败，从节点将返回到这里
    SCI_TICK1_SLAVE_Enter_Safe_State();
    // 现在处于安全状态
    // 在这里等待第一个定时时标
    // (刷新看门狗以避免重启动)
    while (IE0 == 0)
    {
        SCI_TICK1_SLAVE_Watchdog_Refresh();
    }
    // 清除标志
    IE0 = 0;
    // 启动调度器
    EA = 1;
}
/*-----*
SCI_TICK1_SLAVE_Update
这是调度器中断服务程序，其调用频率由 SCI_TICK1_SLAVE_Init_T2()
函数中的定时器设定决定
从节点由外部中断触发

```

```

This Slave is triggered by external interrupts.
-----*/
void SCI_TICK1_SLAVE_Update(void) interrupt INTERRUPT_EXTERNAL_0
{
    tByte Index;
    // 喂看门狗
    // 主节点将监视此引脚以检查从节点的活动
    SCI_TICK1_SLAVE_Watchdog_Refresh();
    // 现在做标准的调度器更新
    // 注意：计算的单位是定时时标，而非毫秒
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // 检查这个位置上是否有一个任务
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // 任务将开始运行
                SCH_tasks_G[Index].RunMe += 1; // 运行标志加 1
                if (SCH_tasks_G[Index].Period)
                {
                    // 再次调度周期性任务运行
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // 还没准备好运行；延迟减 1
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}
/*
SCI_TICK1_SLAVE_Watchdog_Init()
这个函数设置看门狗计时器
如果主节点失效（或者发生了其他的错误），时标消息没有到达，调度器将停止运行。
为了检测这种情况，从节点上有一个硬件看门狗在运转。
这个看门狗设置为大约 100ms 溢出，将把系统置于安全状态。
然后从节点将一直等待，直到问题得到解决。
注意：在这些情况下，从节点不会发送确认消息。主节点（如果在运行）会因此察觉到出了问题。
*/
void SCI_TICK1_SLAVE_Watchdog_Init(void)
{
    // 1232 外部看门狗不需要初始化
    // 其他硬件看门狗可能需要
    //
    // 按需要编写相应的初始化程序
}

```

```

/*
SCI_TICK1_SLAVE_Watchdog_Refresh()
喂外部(1232)看门狗
超时时间在60到250ms间(取决于硬件)
硬件:假定为外部1232看门狗
*/
void SCI_TICK1_SLAVE_Watchdog_Refresh(void) reentrant
{
    static bit WATCHDOG_state;
    //改变看门狗引脚的状态
    if (WATCHDOG_state == 1)
    {
        WATCHDOG_state = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state = 1;
        WATCHDOG_pin = 1;
    }
}
/*
SCI_TICK1_SLAVE_Enter_Safe_State()
当系统发生下列情况时,进入此状态:
(1) 节点上电或者复位
(2) 主节点失效
(3) 网络出错
(4) 因为任何其他的原因,时标消息没有按时收到
在这些情况下,把系统置于安全状态
*/
void SCI_TICK1_SLAVE_Enter_Safe_State(void)
{
    //由用户根据自己的需要编辑
    TRAFFIC_LIGHTS_Display_Safe_Output();
}
/*
---文件结束---
*/

```

源程序清单 26.7 简单交通信号灯系统的部分软件(从节点)

注意,在这个简单的硬件版本中,主节点不能检测到从节点的故障。另外,主节点和从节点间无法相互传输数据。

例子: 交通信号灯(版本2)

上述例子中的交通信号灯设计的主要缺点是主节点不能检测出从节点出了故障。本例将通过在通信协议中同时包含时标消息和确认消息来解决这个问题,所需硬件如图26.10所示。

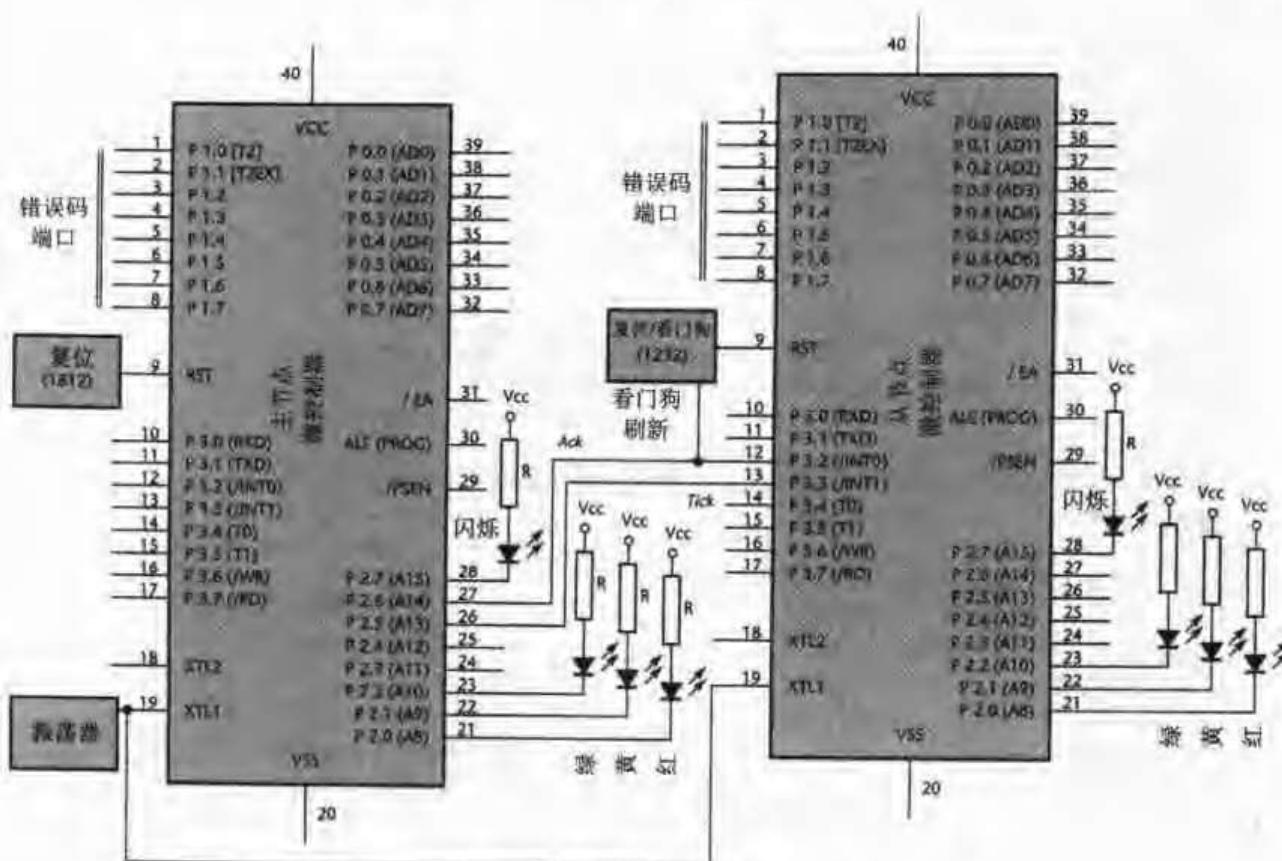


图 26.10 简单交通信号灯例子（版本 2）的硬件

注意，在这个版本的设计中，主节点和从节点间无法相互传输数据。

软件——主节点

这里列出了主节点调度器的核心程序：其余部分的程序包含在本书所附的 CD 上。

```
/*
 * SCI_Ti2m.c (v1.00)
 *
 * 这是 8051/52 的共享时钟中断调度器
 * *** 主节点：惟一的时标信号（双工） ***
 * *** T2 用于定时，16 位自动重新装入 ***
 * *** 12 MHz 振荡器 -> 精确的 1ms 时标间隔 ***
 * --- 设从节点上有 1232 看门狗 ---
 */
#include "Main.h"
#include "Port.h"
#include "SCI_Ti2m.H"
#include "Delay_T0.h"
#include "TLight_A.h"
// ----- 公有变量定义 -----
// 用于检测从节点的活动
```

```

bit First_call_G;
bit Watchdog_input_previous_G;
// -----公有变量声明-----
// 任务数组 (参见 Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量数组 (参见 Sch51.c)
extern tByte Error_code_G;
// 用于从节点发生错误时复位系统 (参见 Main.C)
extern bit System_reset_G;
// -----私有函数原型-----
static void SCI_TICK2_MASTER_Send_Tick_Message(void);
static bit SCI_TICK2_MASTER_Process_Ack(void);
/*-----*/
SCI_TICK2_MASTER_Init_T2()
调度器初始化函数。准备调度器数据结构并将定时器中断设置为指定时标间隔,
在使用调度器前, 必须调用此函数。
-*-----*/
void SCI_TICK2_MASTER_Init_T2(void)
{
    tByte i;
    // 尚无中断
    EA = 0;
    // -----设置调度器-----
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 清除全局错误变量
    // SCH_Delete_Task()将生成一错误代码。
    // (因为任务数组是空的)
    Error_code_G = 0;
    // ----- 设置定时器 2 (开始) -----
    // 现在设置定时器 2
    // 16 位定时器函数, 自动重新装入
    // 晶振频率为 12MHz
    // 定时器 2 的分辨率是 0.000001 秒 (1 微秒)
    // 所需的定时器 2 溢出时间是 0.001 秒 (1ms)
    // 需要 1000 个时标
    // 重新装入的值为 65536 - 1000= 64536 (十进制) = 0xFC18
    T2CON = 0x04; // 装入定时器 2 控制寄存器
    T2MOD = 0x00; // 装入定时器 2 方式寄存器
    TH2 = 0xFC; // 装入定时器 2 高字节
    RCAP2H = 0xFC; // 装入定时器重装捕捉寄存器, 高字节
    TL2 = 0x18; // 装入定时器 2 低字节
    RCAP2L = 0x18; // 装入定时器重装捕捉寄存器, 低字节
    ET2 = 1; // 使能定时器 2 中断
    TR2 = 1; // 启动定时器 2
    // ----- 设置定时器 2 (结束) -----
}

```

```
/*
SCI_TICK2_MASTER_Start()
使能中断，启动调度器
注意：通常在所有的常规任务加入后调用，以保持任务同步。
注意：只能使能调度器中断！！！
*/
void SCI_TICK2_MASTER_Start(void)
{
    // 在启动或发生错误后，将系统置于安全状态。
    SCI_TICK2_MASTER_Enter_Safe_State();
    // 在这里等待，以便从节点超时复位
    // 调节等待时间，和从节点的超时时间匹配
    Hardware_Delay_T0(500);
    // 现在发送第一个定时时标，启动从节点
    // （在下降沿启动）
    Interrupt_output_pin = 1;
    Hardware_Delay_T0(5);
    Interrupt_output_pin = 0;
    Hardware_Delay_T0(5);
    Interrupt_output_pin = 1; // 在下降沿启动
    // 启动调度器
    EA = 1;
}
/*
SCI_TICK2_MASTER_Update_T2
这是调度器中断服务程序。被调用的频率取决于 SCI_TICK2_MASTER_Init_T2()
对定时器的设置，本版本中是由定时器 2 的中断触发，定时器自动重新加载。
*/
void SCI_TICK2_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    TF2 = 0; // 必须人工清除
    // 从从节点收到确认消息
    if (SCI_TICK2_MASTER_Process_Ack() == RETURN_ERROR)
    {
        // 没有收到确认消息！
        Error_code_G = ERROR_SCH_LOST_SLAVE;
        // 进入并保持安全状态，直到复位
        SCI_TICK2_MASTER_Enter_Safe_State();
        while(1);
    }
    // 发送定时时标至从节点
    SCI_TICK2_MASTER_Send_Tick_Message();
    // 注意：计算的单位是定时时标数，而非毫秒。
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // 检查是否有一个任务
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
```

```

    {
        // 任务将开始运行
        SCH_tasks_G[Index].RunMe += 1; // 运行标志加 1
        if (SCH_tasks_G[Index].Period)
        {
            // 再次调度周期性任务运行
            SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
        }
    }
    else
    {
        // 还没准备好运行：延迟减 1
        SCH_tasks_G[Index].Delay -= 1;
    }
}
}

// 准备下次的定时时标到来
Interrupt_output_pin = 1;
}

/*-----*
 * SCI_TICK2_MASTER_Send_Tick_Message()
 * 此函数发送时标消息
 * 收到这个消息将在从节点处产生一个中断，并将调用从节点的调度器更新函数
 *-----*/
void SCI_TICK2_MASTER_Send_Tick_Message(void)
{
    // 发送定时时标（下降沿）至从节点
    Interrupt_output_pin = 0;
}

/*-----*
 * SCI_TICK2_MASTER_Process_Ack()
 * 检查从节点是否仍在工作
 *-----*/
bit SCI_TICK2_MASTER_Process_Ack(void)
{
    if (First_call_G)
    {
        // 这是此函数第一次被调用
        First_call_G = 0;
        // 为随后的看门狗引脚检查做准备
        Watchdog_input_previous_G = Slave_watchdog_pin;
    }
    else
    {
        // 每次看门狗引脚都会改变状态
        // 如果从节点的运行正常
        if (Watchdog_input_previous_G == Slave_watchdog_pin)
        {
            // 出错了！
            return RETURN_ERROR;
        }
    }
}

```

```

        }
        // 从节点正常
        Watchdog_input_previous_G = Slave_watchdog_pin;
    }
    // 设置端口为读
    SCI_transfer_port = 0xFF;
    return RETURN_NORMAL;
}
/*-----*
SCI_TICK2_MASTER_Enter_Safe_State()
当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位
(2) 从节点失效
(3) 网络出错
(4) 因为任何其他的原因，确认消息没有按时收到
在这些情况下，把系统置于安全状态
*-----*/
void SCI_TICK2_MASTER_Enter_Safe_State(void) reentrant
{
    // 由用户根据自己的需要编辑
    // 这里显示一个安全输出
    TRAFFIC_LIGHTS_Display_Safe_Output();
}
/*-----*
-----文件结束-----
*-----*/

```

源程序清单 26.8 简单交通信号灯系统的部分软件（主节点）

软件——从节点

下面列出了从节点调度器的核心程序（源程序清单 26.9），其余部分的程序包含在本书所附的 CD 上。

```

/*-----*
SCI_Ti2s.c (v1.00)

这是基于中断的 8051/52 共享时钟调度器程序。
***定时时标 2—从节点***
***使用 1232 看门狗计时器***
***主节点和从节点的时标间隔都是 1ms ***
***参见主节点的代码***
*-----*/
#include "Main.h"
#include "Port.h"
#include "SCI_Ti2s.h"
#include "TLight_A.h"
// -----公有变量声明-----
// 任务数组（参见 Sch51.c）
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

```

```

// 错误代码变量数组 (参见 Sch51.c)
extern tByte Error_code_G;
// -----私有函数原型-----
static void SCI_TICK2_SLAVE_Enter_Safe_State(void);
static void SCI_TICK2_SLAVE_Watchdog_Init(void);
static void SCI_TICK2_SLAVE_Watchdog_Refresh(void) reentrant;
/*-----*/
SCI_TICK2_SLAVE_Init()
调度器初始化函数。准备调度器数据结构并设置定时器中断频率。
使用调度器前必须调用此函数。
*/
void SCI_TICK2_SLAVE_Init(void)
{
    tByte i;
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // SCH_Delete_Task()将生成一错误代码。
    // (因为任务数组是空的)
    Error_code_G = 0;
    // -----外部中断 0-----
    // 从节点是由中断输入驱动的
    // 使能中断
    // 由引脚 P3.2 上的下降沿触发
    IE0 = 1;
    EX0 = 1;
    // 启动看门狗
    SCI_TICK2_SLAVE_Watchdog_Init();
}
/*-----*/
SCI_TICK2_SLAVE_Start()
使能中断，启动从节点调度器
注意：通常在所有的常规任务加入后调用，以保持任务同步。
注意：只能使能调度器中断！！！
*/
void SCI_TICK2_SLAVE_Start(void)
{
    // 将系统置于安全状态
    // 如果主节点没有启动或者启动失败，从节点将返回到这里
    // 或者发生了其他错误
    SCI_TICK2_SLAVE_Enter_Safe_State();
    // 现在处于安全状态
    // 在这里等待第一个定时时标
    // (刷新看门狗以避免重启动)
    while (IE0 == 0)
    {
        SCI_TICK2_SLAVE_Watchdog_Refresh();
    }
}

```

```

        }
    // 清除标志
    IEO = 0;
    // 启动调度器
    EA = 1;
}
/*-----*
SCI_TICK2_SLAVE_Update
这是调度器中断服务程序，其调用频率由 SCI_TICK2_SLAVE_Init_T2()
函数中的定时器设定决定。
从节点由外部中断触发。
*-----*/
void SCI_TICK2_SLAVE_Update(void) interrupt INTERRUPT_EXTERNAL_0
{
    tByte Index;
    // 喂看门狗
    // 主节点将监视此引脚以检查从节点的活动
    SCI_TICK2_SLAVE_Watchdog_Refresh();
    // 现在做标准的调度器更新
    // 注意：计算单位是定时时标数，而非毫秒
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // 检查是否有一个任务
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // 任务将开始运行
                SCH_tasks_G[Index].RunMe += 1; // 运行标志加 1
                if (SCH_tasks_G[Index].Period)
                {
                    // 再次调度周期性任务运行
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // 还没准备好运行：延迟减 1
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}
/*-----*
SCI_TICK2_SLAVE_Watchdog_Init()
此函数设置看门狗计时器
如果主节点失败（或者发生了其他的错误），时标消息没有到达，调度器将停止运行。
为了检测这种情况，从节点上有一个硬件看门狗在运转。
这个看门狗设置为大约 100ms 溢出，将把系统至于安全状态。
然后从节点将一直等待，直到问题得到解决。
*-----*/

```

注意：在这些情况下，从节点不会发送确认消息。

所以主节点（如果在运行）将察觉到出了问题。

```
-----*/
void SCI_TICK2_SLAVE_Watchdog_Init(void)
{
    // 232 外部看门狗不需要初始化
    // 其他的硬件看门狗可能需要
    //
    // 按需要编写相应的初始化程序
}
/*-----*
SCI_TICK2_SLAVE_Watchdog_Refresh()
喂外部(1232)看门狗。超时时间在60~250ms间(取决于硬件)
硬件: 假定为外部1232看门狗
-----*/
void SCI_TICK2_SLAVE_Watchdog_Refresh(void) reentrant
{
    static bit WATCHDOG_state;
    // 改变看门狗引脚的状态
    if (WATCHDOG_state == 1)
    {
        WATCHDOG_state = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state = 1;
        WATCHDOG_pin = 1;
    }
}
/*-----*
SCI_TICK2_SLAVE_Enter_Safe_State()
当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位
(2) 主节点失效
(3) 网络出错
(4) 因为任何其他的原因，时标消息没有按时收到。
在这些情况下，应把系统置于安全状态。
-----*/
void SCI_TICK2_SLAVE_Enter_Safe_State(void)
{
    // 由用户根据自己的需要编辑
    TRAFFIC_LIGHTS_Display_Safe_Output();
}
/*-----*
---文件结束---
-----*/

```

源程序清单 26.9 简单交通信号灯系统的部分软件（从节点）

注意，主节点和从节点之间无法相互传输数据。

进阶阅读

共享时钟中断调度器（数据）

适用场合

- 用多个 8051 微控制器设计一个嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

两个 8051 微控制器通过中断连接在一起，如何调度其上的任务（和传输数据）？

背景知识

参阅第 25 章有关共享时钟调度器的介绍。

解决方案

共享时钟中断调度器（时标）是一个简单、灵活而可靠的调度器。然而，使用这种调度器要求系统做合理的划分，使主节点和从节点之间无需通信，这是因为这个简单的调度器不支持节点间的数据传输。

共享时钟中断调度器（数据）基于共享时钟中断调度器（时标），并且提供了节点间的数据传输能力，所需的基本硬件框架如图 26.11 所示。

和共享时钟中断调度器（时标）一样，这个调度器同步两个控制器上的任务，并使主节点和从节点能够检查出对方或者通信信道的失效。

另外，这个调度器能够利用定时时标消息在主节点和从节点间传输一个字节的信息；同样，也能够利用确认消息在从节点和主节点间传输一个字节的信息。

完整的源程序清单会在后面给出，并说明数据是如何传输的。

硬件资源

和共享时钟中断调度器（时标）一样，这是个高效的调度器。与标准的合作式调度器相比，主节点上的附加软件负荷小得可以忽略不计。

但是，除了每个微控制器上需要两个端口引脚用于定时时标和确认信号，调度器还需要占用两个端口（主节点上一个，从节点上一个）进行数据传输。

可靠性和安全性

许多和 SCI 调度器相关的可靠性和安全性考虑在这里也适用。

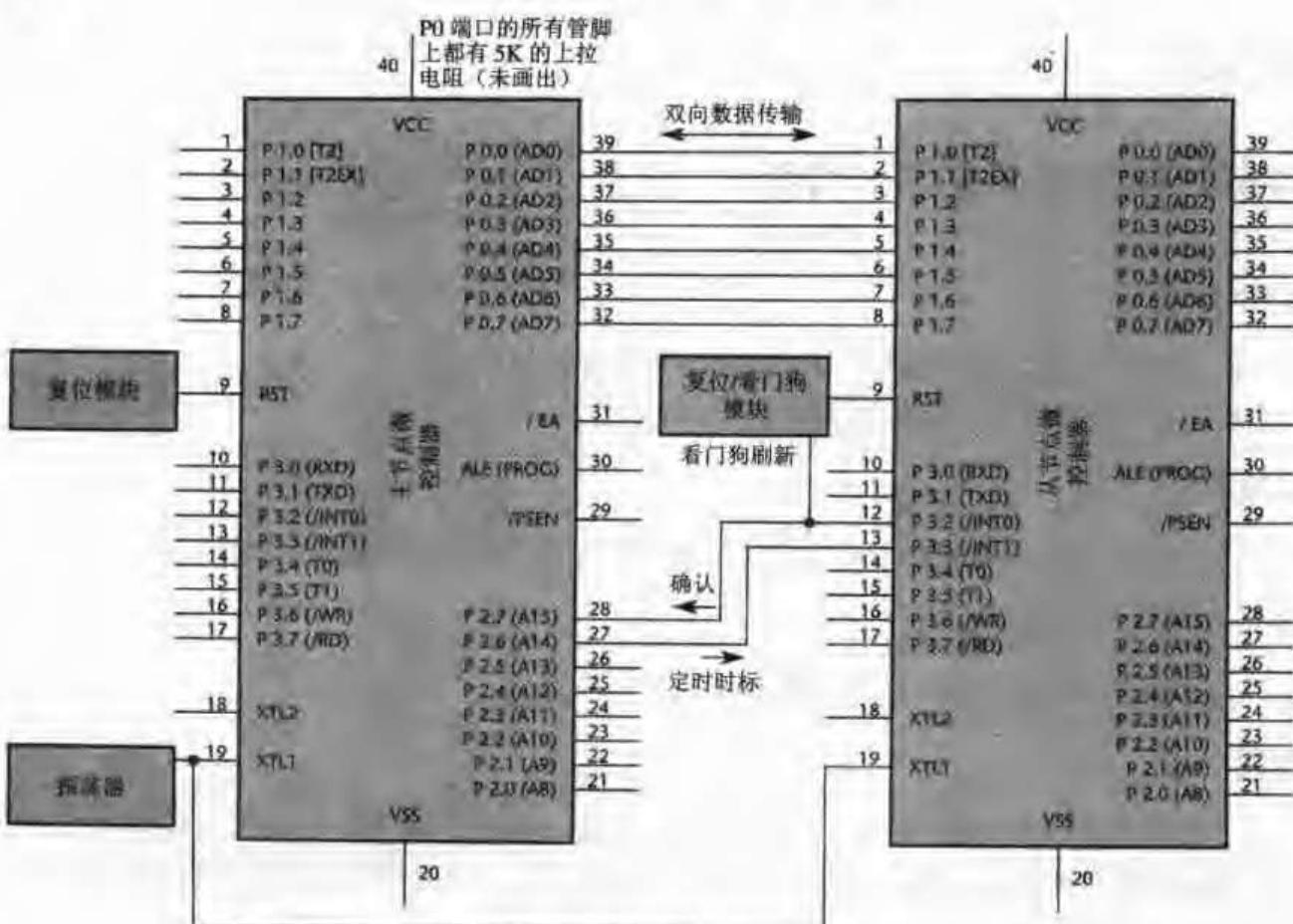


图 26.11 共享时钟(中断)调度器，支持主节点和从节点间的双向数据传输

可移植性

可用于几乎所有的微控制器系列，只要有一个外部中断引脚和一个数据端口就可以。因而，该模式可以很容易地更改用于各种微处理器。

优缺点小结

- ☺ 简单、有效，而且软件开销低。
- ☺ 主节点和从节点间能双向通信。
- ☺ 主节点和从节点间有用于传输数据的双向机制
- ☺ 如果端口没有用于其他用途，每个定时时标可以传输 30 位数据。
- ☹ 系统易于受电磁干扰，除非采取了适当的预防措施，参见共享时钟中断调度器(时标)。

相关的模式和替代方案

本章和第 6 篇的另一个模式提供了连接多个微控制器的替代技术。

例子：交通信号灯

这里再次使用交通信号灯的例子说明同时提供数据和定时时标传输的一些优点。

硬件

所需硬件如图 26.12 所示。

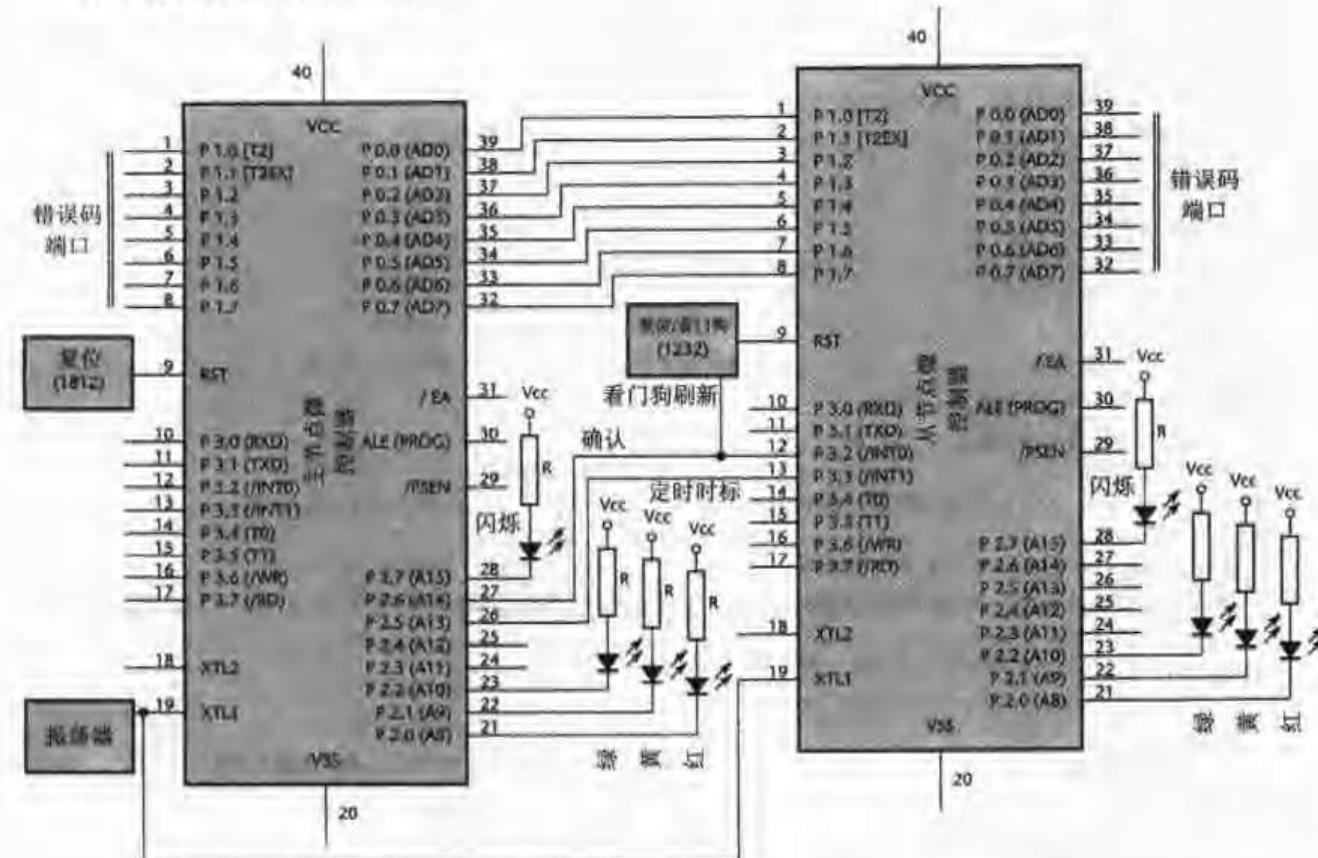


图 26.12 共享时钟中断调度器（数据）版的交通信号灯控制器所需的硬件

注意，为了可靠地工作，端口 0 应使用 10K 上拉电阻。

软件——主节点

下面列出了主节点调度器的核心程序（源程序清单 26.10），其余部分的程序包含在本书所附的 CD 上。

```
/*
 *-----*
 * SCI_Dm.c (v1.00)
 *-----*
 这是 8051/52 的共享时钟中断调度器。
 ***主节点：数据***
 ***T2 用于定时，16 位自动重新装入***
 ***12 MHz 振荡器 ->精确的 1ms 时标间隔***
 */
```

```

---假设从节点上有 1232 看门狗 ---
-----*/
#include "Main.h"
#include "Port.h"
#include "SCI_Dm.H"
#include "Delay_T0.h"
#include "TLight_B.h"
// -----公有变量定义-----
tByte Tick_message_data_G = RETURN_NORMAL;
tByte Ack_message_data_G = RETURN_NORMAL;
// 面用于检测从节点活动的位
bit First_call_G;
bit Watchdog_input_previous_G;
// -----公有变量声明-----
// 任务数组 (参见 Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量 (参见 Sch51.c)
extern tByte Error_code_G;
// 用于从节点发生错误时复位系统 (参见 Main.C)
extern bit System_reset_G;
// -----私有函数原型-----
static void SCI_D_MASTER_Send_Tick_Message(void);
static bit SCI_D_MASTER_Process_Ack(void);
/*-----*
 * SCI_D_MASTER_Init_T2()
 * 调度器初始化函数。准备调度器数据结构并设置定时器中断频率。使用调度器前必须调用此函数。
 */
void SCI_D_MASTER_Init_T2(void)
{
    tByte i;
    // 尚无中断
    EA = 0;
    // -----设置调度器-----
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 清除全局错误变量
    // SCH_Delete_Task()将生成一错误代码
    // (因为任务数组是空的)
    Error_code_G = 0;
    // -----设置定时器 2 (开始) -----
    // 现在设置定时器 2
    // 16 位定时器函数，自动重新装入
    // 晶振频率为 12MHz
    // 定时器 2 的分辨率是 0.000001 秒 (1 微秒)
    // 所需的定时器 2 溢出时间是 0.001 秒 (1ms)
    // 需要 1000 个定时时标
}

```

```

// 重装的值为 65536 - 1000 = 64536 (十进制) = 0xFC18
T2CON = 0x04; // 装入定时器 2 控制寄存器
T2MOD = 0x00; // 装入定时器 2 方式寄存器
TH2 = 0xFC; // 装入定时器 2 高字节
RCAP2H = 0xFC; // 装入定时器重装捕捉寄存器, 高字节
TL2 = 0x18; // 装入定时器 2 低字节
RCAP2L = 0x18; // 装入定时器重装捕捉寄存器, 低字节
ET2 = 1; // 使能定时器 2 中断
TR2 = 1; // 启动定时器 2
// -----设置定时器 2 (结束) -----
}

/*
SCI_D_MASTER_Start()
使能中断, 启动调度器。
注意: 通常在所有的常规任务加入后调用, 以保持任务同步。
注意: 只能使能调度器中断!!!
*/
void SCI_D_MASTER_Start(void)
{
    // 在启动或发生错误后, 应将系统置于安全状态。
    SCI_D_MASTER_Enter_Safe_State();
    // 在这里等待, 以便从节点超时复位
    // 调节等待时间, 使其与从节点的超时时间匹配
    Hardware_Delay_T0(500);
    // 现在发送第一个定时时标, 启动从节点
    // (在下降沿启动)
    Interrupt_output_pin = 1;
    Hardware_Delay_T0(5);
    Interrupt_output_pin = 0;
    Hardware_Delay_T0(5);
    Interrupt_output_pin = 1; // 准备好第一个定时时标
    // 启动调度器
    EA = 1;
}
/*
SCI_D_MASTER_Update_T2
这是调度器中断服务程序。该程序的调用频率由 SCI_D_MASTER_Init_T2()
函数中的定时器设定决定。本版本中由定时器 2 中断触发, 定时器会自动重装。
*/
void SCI_D_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    TF2 = 0; // 必须人工清除
    // 从从节点收到确认消息
    if (SCI_D_MASTER_Process_Ack() == RETURN_ERROR)
    {
        // Error_code_G = ERROR_SCH_LOST_SLAVE;
        // 进入并保持安全状态, 直到复位
        SCI_D_MASTER_Enter_Safe_State();
    }
}

```

```

        while(1);
    }

    // 发送定时时标至从节点
    SCI_D_MASTER_Send_Tick_Message();
    // 注意：计算的单位是定时时标数，而非毫秒。
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // 检查是否有一个任务
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // 任务将开始运行
                SCH_tasks_G[Index].RunMe += 1; // 运行标志加 1
                if (SCH_tasks_G[Index].Period)
                {
                    // 再次调度周期性任务运行
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // 还没准备好运行：延迟减 1
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }

    // 为下次定时时标的到来做准备
    Interrupt_output_pin = 1;
}

/*-----*
 * SCI_D_MASTER_Send_Tick_Message()
 * 此函数发送时标消息,
 * 收到这个消息将在从节点处产生一个中断，并将调用从节点的调度器更新函数。
 *-----*/
void SCI_D_MASTER_Send_Tick_Message(void)
{
    // 将定时时标数据送至端口
    SCI_transfer_port = Tick_message_data_G;
    // 发送定时时标（下降沿）至从节点
    Interrupt_output_pin = 0;
}

/*-----*
 * SCI_D_MASTER_Process_Ack()
 * 检查从节点是否仍在工作
 * 从从节点读出数据。
 *-----*/
bit SCI_D_MASTER_Process_Ack(void)
{
}

```

```

if (First_call_G)
{
    // 这是此函数第一次被调用
    First_call_G = 0;
    // 为随后的看门狗引脚检查做准备
    Watchdog_input_previous_G = Slave_watchdog_pin;
}
else
{
    //
    // 每次看门狗引脚都会改变状态
    // 如果从节点的运行正常
    if (Watchdog_input_previous_G == Slave_watchdog_pin)
    {
        //
        // 出错了!
        return RETURN_ERROR;
    }
    // 从节点正常
    Watchdog_input_previous_G = Slave_watchdog_pin;
}
// 设置端口为读
SCI_transfer_port = 0xFF;
// 读确认消息
Ack_message_data_G = SCI_transfer_port;
return RETURN_NORMAL;
}

/*
SCI_D_MASTER_Enter_Safe_State()
当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位
(2) 从节点失效
(3) 网络出错
(4) 因为任何其他的原因，确认消息没有按时收到
在这些情况下，应把系统置于安全状态。
*/
void SCI_D_MASTER_Enter_Safe_State(void) reentrant
{
    //
    // 由用户根据自己的需要编辑
    // 这里显示一个安全输出
    TRAFFIC_LIGHTS_Display_Safe_Output();
}
/*
-----文件结束-----
*/

```

源程序清单 26.10 简单交通信号灯系统的部分软件（主节点）

注意：本版本支持主节点和从节点间的数据传输。

软件——从节点

下面列出了从节点调度器的核心程序（源程序清单 26.11），其余部分的程序包含在本书所

附的 CD 上。

```
/*
  SCI_Ds.c (v1.00)

这是基于中断的 8051/52 共享时钟调度器程序
***从节点***
***使用 1232 看门狗计时器***
***主节点和从节点的时标间隔都是 1ms***

***参见主节点的代码 ***
*/
#include "Main.h"
#include "Port.h"
#include "SCT_Ds.h"
#include "TLight_B.h"
// -----公有变量定义-----
// 数据从主节点发送至从节点
tByte Tick_message_data_G = RETURN_NORMAL;
// 从该从节点发送至主节点的数据
// 数据可能被主节点转送给另一个从节点
tByte Ack_message_data_G = RETURN_NORMAL;
// -----公有变量声明-----
// 任务数组 (参见 Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量数组 (参见 Sch51.c)
extern tByte Error_code_G;
// -----私有函数原型-----
static void SCI_D_SLAVE_Enter_Safe_State(void);
static void SCI_D_SLAVE_Send_Ack_Message_To_Master(void);
static void SCI_D_SLAVE_Process_Tick_Message(void);
static void SCI_D_SLAVE_Watchdog_Init(void);
static void SCI_D_SLAVE_Watchdog_Refresh(void) reentrant;
/*-----*
SCI_D_SLAVE_Init()
调度器初始化函数。准备调度器数据结构并设置定时器中断频率。
使用调度器前必须调用此函数。
*-----*/
void SCI_D_SLAVE_Init(void)
{
    tByte i;
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // SCH_Delete_Task() 将生成一错误代码。
```

```
// (因为任务数组是空的)
Error_code_G = 0;
// -----外部中断 0-----
// 从节点是由中断输入驱动的
// 使能中断
// 由引脚 P3.2 的下降沿触发
IT0 = 1;
EX0 = 1;
// 启动看门狗
SCI_D_SLAVE_Watchdog_Init();
}

/*
SCI_D_SLAVE_Start()
使能中断，启动从节点调度器
注意：通常在所有的常规任务加入后调用，以保持任务同步。
注意：只能使能调度器中断！！！
*/
void SCI_D_SLAVE_Start(void)
{
    // 将系统置于安全状态
    // 从节点将返回到这里，如果主节点没有启动或者启动失败。
    SCI_D_SLAVE_Enter_Safe_State();
// 现在处于安全状态
    // 在这里等待第一个定时时标
    // (刷新看门狗以避免重启)
    while (IE0 == 0)
    {
        SCI_D_SLAVE_Watchdog_Refresh();
    }
    // 清除标志
    IE0 = 0;
    // 启动调度器
    EA = 1;
}
/*
SCI_D_SLAVE_Update
这是调度器中断服务程序，其调用频率由 SCI_D_SLAVE_Init() 函数中的定时器设定决定。
从节点由外部中断触发。
*/
void SCI_D_SLAVE_Update(void) interrupt INTERRUPT_EXTERNAL_0
{
    tByte index;
    // 提取定时时标消息数据
    SCI_D_SLAVE_Process_Tick_Message();
    // 数据回传至主节点
    SCI_D_SLAVE_Send_Ack_Message_To_Master();
    // 喂看门狗
    SCI_D_SLAVE_Watchdog_Refresh();
```

```

// 现在做标准的调度器更新
// 注意：计算的单位是定时时标数，而非毫秒。
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
    // 检查是否有一个任务
    if (SCH_tasks_G[Index].pTask)
    {
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // 任务将开始运行
                SCH_tasks_G[Index].RunMe += 1; // 运行标志加 1
                if (SCH_tasks_G[Index].Period)
                {
                    // 再次调度周期性任务运行
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // 还没准备好运行；延迟减 1
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}

/*-----*
SCI_D_SLAVE_Send_Ack_Message_To_Master()
收到时标消息后，从节点必须向主节点发送确认消息。
确认消息起两个作用：
[1]使主节点确认从节点仍然有效并且运行良好。
[2]提供向主节点传输数据的途径。
-*-----*/
void SCI_D_SLAVE_Send_Ack_Message_To_Master(void)
{
    SCI_Transfer_Port = Ack_message_data_G;
}

/*-----*
SCI_D_SLAVE_Process_Tick_Message()
时标消息对共享时钟调度器的运行至关重要：每隔一定间隔到达的时标消息调用
更新中断服务程序，驱动调度器。
时标消息自身可能包含数据，这些数据由此函数提取。
-*-----*/
void SCI_D_SLAVE_Process_Tick_Message(void)
{
    // 设置端口为读
    SCI_Transfer_Port = 0xFF;
}

```

```
// 读数据
Tick_message_data_G = SCI_Transfer_Port;
}

/*
SCI_D_SLAVE_Watchdog_Init()
此函数设置看门狗计时器
如果主节点失败（或者发生了其他的错误），时标消息没有到达，调度器将停止运行。
为了检测这种情况，从节点上有一个硬件看门狗在运转。这个看门狗设置为大约 100ms 溢出，
应把系统至于安全状态。然后从节点将一直等待，直到问题得到解决。
注意：在这些情况下，从节点不会发送确认消息。所以主节点（如果在运行）将察觉到出了问题
*/
void SCI_D_SLAVE_Watchdog_Init(void)
{
    // 1232 外部看门狗不需要初始化
    // 其他的硬件看门狗可能需要
    //
    // 按需要编写相应的初始化程序
}

/*
SCI_D_SLAVE_Watchdog_Refresh()
喂外部（1232）看门狗、超时时间在 60~250ms 间（取决于硬件）
硬件：假定为外部 1232 看门狗
*/
void SCI_D_SLAVE_Watchdog_Refresh(void) reentrant
{
    static bit WATCHDOG_state;
    // 改变看门狗引脚的状态
    if (WATCHDOG_state == 1)
    {
        WATCHDOG_state = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state = 1;
        WATCHDOG_pin = 1;
    }
}

/*
SCI_D_SLAVE_Enter_Safe_State()
当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位
(2) 主节点失效
(3) 网络出错
(4) 因为其他的原因，时标消息没有按时收到
在这些情况下，应把系统置于安全状态。
*/
void SCI_D_SLAVE_Enter_Safe_State(void)
```

```
{  
    // 由用户根据自己的需要编辑  
    TRAFFIC_LIGHTS_Display_Safe_Output();  
}  
/*-----*  
---文件结束---  
-*-----*/
```

源程序清单 26.11 简单交通信号灯系统的部分软件（从节点）

注意，本版本支持主节点和从节点间的数据传输。

进阶阅读

使用 UART（通用异步收发器）的共享时钟调度器

引言

正如在第 18 章讨论的，对 8051 系列的微控制器，基于 UART 进行数据传输是一种简单的设计方案。可以将信息从嵌入式系统传输至台式个人计算机，反之亦然。本章中将举例说明，对于简单、低成本的两个或多个微控制器的通信设计中，可以用 UART 作为通信硬件基础。

注意，这类模式对本地和分布式网络都适用。对于本地网络，一般的系统包含几个微控制器，相互之间距离仅仅几厘米；大多数情况下，微控制器被装在同一个机箱里。实际上，它们经常被安装在同一块 PCB（印制电路板）上。在分布式网络中，微控制器之间的距离可能相当远，例如 RS-485 网络，可能节点间的距离长达一千米，如果每一千米增加一块中继板，甚至可以达到更远的传输距离。

使用 UART 的共享时钟调度器（本地）

适用场合

- 用多个 8051 系列的微控制器开发一个嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

两个（或更多的）8051 微控制器通过 UART 组成的本地网进行通信，如何调度其上的任务（和数据传输）？

背景知识

一般的背景材料参见第 25 章的共享时钟调度器。

解决方案

在本节中，讨论如何设计基于 UART 的共享时钟调度器。

方法的基础

书中的所有共享时钟调度器都有一样的底层结构（如图 27.1 所示）。

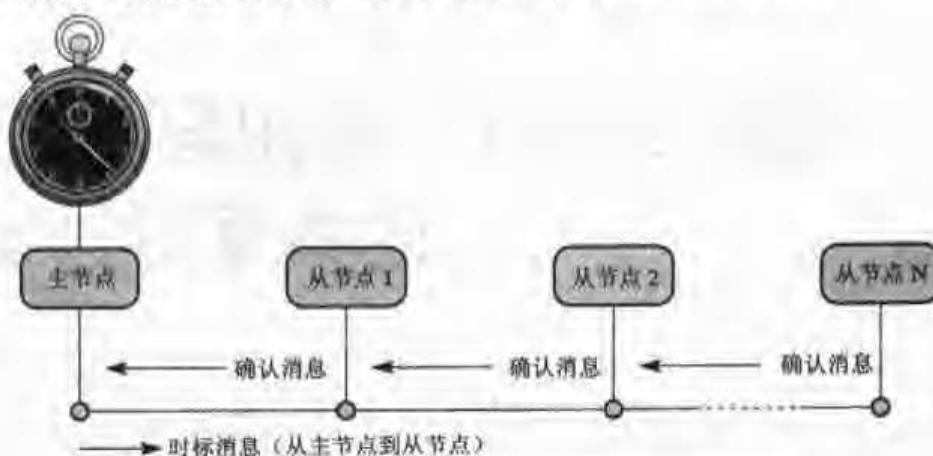


图 27.1 书中讨论的所有共享时钟调度器的体系结构

在网络上，主节点处有一个精确的时钟。这个时钟按第 3 篇中讲述的方式驱动主节点的调度器。

从节点也有调度器，用于驱动这个调度器的中断来自于主节点产生的定时消息（如图 25.8 所示）。这样，在基于 UART 网络的模式中，从节点的共享时钟调度器将由接收到主节点数据产生的接收中断驱动。

注意，当从节点的 UART 收到一个数据字节时，会产生一个中断，这使得底层调度器能以非常简单的两阶段过程运作：

1. 主节点定时器溢出，调用调度器的更新函数，于是一个字节的数据（经 UART）发送至所有从节点：

```

void MASTER_Update_T2 ( void ) interrupt INTERRUPT_Timer_2_Overflow{
    ...
    MASTER_Send_Tick_Message ( ... );
    ...
}
  
```

2. 所有的从节点在收到这个数据时都会产生一个中断，这将调用从节点调度器的更新函数。其中一个从节点将向主节点发回一个确认消息（通过 UART）。

```

void SLAVE_Update ( void ) interrupt INTERRUPT_UART_RX_Tx{
    ...
    SLAVE_Send_Ack_Message_To_Master ();
    ...
}
  
```

消息结构

基于 UART 的网络使用的消息结构设计有些需要注意的地方。

例如，看一下图 27.2。假定希望控制和监视三个液压传动器，以控制工程挖掘机的操作。

假设希望将传动装置 A 调整为 90°，怎样才能办到？UART 的 8 位数据长度成了一个限制，因为需要发送的消息要同时指定接收节点和所需的调整角度。

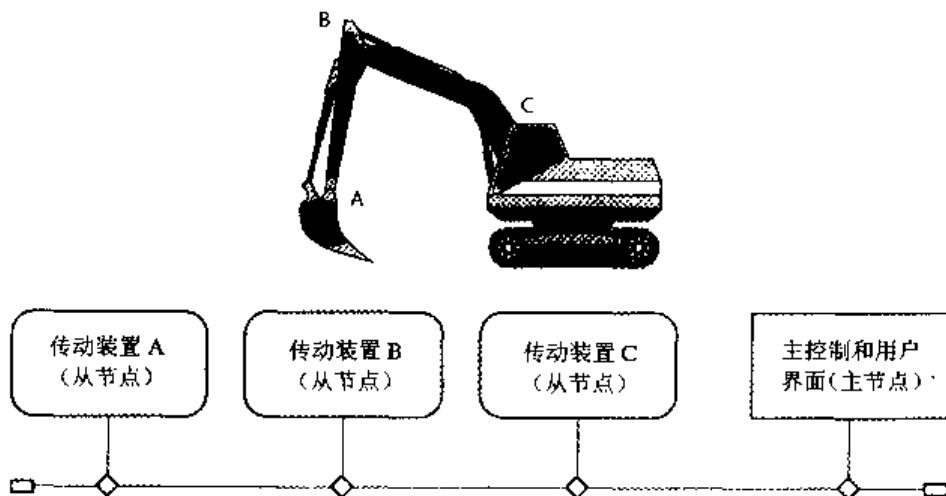


图 27.2 控制和监视工程挖掘机的操作，从节点的位置为 A、B、C，主节点在驾驶舱

没有理想的方法可以解决这个问题，这里采用以下的解决方法：

- 每个从节点有一个唯一的标识符（0x01~0xFF）。
- 每个主节点发出的定时消息长两个字节，这两个字节在一个定时间隔内发送。第一个字节是一个“地址字节”，包含接收从节点的标识符。第二个字节是消息字节，包含消息数据。
- 对每个定时消息的每个字节，所有的从节点都会产生中断响应。
- 只有当前定时消息寻址的从节点才会回答主节点，答复采取确认消息的形式。
- 每个从节点发送的确认消息长两个字节，这两个字节也是在一个定时间隔内发送。第一个字节是一个“地址字节”，包含发送消息的从节点的标识符。第二个字节是消息字节，包含消息数据。
- 对于超过单个字节的数据传输，必须发送多条消息（所用技术参见数据联合）。

图 27.3 说明了消息是如何传输的。

还有一个进一步的问题：为了避免可能出现的混乱，需要区分地址字节和数据字节。

利用 8051 的 9 位串行数据传输功能可以解决这一问题（如表 27.1 所示）。

在这种配置下（UART 一般被设置为模式 3），一次发送/接收 11 位。注意，第 9 位是通过寄存器 SCON 的 TB8 位传输的，接收时从同一寄存器的 RB8 位读取。在这个模式下，波特率的控制方法和第 18 章中讨论的一样。

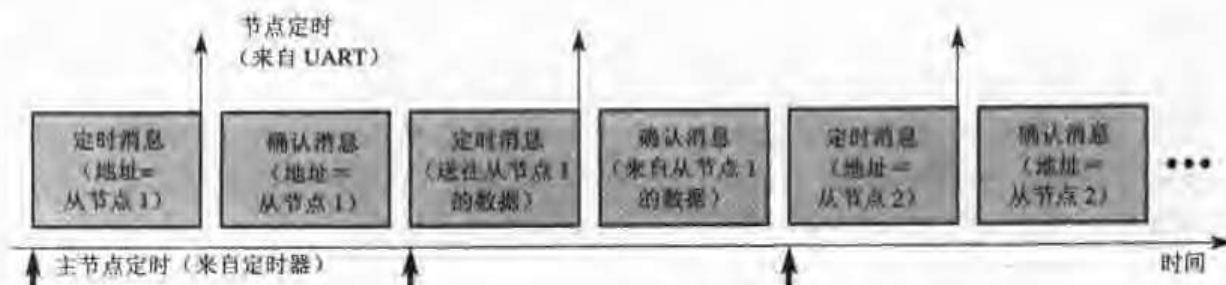


图 27.3 基于 UART 的共享时钟网络中主节点和两个从节点间的通信

表 27.1 9 位串行消息的结构

Description	Size(bits)
Data	9 bits
Start bit	1 bit
Stop bit	1 bit
Total	11 bits/message

地址字节通过设置命令位 (TB8) 为 1 来识别；数据字节设置 TB8 为 0，参见下文和本书所附 CD 中的代码例子。

确定所需的波特率

主节点定时器的定时应设置得足够长，足以发送一个字节的定时消息和接收一个字节的确认消息。显然，定时的长短取决于通信的波特率。

如上文所述，9 位通信协议的每个消息包括了起始和停止位，每个定时间隔共需要收发 22 位（定时消息需要 11 位，确认消息需要 11 位）；也就是说，所需的波特率为：调度器每秒的定时时标数 $\times 22$ 。

例如，如果定时间隔为 1ms，相当于波特率最低为 22 000 波特，标准的 28 800 波特率提供了很好的安全裕度。

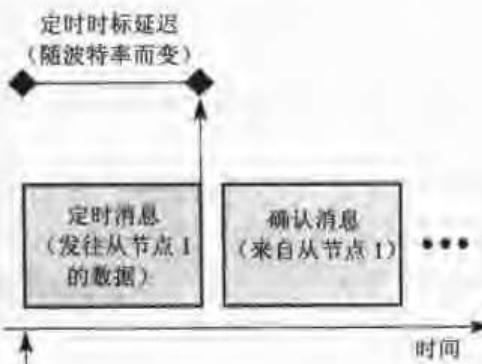


图 27.4 主节点和从节点之间的定时时标延迟

注意，使用标准的波特率在调试时很有用，因为（例如）用个人计算机可以很容易地监听网络通信。可是，标准的波特率并不是必需的，任何与所有的网络节点兼容的波特率都能用。而且这样做还有好处；例如，设置为模式 2 的 UART 不需要使用定时器就可以支持高速的 9 位串行通信。如，12MHz 晶振，每机器周期 12 次振荡的 8051 可以用模式 2 生成 375 000 波特的信号。

同样还要注意到，和任何共享时钟网络一样，主节点的定时器和从节点的 UART 接收中断间存在一个延迟（如图 27.4 所示）。延迟的原因是因为从节点的中断是在定时消息的末尾才产生的。当不存在网络错误时，这个延迟是固定的，而且能由 UART 传送一个字节所需的时间大致算出；也就是说，这个延迟随波特率而变。正如先前讨论过的，大多数的共享时钟应用至少使用 28 800 的波特率，此时的延迟大约为 0.4ms。对于 375 000 波特而言，延迟大约为 0.03ms。

注意，这个延迟是固定的，而且可以从理论上精确地预计，并通过模拟和测试确认。如果需要在主节点和从节点间精确同步，请注意：

- 所有的从节点操作应尽可能地保持步调一致。
- 为了让主节点和从节点步调一致，只需在主节点的更新函数中增加一个短延时。

节点硬件

典型的网络节点硬件如图 27.5a 和图 27.5b 所示。

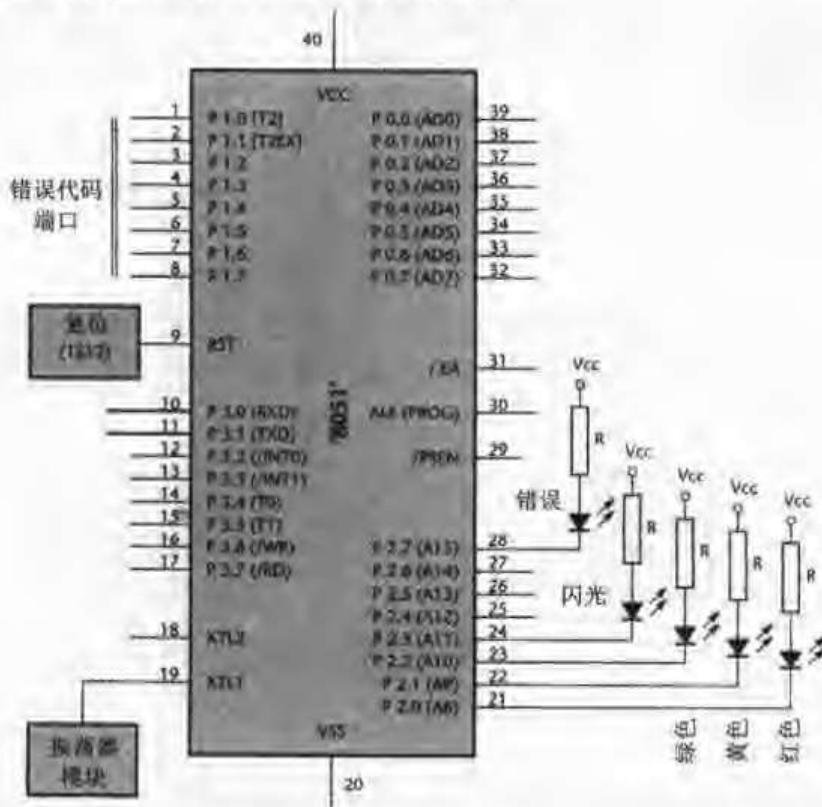


图 27.5a 本地 SCU 网络的硬件，使用外部 1232 看门狗

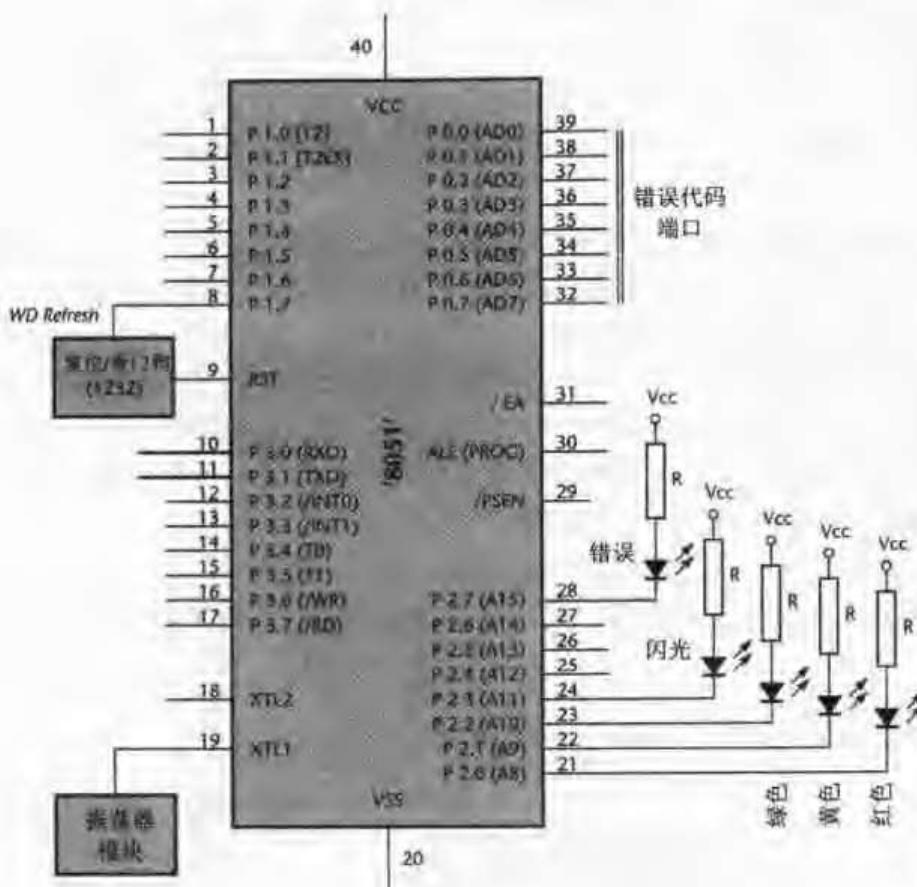


图 27.5b 本地 SCU 网络的硬件，假定片上有看门狗

网络接线

图 27.6 示范了一个两节点的网络该如何连接，而图 27.7 示范了包含三个或更多从节点的网络的连接方法。注意，所有的电缆都应尽可能的短，最理想的是所有的微控制器位于同一块印制电路板上。

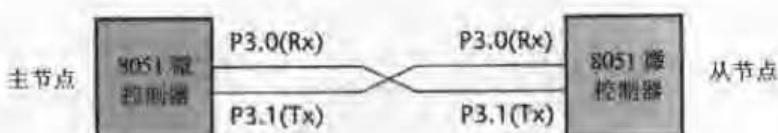


图 27.6 两节点（本地）SCU 调度器的节点间互联

硬件资源

显而易见，主要的硬件资源是 UART；还可以利用本模式使用 UART 向个人计算机传送数据，相关的技术参见本模式的第二个例子。

注意，本模式不一定需要定时器作为波特率发生器，细节请参见解决方案一节。

可靠性和安全性

许多和 SCI 调度器相关的可靠性和安全性考虑在这里也适用。

可移植性

能应用于整个 8051 系列和其他微控制器/微处理器/DSP。

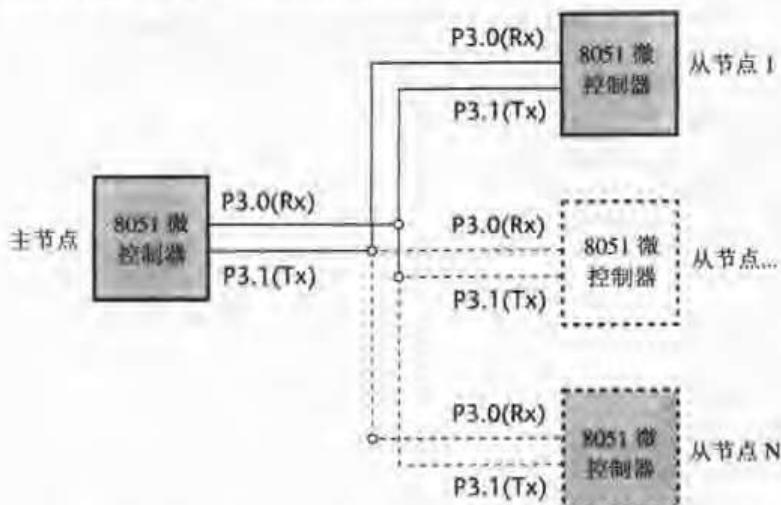


图 27.7 三个节点(本地)SCU 调度器的节点间互联

优缺点小结

- ⊕ 对有两个或更多 8051 微控制器的本地系统是一个简单的调度器。
- ⊕ 所有必需的硬件都是 8051 内核的一部分，因此，该方法在 8051 系列微控制器间有很好的可移植性。
- ⊕ 易于用最小的 CPU 和存储开销实现。
- ⊕ UART 只支持字节通信：每两个定时时标之间，主节点和从节点之间只能传输 0.5 字节。
- ⊖ 占用了一个重要的硬件资源——UART。
- ⊖ 必须由软件进行大多数的错误检测/修正。
- ⊖ 这个模式不适用于分布式系统。

相关的模式和替代方案

参见 SCU 调度器，其中给出了一个适用于多点网络的体系结构，能在网络节点间传输大量的数据。

例子：一个本地 UART 调度器的程序库

下面给出了一个简单的代码库（源程序清单 27.1 和源程序清单 27.2）。相应的硬件可使用图 27.5 中所示的设计方案。

主节点软件

```
/*
SCU_Am.c (v1.00)

这是一个 SCU (使用 UART) 调度器 (本地) 的 8051/52 实现方案。
和一个 SCU (使用 UART) 调度器 (RS-232) 的 8051/52 实现方案。
*** 主节点 ***
*** 检查从节点的确认消息 ***
*** 本地/RS-232 版 (无使能支持) ***
*** 使用 1232 看门狗计时器 ***
*** 主节点和从节点的定时间隔都是 5ms ***
*/
#include "Main.h"
#include "Port.h"

#include "SCU_Am.H"
#include "Delay_T0.h"
#include "TLight_B.h"
// ----- 公有变量定义 -----
tByte Tick_message_data_G[NUMBER_OF_SLAVES] = {'M'};
tByte Ack_message_data_G[NUMBER_OF_SLAVES];
// ----- 公有变量声明 -----
// 任务数组 (参见 Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量 (参见 Sch51.c)
extern tByte Error_code_G;
// ----- 私有变量定义 -----
static tByte Current_slave_index_G = 0;
static bit First_ack_G = 1;
static bit WATCHDOG_state_G = 0;
// ----- 私有函数原型 -----
static void SCU_A_MASTER_Reset_the_Network(void);
static void SCU_A_MASTER_Shut_Down_the_Network(void);
static void SCU_A_MASTER_Switch_To_Backup_Slave(const tByte);
static void SCU_A_MASTER_Send_Tick_Message(const tByte);
static bit SCU_A_MASTER_Process_Ack(const tByte);
static void SCU_A_MASTER_Watchdog_Init(void);
static void SCU_A_MASTER_Watchdog_Refresh(void) reentrant;
// ----- 私有常数 -----
// 从节点 ID 可以是任意非零 tByte 值 (必须互不相同)
// 注意: ID 0x00 是用于错误处理的。
static const tByte MAIN_SLAVE_IDS[NUMBER_OF_SLAVES] = {0x31};
static const tByte BACKUP_SLAVE_IDS[NUMBER_OF_SLAVES] = {0x31};
```

```
#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)
// 可调整(如果需要的话)以增加网络复位的间隔时间
// (当网络发生错误时)
#define SLAVE_RESET_INTERVAL 0U
// -----私有变量定义-----
static tWord Slave_reset_attempts_G[NUMBER_OF_SLAVES];
// 参见上面的MAIN_SLAVE_IDS[]
static tByte Current_Slave_IDs_G[NUMBER_OF_SLAVES] = {0};
static bit Message_byte_G = 1;
/*-----*
SCU_A_MASTER_Init_T1_T2()
调度器初始化函数。准备调度器数据结构并设置定时器中断频率。使用调度器前必须调用此函数。
BAUD_RATE -所需的波特率。
-----*/
void SCU_A_MASTER_Init_T1_T2(const tWord BAUD_RATE)
{
    tByte Task_index;
    tByte Slave_index;
    // 尚无中断
    EA = 0;
    // 启动看门狗
    SCU_A_MASTER_Watchdog_Init();
    Network_error_pin = NO_NETWORK_ERROR;
    // -----设置调度器-----
    // 清理所有任务
    for (Task_index = 0; Task_index < SCH_MAX_TASKS; Task_index++)
    {
        SCH_Delete_Task(Task_index);
    }
    // 清除全局错误变量
    // SCH_Delete_Task()将生成一错误代码
    // (因为任务数组是空的)
    Error_code_G = 0;
    // 对从节点, ID可任意取值
    // -但是ID 0x00必须保留
    for (Slave_index = 0; Slave_index < NUMBER_OF_SLAVES; Slave_index++)
    {
        Slave_reset_attempts_G[Slave_index] = 0;
        Current_Slave_IDs_G[Slave_index] = MAIN_SLAVE_IDS[Slave_index];
    }
    // -----设置波特率(开始)-----
    PCON &= 0x7F; // 设置SMOD位为0(不加倍波特率)
    // 使能接收
    // *9位数据*, 1位起始位, 1位停止位, 波特率可变(异步)
    SCON = 0xD2;
    TMOD |= 0x20; // T1设为模式2, 8位自动重新加载
    TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
        / ((tLong)BAUD_RATE * OSC_PER_INST * 1000)));
}
```

```

TL1 = TH1;
TR1 = 1; // 启动定时器
TI = 1; // 发送第一个字符(空字符)
// -----设置波特率(结束) -----
// 串口中断没有使能
ES = 0;
//-----建立串行连接(结束) -----
// -----设置定时器2(开始) -----
// 晶振频率为12MHz, 每振荡12次加1
// 定时器2的分辨率是0.000001s(1微秒)
// 所需的定时器2溢出时间是0.005s(5ms)
// 需要5000个时标
// 重装的值为65536 - 5000 = 60536(十进制) = 0xEC78
T2CON = 0x04; // 装入定时器2控制寄存器
T2MOD = 0x00; // 装入定时器2方式寄存器
TH2 = 0xEC; // 装入定时器2高字节
RCAP2H = 0xEC; // 装入定时器重装捕捉寄存器, 高字节
TL2 = 0x78; // 装入定时器2低字节
RCAP2L = 0x78; // 装入定时器重装捕捉寄存器, 低字节
ET2 = 1; // 使能定时器2中断
TR2 = 1; // 启动定时器2
// -----设置定时器2(结束) -----
}

/*-----*
SCU_A_MASTER_Start()
使能中断, 启动调度器。
注意: 通常在所有的常规任务加入后调用, 以保持任务同步。
注意: 只能使能调度器中断!!!
注意: 假设波特率至少为9600, 延迟2ms。
如果选择了更低的波特率, 则需要调整代码。
*-----*/
void SCU_A_MASTER_Start(void)
{
    tByte Slave_ID;
    tByte Num_active_slaves;
    tByte Id, i;
    bit First_byte = 0;
    bit Slave_replied_correctly;
    tByte Slave_index;
    // 刷新看门狗
    SCU_A_MASTER_Watchdog_Refresh();
    // 将系统置于安全状态
    SCU_A_MASTER_Enter_Safe_State();
    // 在等待启动时报告错误
    Network_error_pin = NETWORK_ERROR;
    Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
    SCH_Report_Status(); // 调度器尚未运行——人工处理
    // 暂停3000ms, 使所有的从节点超时
    // (这么做是为了同步整个网络)
}

```

```
for (i = 0; i < 100; i++)
{
    Hardware_Delay_T0(30);
    SCU_A_MASTER_Watchdog_Refresh();
}
// 此时和所有从节点的连接断开
Num_active_slaves = 0;
// 在初始的长延迟之后，所有运行的从节点将已经超时
// 所有的工作从节点现在将处于准备启动状态
// 发送一个单字节消息启动这些从节点
// 连着做两遍，带命令位
// 在正常情况下，这不可能发生
Slave_index = 0;
do {
// 刷新看门狗
    SCU_A_MASTER_Watchdog_Refresh();
// 清除第一字节标志
    First_byte = 0;
// 找到该从节点的ID
    Slave_ID = (tByte) Current_Slave_IDS_G[Slave_index];
// 发送一个从节点ID消息
    TI = 0;
    TB8 = 1; // 置命令位
    SBUF = Slave_ID;
// 等待从节点回答
    Hardware_Delay_T0(5);
// 检查是否有了回答
    if (RI == 1)
    {
        // 取回答数据
        Id = (tByte) SBUF;
        RI = 0;
        // 检查回答，带命令位
        if ((Id == Slave_ID) && (RB8 == 1))
        {
            First_byte = 1;
        }
    }
// 发送第二个字节
    TI = 0;
    TB8 = 1;
    SBUF = Slave_ID;
// 等待从节点回答
    Hardware_Delay_T0(5);
// 检查是否有了回答
    Slave_replied_correctly = 0;
    if (RI != 0)
    {
        // 取回答数据
    }
}
```

```

Id = (tByte) SBUF;
RI = 0;
if ((Id == Slave_ID) && (RB8 == 1) && (First_byte == 1))
{
    Slave_replied_correctly = 1;
}
}
if (Slave_replied_correctly)
{
    // 从节点应答正确
    Num_active_slaves++;
    Slave_index++;
}
else
{
    // 从节点没有正确应答
    // -设法切换到备用节点（如果有的话）
    if (Current_Slave_IDS_G[Slave_index] !=
        = BACKUP_SLAVE_IDS [Slave_index])
    {
        // 有后备节点可用：切换到后备节点，再试一次
        Current_Slave_IDS_G[Slave_index]=
        BACKUP_SLAVE_IDS [Slave_index];
    }
    else
    {
        // 没有可用的后备节点（或者后备节点也失效了）-不得不继续
        // 注意：为了更加灵活，这里没有退出循环
        // 以下是错误处理
        Slave_index++;
    }
}
} while (Slave_index < NUMBER_OF_SLAVES);
// 处理无响应的从节点...
if (Num_active_slaves < NUMBER_OF_SLAVES)
{
// 用户自定义的错误处理...
// 一个或多个的从节点没有应答
// 注意：在某些情况下，如果从节点无响应，可能需要退出
// -或者重新配置网络
// 这里的错误处理方法是显示出错并关闭网络
    Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
    SCU_A_MASTER_Shut_Down_the_Network();
}
else
{
    Error_code_G = 0;
    Network_error_pin = NO_NETWORK_ERROR;
}

```

```

// 准备好发送第一个定时消息
Message_byte_G = 1;
First_ack_G = 1;
Current_slave_index_G = 0;
// 启动调度器
EA = 1;
}
/* -----
SCU_A_MASTER_Update_T2
这是调度器中断服务程序。该程序的调用频率由 SCU_A_MASTER_Init_T2() 函数中的定时器
设定决定。本版本是由定时器 2 的中断触发：定时器将自动重新加载。
-----*/
void SCU_A_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
tByte Task_index;
tByte Previous_slave_index;
TF2 = 0; // 必须人工清除
// 刷新看门狗
SCU_A_MASTER_Watchdog_Refresh();
// 默认
Network_error_pin = NO_NETWORK_ERROR;
// 用于记录当前从节点
// 第一个值是 0
Previous_slave_index = Current_slave_index_G;
// 假设发送和接收 2 字节的消息
// -每个消息需要两个定时标传递
if (Message_byte_G == 0)
{
Message_byte_G = 1;
}
else
{
Message_byte_G = 0;
if (++Current_slave_index_G >= NUMBER_OF_SLAVES)
{
Current_slave_index_G = 0;
}
}
// 检查是否是正确的从节点应答了上述的消息：
// (如果是的，存储该从节点发送的数据)
if (SCU_A_MASTER_Process_Ack(Previous_slave_index) == RETURN_ERROR)
{
Network_error_pin = NETWORK_ERROR;
Error_code_G = ERROR_SCH_LOST_SLAVE;
// If we have lost contact with a slave, we attempt to
// 如果和一个从节点失去联系，复位网络后尝试切换到备用节点（如果有的话）
// 注意：并不在每个定时这样做，否则网络将被不断地复位。
}

```

```

    // 选择 SLAVE_RESET_INTERVAL 的值，每 5s 复位一次。
    if(++Slave_reset_attempts_G[Previous_slave_index] >=
        SLAVE_RESET_INTERVAL)
    {
        SCU_A_MASTER_Reset_the_Network();
    }
}
else
{
// 复位计数器
    Slave_reset_attempts_G[Previous_slave_index] = 0;
}
// 发送定时时标消息至所有联络上的从节点
// (发送一个数据字节给当前从节点)
SCU_A_MASTER_Send_Tick_Message(Current_slave_index_G);
// 注意：计算的单位是定时时标数，而非毫秒。
for (Task_index = 0; Task_index < SCH_MAX_TASKS; Task_index++)
{
// 检查是否有任务
    if (SCH_tasks_G[Task_index].pTask)
    {
        if (SCH_tasks_G[Task_index].Delay == 0)
        {
// 任务将开始运行
            SCH_tasks_G[Task_index].RunMe += 1; // 运行标志加 1
            if (SCH_tasks_G[Task_index].Period)
            {
// 再次调度周期性任务运行
                SCH_tasks_G[Task_index].Delay
                = SCH_tasks_G[Task_index].Period;
            }
        }
        else
        {
// 还没准备好运行：延迟减 1
            SCH_tasks_G[Task_index].Delay -= 1;
        }
    }
}
/*
SCU_A_MASTER_Send_Tick_Message()
此函数在 USART (通用同步-异步收发器) 网络上发送一个定时消息。
定时消息的接收会在微控制器上产生中断
在从节点侧：会调用调度器的更新函数。
*/
void SCU_A_MASTER_Send_Tick_Message(const tByte SLAVE_INDEX)
{
    tLong Timeout;

```

```

// 找到该从节点的 ID
tByte Slave_ID = (tByte) Current_Slave_IDS_G[SLAVE_INDEX];
// 根据下标，每次发送一个字节的数据
// 如果 Message_byte_G 是 0，发送第一个字节（从节点的 ID）
if (Message_byte_G == 0)
{
    Timeout = 0;
    while ((++Timeout) && (TI == 0));
    if (Timeout == 0)
    {
        // UART 无应答——出错了
        Error_code_G = ERROR_USART_TI;
        return;
    }
    TI = 0;
    TB8 = 1; // 设置命令位
    SBUF = Slave_ID;
}
else
{
    // Message_byte_G 是 1，发送数据字节
    Timeout = 0;
    while ((++Timeout) && (TI == 0));
    if (Timeout == 0)
    {
        // USART（通用同步收发器）无应答——出错了
        Error_code_G = ERROR_USART_TI;
        return;
    }
    TI = 0;
    TB8 = 0;
    SBUF = Tick_message_data_G[SLAVE_INDEX];
}
// 发送数据——返回
}
/*-----*
SCU_A_MASTER_Process_Ack()
确认从节点 (SLAVE_ID) 应答了上面发送的消息。如果有应答，从 USART 读取消息数据；如果没有，调用相应的错误处理程序。
Slave_index - 从节点的索引
返回： RETURN_NORMAL - 收到了确认 (Ack_message_data_G 中的数据)
      RETURN_ERROR - 没有收到确认 (没有数据)
-----*/
bit SCU_A_MASTER_Process_Ack(const tByte Slave_index)
{
    tByte Message_contents;
    tByte Slave_ID;
    // 第一次调用时，没有确认定时消息可供检查
    // - 返回"OK"即可
}

```

```

if (First_ack_G)
{
    First_ack_G = 0;
    return RETURN_NORMAL;
}
// 找到该从节点的 ID
Slave_ID = (tByte) Current_Slave_IDs_G[Slave_index];
// 数据应该已经在缓冲区中了
if (RI == 0)
{
    // 从节点没有应答上次的定时消息
    // 设置出错 LED
    Network_error_pin = NETWORK_ERROR;
    return RETURN_ERROR;
}
// 有数据——读取之
Message_contents = (tByte) SBUF;
RI = 0;
// 这是对上次消息的应答
// -反向解释消息字节
if (Message_byte_G == 1)
{
    // 检查命令位是否设置了

    if (RB8 == 1)
    {
        // 检查 ID 是否正确
        if (Slave_ID == (tByte) Message_contents)
        {
            // 所需的确认消息已经收到
            return RETURN_NORMAL;
        }
    }
    // 什么地方出错了...
}
// 设置出错 LED
Network_error_pin = NETWORK_ERROR;
return RETURN_ERROR;
}
else
{
    // 有可用的数据
    // 从从节点消息中提取数据
    //
    // 注意：假定数据是正确的
    // -对重要的数据可能需要发送两次
    Ack_message_data_G[Slave_index] = Message_contents;
    return RETURN_NORMAL; // 返回从节点数据
}
}

```

```

/*
SCU_A_MASTER_Reset_the_Network()
此函数复位(即重新启动)整个网络。
*/
void SCU_A_MASTER_Reset_the_Network(void)
{
    EA = 0; // 禁止中断
    while(1); // 看门狗将超时
}

/*
SCU_A_MASTER_Shut_Down_the_Network()
此函数关闭整个网络。
*/
void SCU_A_MASTER_Shut_Down_the_Network(void)
{
    EA = 0; // 禁止中断
    Network_error_pin = NETWORK_ERROR;
    SCH_Report_Status(); // 调度器尚未运行——人工处理
    while(1)
    {
        SCU_A_MASTER_Watchdog_Refresh();
    }
}

/*
SCU_A_MASTER_Enter_Safe_State()
当系统发生下列情况时,进入此状态:
(1) 节点上电或者复位
(2) 主节点检测不到一个从节点
(3) 网络出错
在这些情况下,应把系统置于安全状态。
*/
void SCU_A_MASTER_Enter_Safe_State(void)
{
    // 由用户根据自己的需要编辑
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*
SCU_A_MASTER_Watchdog_Init()
此函数设置看门狗计时器。
如果主节点失效(或者发生了其他的错误),收不到定时消息,调度器将停止运行。
为了检测这种情况,从节点上有一个硬件看门狗在运转。这个看门狗设置为大约100ms溢出,
用于把系统置于安全状态。然后从节点将一直等待,直到问题得到解决。
注意:在这些情况下,从节点不会发送确认消息。所以主节点(如果在运行)将察觉到出了问题。
*/
void SCU_A_MASTER_Watchdog_Init(void)
{
    // 1232外部看门狗不需要初始化
    // 其他的硬件看门狗可能需要
    // 按需要编写相应的初始化程序
}

```

```

    }

/*
SCU_A_MASTER_Watchdog_Refresh()
喂外部(1232)看门狗
超时时间在60~250ms间(取决于硬件)
硬件:假定为外部1232看门狗
*/
void SCU_A_MASTER_Watchdog_Refresh(void) reentrant
{
    //改变看门狗引脚的状态
    if (WATCHDOG_state_G == 1)
    {
        WATCHDOG_state_G = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state_G = 1;
        WATCHDOG_pin = 1;
    }
}
/*
---文件结束---
*/

```

源程序清单 27.1 一个本地 UART 调度器的部分库代码(主节点)

从节点软件

```

/*
SCU_As.c (v1.00)

This is an implementation of SCU SCHEDULER (LOCAL) for 8051/52.
AND an implementation of SCU SCHEDULER (RS-232) for 8051/52.

*** SLAVE NODE ***
*** Local / RS-232 version (no 'enable' support) ***
*** Uses 1232 watchdog timer ***
*** Assumes 12 MHz osc (same as Master) ***
*** Both Master and Slave share the same tick rate (5 ms) ***
*** - See Master code for details ***

#include "Main.h"
#include "Port.h"
#include "SCU_As.h"
#include "TLight_B.h"
// -----公有变量定义-----
// 从该从节点发送至主节点的数据
tByte Tick_message_data_G;
// 从该从节点发送至主节点的数据
// -数据可能被主节点转送给另一个从节点

```

```

tByte Ack_message_data_G = 'S';
// -----公有变量声明-----
// 任务数组 (参见 Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
//错误代码变量数组 (参见 Sch51.c)
extern tByte Error_code_G;
// -----私有函数原型-----
static void SCU_A_SLAVE_Enter_Safe_State(void);
static void SCU_A_SLAVE_Send_Ack_Message_To_Master(void);
static tByte SCU_A_SLAVE_Process_Tick_Message(void);
static void SCU_A_SLAVE_Watchdog_Init(void);
static void SCU_A_SLAVE_Watchdog_Refresh(void) reentrant;
// -----私有常数 -----
//每个从节点必须有一个独一无二的非零 ID
#define SLAVE_ID 0x31
#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)
// -----私有变量 -----
static bit Message_byte_G;
static bit WATCHDOG_state_G = 0;
static tByte Message_ID_G = 0;
/*-----*
SCU_A_SLAVE_Init_T1()
调度器初始化函数。准备调度器数据结构并设置定时器中断频率。
使用调度器前必须调用此函数。
BAUD_RATE - 所需的波特率
-----*/
void SCU_A_SLAVE_Init_T1(const tWord BAUD_RATE)
{
    tByte i;
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // SCH_Delete_Task() 将生成一错误代码。
    // (因为任务数组是空的)
    Error_code_G = 0;
    // 设置网络出错引脚 (收到定时消息后复位)
    Network_error_pin = NETWORK_ERROR;
    // 准备接收第一个定时消息
    Message_byte_G = 1;
    // -----设置波特率 (开始) -----
    PCON &= 0x7F; // 设置SMOD位为0 (不加倍波特率)
    // 使能接收
    // *9位数据*, 1位起始位, 1位停止位, 波特率可变 (异步)
    SCON = 0xD2;
    TMOD |= 0x20; // T1 设为模式2, 8位自动重新加载
}

```

```

TH1 = (256 - (tByte) (((tLong)OSC_FREQ / 100) * 3125)
      / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));
TL1 = TH1;
TR1 = 1; // 启动定时器
TI = 1; // 发送第一个字符(空字符)
// -----设置波特率(结束) -----
// 开放中断
// (接收和发送一个字节都将产生一个串行中断)
// 全局中断还没有开放
ES = 1;
// 启动看门狗
SCU_A_SLAVE_Watchdog_Init();
}

/*-----*
SCU_A_SLAVE_Start()
使能中断, 启动从节点调度器
注意: 为保持任务同步, 通常在所有的常规任务加入后调用。
注意: 只能使能调度器中断!!!
-*-----*/
void SCU_A_SLAVE_Start(void)
{
    tByte Command = 0;
    tByte Message_byte;
    tByte Count = 0;
    bit Slave_started = 0;
    // 禁止中断
    EA = 0;
    // 程序会因为以下原因运行到这里:
    // 1. 网络刚刚上电
    // 2. 主节点出错, 停止生成定时时标
    // 3. 网络损坏, 从节点收不到定时时标
    // 设法使系统处于安全状态
    SCU_A_SLAVE_Enter_Safe_State();
    // 注意: 在这里中断被禁止
    Count = 0;
    Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
    SCH_Report_Status(); // 调度器尚未运行——人工处理
    // 现在等待主节点的信号(无限期地)
    do {
        // 等待定时消息(字节1)的接收, 所有位设置为0
        do {
            SCU_A_SLAVE_Watchdog_Refresh(); // 必须不间断地喂看门狗
            } while (RI == 0);
            Message_byte = (tByte) SBUF;
            RI = 0;
        // 连续收到两个ID消息
        // (带命令位)
            // 确认这两个消息
        if ((Message_byte == (tByte) SLAVE_ID) && (RB8 == 1))

```

```

{
Count++;
//这个从节点接收到的消息——发送确认
TI = 0;
TB8 = 1; // 置命令位
SBUF = (tByte) SLAVE_ID;
}
else
{
Count = 0;
}
} while (Count < 2);
// 启动调度器
EA = 1;
}

/*-----*
SCU_A_SLAVE_Update
这是调度器中断服务程序。该程序的调用频率由 SCU_A_SLAVE_Init() 函数中的定时器设定决定。
从节点由 USART(通用同步-异步收发器) 中断触发。
*-----*/
void SCU_A_SLAVE_Update(void) interrupt INTERRUPT_UART_Rx_Tx
{
tByte Index;
if (RI == 1) // 必须检查这里
{
// 默认
Network_error_pin = NO_NETWORK_ERROR;
// 发送(确认)和接收(定时时标)2字节的消息
// -每个消息需要两个定时时标的时间处理
//
// 用于跟踪当前字节
if (Message_byte_G == 0)
{
Message_byte_G = 1;
}
else
{
Message_byte_G = 0;
}
// 检查定时时标数据—必要时发送确认消息
// 注意：启动消息只在超时后发送。
if (SCU_A_SLAVE_Process_Tick_Message() == SLAVE_ID)
{
SCU_A_SLAVE_Send_Ack_Message_To_Master();
// 喂看门狗(当收到正确的消息后)
// (总线上的噪音等等不会使看门狗停止运转)
}
}
}

```

```

// 启动消息不会刷新从节点
// -必须定时与每个从节点通信
SCU_A_SLAVE_Watchdog_Refresh();
}

// 注意：计算的单位是定时时标数，而非毫秒
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
    // 检查是否有一个任务
    if (SCH_tasks_G[Index].pTask)
    {
        if (SCH_tasks_G[Index].Delay == 0)
        {

// 任务将开始运行
SCH_tasks_G[Index].RunMe = 1; // 置运行标志
if (SCH_tasks_G[Index].Period)
{
    {
        // 再次调度周期性任务运行
        SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
    }
}
else
{
}

// 还没准备好运行：减少延迟
SCH_tasks_G[Index].Delay -= 1;
}
}
}

RI = 0; // 复位 RI 标志
}
else
{
}

// 中断服务程序在最后一个字节发送后产生的 TI 标志触发调用
// 必须清除 TI 标志
TI = 0;
}

}

/*
SCU_A_SLAVE_Send_Ack_Message_To_Master()
收到定时消息后，从节点必须向主节点发送确认消息。
注意：只有发给此从节点的定时消息才会被确认。
确认消息起两个作用：
[1]使主节点确认从节点仍然有效并且运行良好。
[2]提供向主节点（进而从节点）传输数据的途径。
注意：从节点间的直接数据传输是不可能的。
*/
void SCU_A_SLAVE_Send_Ack_Message_To_Master(void)

```

```

{
// 根据下标，每次发送一个字节的数据
// 如果 Message_byte_G 是 0，发送第一个字节（从节点的 ID）
if (Message_byte_G == 0)
{
    TI = 0;
TB8 = 1; // 设置命令位
SBUF = SLAVE_ID;
}
else
{
    // Message_byte_G 是 1，发送数据字节
    TI = 0;
    TB8 = 0;
    SBUF = Ack_message_data_G;
}
// 发送数据——返回
}
/*-----*
SCU_A_SLAVE_Process_Tick_Message()
定时消息对共享时钟调度器的运转至关重要：定时消息的接收将调用更新中断服务程序，进而驱动调度器。
定时消息中可能包含数据。这些数据由此函数提取。
*-----*/
tByte SCU_A_SLAVE_Process_Tick_Message(void)
{
    tByte Data;
    // 取出数据字节
    if (RI == 0)
    {
        // 没有数据——什么地方出错了...
        // 设置出错标志位
        Network_error_pin = NETWORK_ERROR;
        // 返回从节点 ID 0
        return 0x00;
    }
    // 有可用的数据
    Data = (tByte) SBUF;
    RI = 0; // 清除 RI 标志
    // 如何处理消息取决于该消息是第一个还是第二个字节
    if (Message_byte_G == 0)
    {
        // 这应该是 ID 字节
        Message_ID_G = Data;
        if (RB8 == 0)
        {

```

```

        Message_ID_G = 0; // 应该设置了命令位
    }
}
else
{
// 这应该是数据字节
// 应该没有设置命令位
if ((Message_ID_G == SLAVE_ID) && (RB8 == 0))
{
    Tick_message_data_G = Data;
}
else
{
// 什么地方出错了——将 Message_ID 设置为 0
Message_ID_G = 0;
// 设置出错标志位
Network_error_pin = NETWORK_ERROR;
}
}
return Message_ID_G;
}

```

/*-----*

SCU_A_SLAVE_Watchdog_Init()

此函数设置看门狗计时器

如果主节点失败（或者发生了其他的错误），定时消息没有到达，调度器将停止运行。

为了检测这种情况，从节点上有一个硬件看门狗在运转。这个看门狗设置为大约 100ms 溢出，用于把系统置于安全状态。然后从节点将一直等待，直到问题得到解决。

注意：在这些情况下，从节点不会发送确认消息。所以主节点（如果在运行）将察觉到出了问题。

-----*/

void SCU_A_SLAVE_Watchdog_Init(void)

{

// 1232 外部看门狗不需要初始化

// 其他的硬件看门狗可能需要

// 按需要编写相应的初始化程序

}

-----*/

SCU_A_SLAVE_Watchdog_Refresh()

喂外部 1232 看门狗。

超时时间在 60~250ms 间（取决于硬件）

硬件：假定为外部 1232 看门狗

-----*/

void SCU_A_SLAVE_Watchdog_Refresh(void) reentrant

{

// 改变看门狗引脚的状态

if (WATCHDOG_state_G == 1)

{

```

WATCHDOG_state_G = 0;
WATCHDOG_pin = 0;
}
else
{
WATCHDOG_state_G = 1;
WATCHDOG_pin = 1;
}
}

/*
SCU_A_SLAVE_Enter_Safe_State()
当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位
(2) 主节点失效，而且没有可用的工作着的后备节点
(3) 网络出错
(4) 因为任何其他的原因，定时消息没有按时收到。
在这些情况下，把系统置于安全状态。
*/
void SCU_A_SLAVE_Enter_Safe_State(void)
{
// 由用户根据自己的需要编辑
TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*
---文件结束---
*/

```

源程序清单 27.2 一个本地 UART 调度器的部分库程序（从节点）

例子：一个 3 节点的本地调度器

源程序见本书所附 CD。

例子：一个带后备从节点的 2 节点本地调度器

源程序见本书所附 CD。

例子：增加一个 UART (通用异步收发器)

正如第 18 章所述，使用调度用的 RS-232 口在 PC 和嵌入式系统间传输数据是很简单的（如图 27.8 所示）。

假设，基于 UART 的调度器连接的两个微控制器，需要和一台 PC 通信；在大多数情况下，没有空余的 UART 可用于和 PC 通信。

有几种可能的解决方法：

- 使用有多个 UART 的 8051，例如，某些 Dallas 生产的微控制器具有一个以上的 UART，Infineon 生产的 c517 和 c509 也有这样的特性。
- 增加一个外部 UART，如 Maxim Max3110E（带 SPI 接口）。
- 使用 CAN 将节点连接在一起，UART 留作和 PC 通信，详见 scc 调度器。
- 组合使用 UART 调度器和中断调度器（连到一块板子上），参见图 27.10。

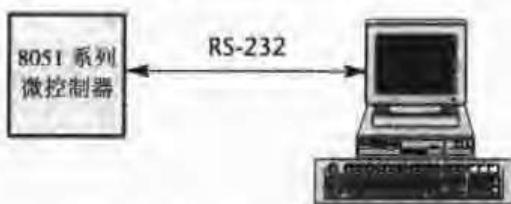


图 27.8 用 RS-232 和台式 PC 互传数据



图 27.9 只有单 UART 的微控制器，如何在一条 RS-232 链路上用基于 UART 的调度器方案和另一个微控制器通信，并且和 PC 互传数据

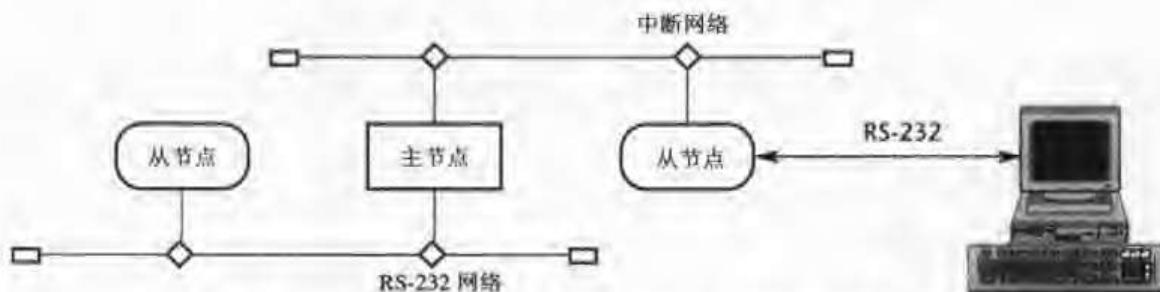


图 27.10 组合使用 UART 调度器和中断调度器

进阶阅读

使用 UART 的共享时钟调度器 (RS-232)

适用场合

- 用多个 8051 系列的微控制器开发一个嵌入式系统。

- 该系统为基于调度器的时间触发体系结构。

问题

两个 8051 微控制器通过 RS-232 分布式网络连接在一起，如何调度其上的任务（和传输数据）？

背景知识

RS-232 协议的背景资料参见第 18 章，所需的软件参见 SCU 调度器（本地）。

解决方案

本模式扩展了 SCU（使用 UART 的）调度器（本地）的设计：两个模式间的关键差异在于 SCU 调度器（RS-232）能够用 RS-232 兼容的收发器建立 2 节点的分布式网络。图 27.11 展示了一种可能的节点配置，其中使用了一个外部 1232 看门狗。

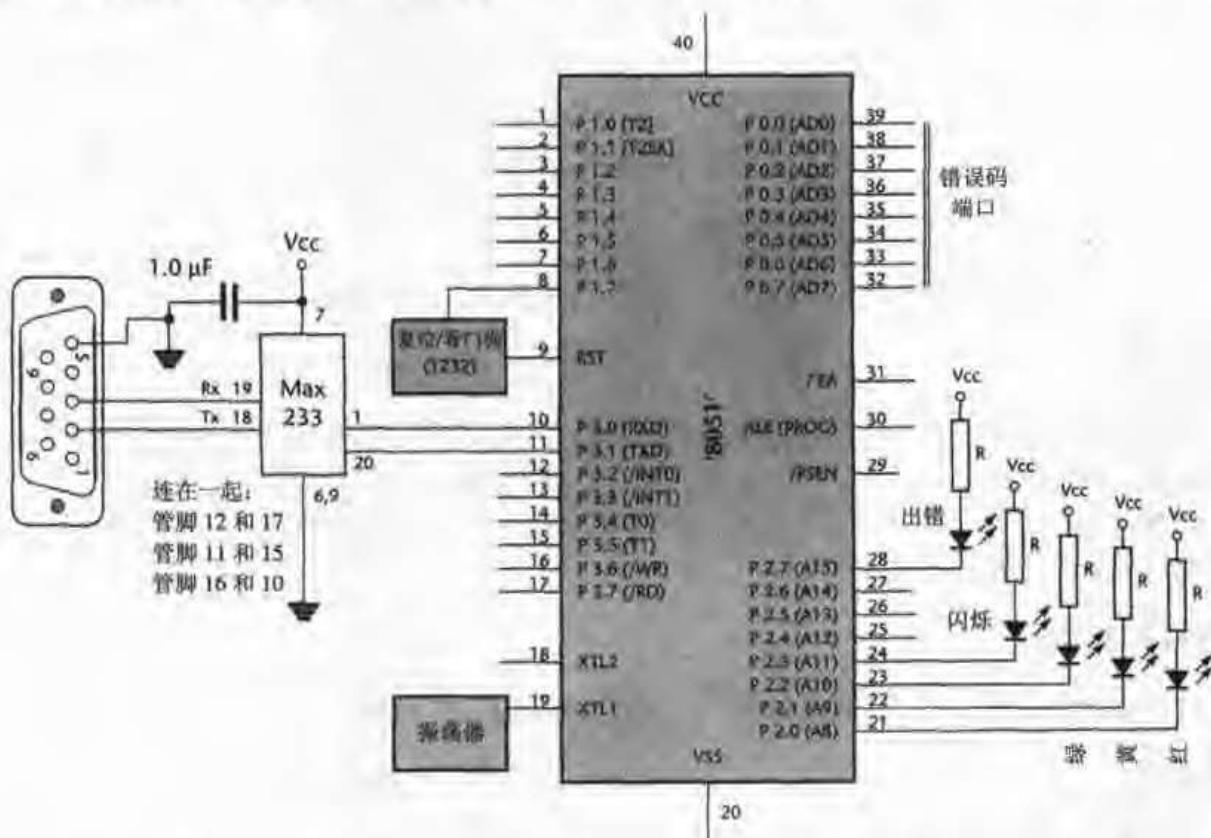


图 27.11 基于 UART 的调度器的一个节点配置例子，其中使用了外部 1232 看门狗

图 27.12 展示了另一种节点配置，其中使用的是片上看门狗。

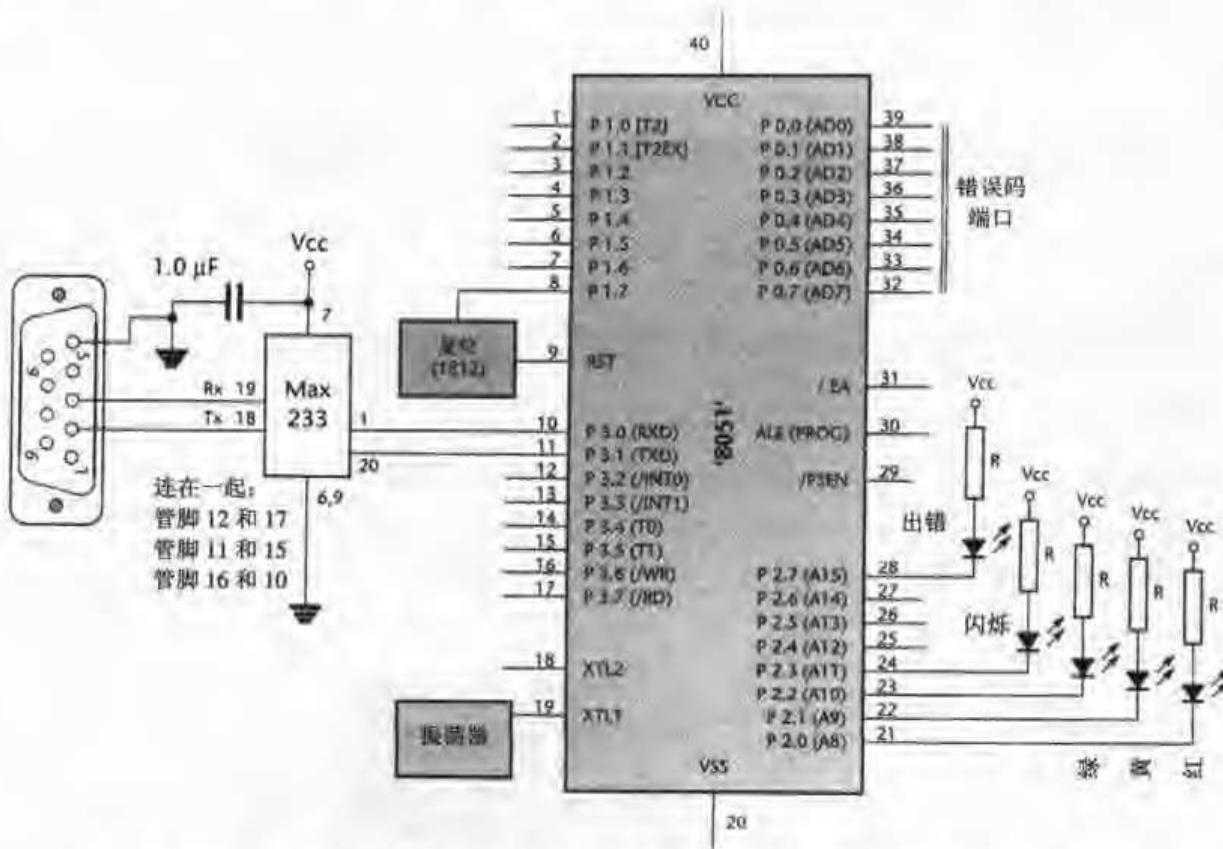


图 27.12 基于 UART 的调度器的一个节点配置例子，其中使用了内部看门狗

硬件资源

显而易见，主要的（微控制器）硬件资源是 UART；本模式还可以用 UART 向个人计算机传送数据，相关的技术参见 SCU（使用 UART 的）调度器（本地）的第二个例子。

注意，本模式不一定需要定时器作为波特率发生器，细节请参见 SCU（使用 UART 的）调度器（本地）的解决方案一节。

可靠性和安全性

许多和 SCI 调度器相关的可靠性和安全性考虑在这里也适用。

可移植性

能在整个 8051 系列和其他许多微控制器/微处理器/DSP 上应用。

优缺点小结

- ⑤ 对两个 8051 微控制器组成的分布式系统是一个简单的调度器。
- ⑤ 所有必需的硬件都是 8051 内核的一部分（收发器除外），因此，该方法在 8051 系列

的微控制器间有很好的可移植性。

- ◎ 易于用最小的 CPU 和存储开销实现。
- ◎ UART 只支持字节通信：每两个定时时标之间，主节点和从节点之间只能传输 0.5 字节。
- ◎ 占用了一个重要的硬件资源——UART。
- ◎ 必须由软件进行大多数的错误检测/修正。

相关的模式和替代方案

适用于多点网络的一个类似调度器参见 SCU 调度器 (RS-485)。

参见 SCC 调度器，其中给出了一个适用于多点网络的体系结构，能在网络节点间传输大量的数据。

例子：交通信号灯

任何 SCU (使用 UART 的) 调度器 (本地) 中的两节点的例子在这里都适用，硬件平台可以基于图 27.11 或者图 27.12 中的设计。

进阶阅读

Axelson, J. (1998) *Serial Port Complete*, Lakeview Research.

使用 UART 的共享时钟调度器 (RS-485)

适用场合

- 用多个 8051 系列的微控制器开发一个嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

两个 (或更多的) 8051 微控制器通过 RS-485 分布式网络连接在一起，如何调度其上的任务 (和传输数据) ?

背景知识

共享时钟调度器的关键特性已经在第 25 章讨论过了。

解决方案

RS-485 通信标准是多点通信体系的一个电气规范，应用于本模式时意味着设计至少包含三个节点，每个节点一个微控制器。

注意，RS-485 的标准文件 EIA/TIA-485-A 只定义了线路、驱动器和接收器的电气特性，这是该标准的一个重要限制。所以和 RS-232 不同，RS-485 的标准中没有规定软件协议或连接器规格。

RS-232 协议在 SCU 调度器（RS-232）和 PC 连接（RS-232）讨论过。RS-485 和 RS-232 通信协议有很多相似之处：

- 两者都是串行标准。
- 两者都被广泛应用。
- 在本书的应用中，两者都需要使用适当的收发器，并连接到 UART。
- 两者的软件库很相似。

RS-485 和 RS-232 也有一些重要的差异：

- RS-232 是单线标准（每个信道一根信号线，再加上地）。环境电噪声可能破坏数据传输。这使得通信的最大距离被限制为 30 米左右，通信速度被限制在 115kbaud 左右（对于较新的收发器）。
- RS-485 是一个双线或者差分通信标准。这意味着每个信道有两根线，一根传送正相信号，一根传送反相信号。接收器会检测出两条线路上的电压差。电噪声会同时影响两条线路，并在接收器检测电压差时抵消，因而，RS-485 网络可以延伸到 1 千米远，同时数据传输速率仍达 90kbaud。如果距离较短（15 米以内），数据传输速率可以更高（达 10MB/s）。
- RS-232 是点对点通信标准。对本书来说，这意味着 RS-232 适合于两个节点的系统，每个节点包含一个微控制器（或者如第 18 章的应用，其中一个节点是台式计算机或类似的个人计算机）。
- RS-485 是一个多点通信标准。大的 RS-485 网络可以支持 32 个节点的负载，如果使用高阻抗接收器，甚至可以达到 256 个节点。
- RS-232 使用廉价的平行电缆，用 3 根线就能实现全双工（发送，接收，地线）。
- 要想达到最好的性能，RS-485 必须使用双绞线缆，其中包含一个双绞线对、地线（通常还有屏蔽层）。这种电缆比 RS-232 所使用的电缆要粗重而且价格更贵。
- RS-232 电缆不需要终端电阻。
- RS-485 电缆通常需要在线路两端的节点处并联一个 120Ω 的终端电阻（假设使用的是 24-AWG 双绞线），如图 27.13 所示。终端电阻能减少电压反射，而电压反射可能会导致接收器误判逻辑电平。
- RS-232 收发器简单而标准。
- RS-485 收发器的选择取决于具体应用。对于基本系统而言，常用的选择是 Maxim^① 的 Max489 系列收发器（参见图 27.13 和图 27.14）。如果需要更高的可靠性，可以选

① www.maxim-ic.com

用 Linear Technology^② 的 LTC1482、National Semiconductors^③ 的 DS36276 和 Maxim 的 Max3080-89 系列收发器，这些收发器都包含电缆短路的保护电路。Maxim 的 MAX1480 包含变压器隔离的供电回路和光电隔离的信号回路，这有助于避免供电线和网络线之间的感应损坏微控制器。

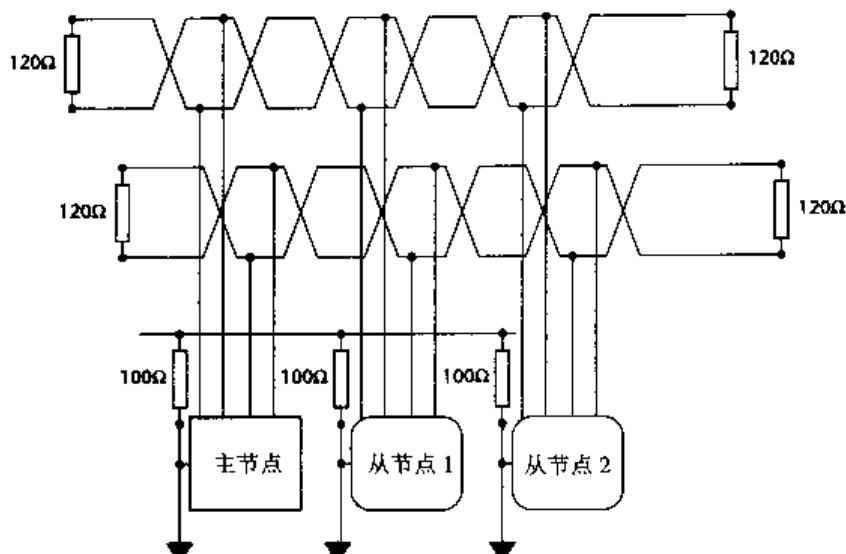


图 27.13 一种可行的 RS-485 网络连线（双绞线）

注意，RS-485 标准推荐在每个节点的信号地线和网络的接地线间串联一个 100Ω 的电阻（至少 $0.5W$ ）。这有助于限制当节点地电位变化时，流过地线的电流。

软件设计考虑：使能输入

此模式所需的软件几乎和 SCU（使用 UART 的）调度器（本地）所用的软件一样，惟一的例外是多节点系统需要控制 RS-485 收发器的使能信号，这是因为同一时间只能有一个收发器在网络上发送。

共享时钟调度器的时间触发特性使得收发器的激活非常简单，详见随后的源代码例子。

硬件资源

显而易见，主要的（微控制器）硬件资源是 UART。本模式还可以用 UART 向个人计算机传送数据，相关的技术参见 SCU（使用 UART 的）调度器（本地）的第二个例子。

注意，本模式不一定要将定时器作为波特率发生器，细节请参见 SCU（使用 UART 的）调度器（本地）的解决方案的内容。

可靠性和安全性

关于使用多微控制器可能会减少系统可靠性的讨论参见第 435 页。

② www.linear-tech.com

③ www.national.com

不过，如果分布式的设计是必需的，RS-485 的差分通信特性会让设计具有较好的抗干扰性。

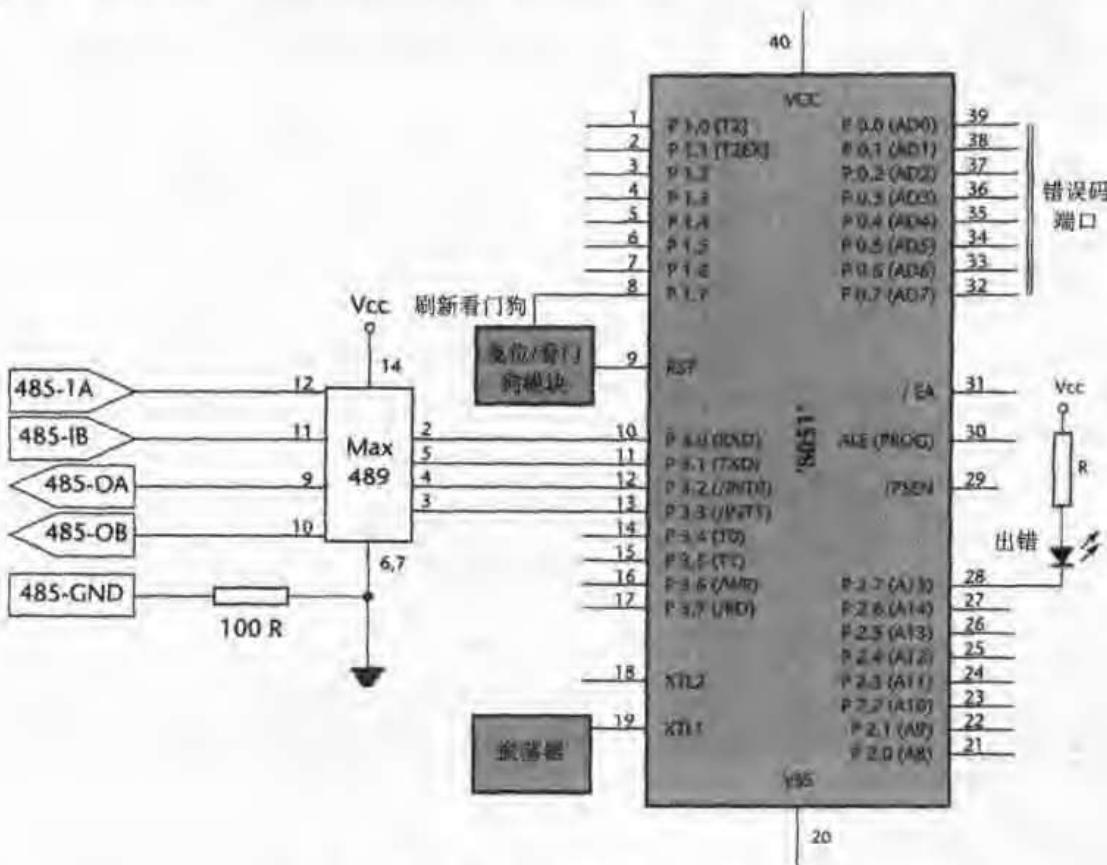


图 27.14 RS-485 网络的一个简单节点

可移植性

此模式应用于整个 8051 系列和其他许多微控制器/微处理器/DSP。

优缺点小结

- ⑤ 对多个 8051 微控制器组成的分布式系统而言是一个简单的调度器。
- ⑤ 易于用较低的 CPU 和存储开销实现。
- ⑤ 双绞线电缆和差分信号比基于 RS-232 的方案更加可靠。
- ⑤ UART 只支持字节通信：每两个定时时标之间，主节点和从节点之间只能传输 0.5 字节。
- ⑧ 占用了一个重要的硬件资源——UART。
- ⑧ 硬件监测错误的能力依然很有限，大多数的错误检测/纠正必须由软件进行。

相关的模式和替代方案

参见SCC调度器，其中给出了一个适用于多点网络体系结构的替代方案，能在网络节点间传输大量的数据。

例子：使用Max489收发器的网络

在这个例子中，将介绍适用于图27.15中网络节点的SCU调度器(RS-485)的代码框架(源程序清单27.3和源程序清单27.4)。

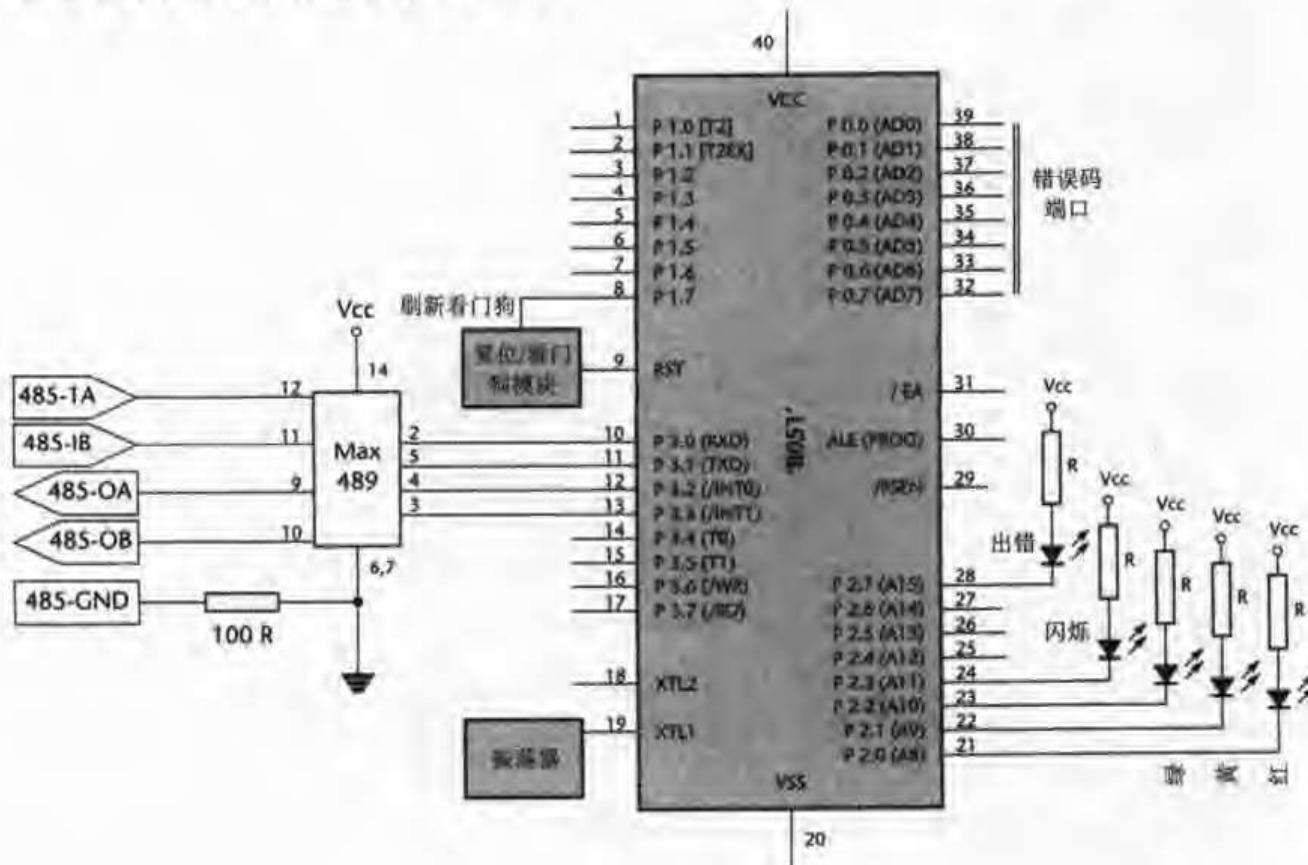


图27.15 RS-485网络的一个节点

主节点软件

```
/*
 * SCU_Bm.c (v1.00)
 *
 * 这是一个SCU(使用UART)调度器(RS-485)的8051/52实现方案。
 * ***主节点***
 * ***检查从节点的确认消息***
 * ***包括支持收发器使能***
 * ---假定使用12.00MHz晶振->定时标周期为5ms ---
*/
```

```

--- 主节点和从节点使用相同的定时周期 ---
--- 假定主节点和从节点都使用 1232 看门狗 ---
-----*/
#include "Main.h"
#include "Port.h"
#include "SCU_Bm.H"
#include "Delay_T0.h"
#include "TLight_B.h"
//-----公有变量定义-----
tByte Tick_message_data_G[NUMBER_OF_SLAVES] = {'M'};
tByte Ack_message_data_G[NUMBER_OF_SLAVES];
// -----公有变量声明-----
// 任务数组 (参见 Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量数组 (参见 Sch51.c)
extern tByte Error_code_G;
// -----私有变量定义-----
static tByte Current_slave_index_G = 0;
static bit First_ack_G = 1;
static bit WATCHDOG_state_G = 0;
// -----私有函数原型-----
static void SCU_B_MASTER_Reset_the_Network(void);
static void SCU_B_MASTER_Shut_Down_the_Network(void);
static void SCU_B_MASTER_Switch_To_Backup_Slave(const tByte);
static void SCU_B_MASTER_Send_Tick_Message(const tByte);
static bit SCU_B_MASTER_Process_Ack(const tByte);
static void SCU_B_MASTER_Watchdog_Init(void);
static void SCU_B_MASTER_Watchdog_Refresh(void) reentrant;
// -----私有常数-----
// 从节点 ID 可以是任何非零 tByte 值 (必须互不相同)
// 注意: ID 0x00 是用于错误处理的
static const tByte MAIN_SLAVE_IDS[NUMBER_OF_SLAVES] = {0x31};
static const tByte BACKUP_SLAVE_IDS[NUMBER_OF_SLAVES] = {0x31};
#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)
// 可调整 (如果需要的话) 以增加网络复位的间隔时间
//(当网络发生错误时)
#define SLAVE_RESET_INTERVAL 0U
// -----私有变量 -----
static tWord Slave_reset_attempts_G[NUMBER_OF_SLAVES];
// 参见上面的 MAIN_SLAVE_IDS[]
static tByte Current_Slave_IDS_G[NUMBER_OF_SLAVES] = {0};
static bit Message_byte_G = 1;
/*-----*
SCU_B_MASTER_Init_T1_T2()
调度器初始化函数。准备调度器数据结构并设置定时器中断频率。
使用调度器前必须调用此函数。
BAUD_RATE - 所需的波特率。
-----*/

```

```

void SCU_B_MASTER_Init_T1_T2(const tWord BAUD_RATE)
{
    tByte Task_index;
    tByte Slave_index;
    // 尚无中断
    EA = 0;
    // 启动看门狗
    SCU_B_MASTER_Watchdog_Init();
    Network_error_pin = NO_NETWORK_ERROR;
    // 配置RS-485收发器
    RS485_Rx_NOT_Enable = 0; // 主节点的接受始终是使能的
    RS485_Tx_Enable = 1; // 主节点的发送始终是使能的
// -----设置调度器-----
// 清理所有任务
for (Task_index = 0; Task_index < SCH_MAX_TASKS; Task_index++)
{
    SCH_Delete_Task(Task_index);
}
// 清除全局错误变量
// - SCH_Delete_Task()将生成一错误代码。
// (因为任务数组是空的)
Error_code_G = 0;
// 对从节点，ID可任意取值
// -但是ID0x00必须保留
for (Slave_index = 0; Slave_index < NUMBER_OF_SLAVES; Slave_index++)
{
    Slave_reset_attempts_G[Slave_index] = 0;
    Current_Slave_IDs_G[Slave_index] = MAIN_SLAVE_IDS[Slave_index];
}
// -----设置波特率(开始)-----
PCON &= 0x7F; // 设置SMOD位为0(不加倍波特率)
// 使能接收
// *9位数据*,1位起始位,1位停止位,波特率可变(异步)
SCON = 0xD2;
TMOD |= 0x20; // T1设为模式2,8位自动重新加载
TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
    / ((tLong)BAUD_RATE * OSC_PER_INST * 1000)));
TL1 = TH1;
TR1 = 1; // 启动定时器
TI = 1; // 发送第一个字符(哑字符)
// -----设置波特率(结束)-----
// 串口中断没有使能
ES = 0;
// -----建立串联的连接(结束)-----
// -----设置定时器2(开始)-----
// 晶振频率为12MHz,每加1振荡12次
// 
// 定时器2的分辨率是0.000001s(1微秒)
// 所需的定时器2溢出时间是0.005s(5ms)

```

```

// — 需要定时器走 5000 次
// 重装的值为 65536 - 5000= 60536 (十进制) = 0xEC78
T2CON = 0x04; // 装入定时器 2 控制寄存器
T2MOD = 0x00; // 装入定时器 2 方式寄存器
TH2 = 0xEC; // 装入定时器 2 高字节
RCAP2H = 0xEC; // 装入定时器重装捕捉寄存器, 高字节
TL2 = 0x78; // 装入定时器 2 低字节
RCAP2L = 0x78; // 装入定时器重装捕捉寄存器, 低字节
ET2 = 1; // 使能定时器 2 中断
TR2 = 1; // 启动定时器 2
// -----设置定时器 2 (结束) -----
}

/*
SCU_B_MASTER_Start()
使能中断, 启动调度器。
注意: 通常在所有的常规任务加入后调用, 以保持任务同步。
注意: 只能使能调度器中断!!!
注意: 假设波特率至少为 9600, 延迟 2ms。
如果选择了更低的波特率, 需要调整代码。
*/
void SCU_B_MASTER_Start(void)
{
    tByte Slave_ID;
    tByte Num_active_slaves;
    tByte Id, i;
    bit First_byte = 0;
    bit Slave_replied_correctly;
    tByte Slave_index;
    // 刷新看门狗
    SCU_B_MASTER_Watchdog_Refresh();
    // 将系统置于安全状态
    SCU_B_MASTER_Enter_Safe_State();
    // 等待启动时报告错误
    Network_error_pin = NETWORK_ERROR;
    Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
    SCH_Report_Status(); // 调度器尚未运行——人工处理
    // 暂停 3000ms, 以便所有的从节点超时
    // (这么做是为了同步整个网络)
    for (i = 0; i < 100; i++)
    {
        Hardware_Delay_T0(30);
        SCU_B_MASTER_Watchdog_Refresh();
    }
    // 此时和所有的从节点的连接断开
    Num_active_slaves = 0;
    // 在初始的长延迟之后, 所有运行的从节点将已经超时
    // 所有的工作从节点现在将处于准备启动状态
    // 发送一个单字节消息启动这些从节点
    // 连着做两遍, 带命令位
}

```

```
// 在正常情况下，这不可能发生
Slave_index = 0;
do {
    // 刷新看门狗
    SCU_B_MASTER_Watchdog_Refresh();
    // 清除第一字节标志
    First_byte = 0;
    // 找到该从节点的从节点ID
    Slave_ID = (tByte) Current_Slave_IDS_G[Slave_index];
    // 发送一个从节点ID消息
    TI = 0;
    TB8 = 1; // 置命令位
    SBUF = Slave_ID;
    // 等待从节点回答
    Hardware_Delay_T0(5);
    // 检查是否有了回答
    if (RI == 1)
    {
        // 取回答数据
        Id = (tByte) SBUF;
        RI = 0;
        // 核对回答——带命令位
        if ((Id == Slave_ID) && (RB8 == 1))
        {
            First_byte = 1;
        }
    }
    // 发送第二个字节
    TI = 0;
    TB8 = 1;
    SBUF = Slave_ID;
    // 等待从节点回答
    Hardware_Delay_T0(5);
    // 检查是否有了回答
    Slave_replied_correctly = 0;
    if (RI != 0)
    {
        // 取回答数据
        Id = (tByte) SBUF;
        RI = 0;
        if ((Id == Slave_ID) && (RB8 == 1) && (First_byte == 1))
        {
            Slave_replied_correctly = 1;
        }
    }
    if (Slave_replied_correctly)
    {
        // 从节点应答正确
        Num_active_slaves++;
    }
}
```

```
Slave_index++;
}
else
{
// 从节点没有正确应答
// -设法切换到备用节点（如果有的话）

if (Current_Slave_IDs_G[Slave_index] != BACKUP_SLAVE_IDS[Slave_index])
{
// 有后备节点可用：切换到后备节点，再试一次
Current_Slave_IDs_G[Slave_index]
= BACKUP_SLAVE_IDS[Slave_index];
}
else
{
// 没有可用的后备节点（或者后备节点也失效了）——不得不继续
// 注意：这里没有退出循环，以保留灵活性
// 以下是错误处理
Slave_index++;
}
}

} while (Slave_index < NUMBER_OF_SLAVES);
// 处理无响应的从节点...
if (Num_active_slaves < NUMBER_OF_SLAVES)
{
// 用户自定义的错误处理...
// 一个或更多的从节点没有应答
// 注意：在某些情况下，如果从节点无响应，可能需要退出
// -或者重新配置网络。
// 这里的错误处理方法是显示出错并关闭网络
Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
SCU_B_MASTER_Shut_Down_the_Network();
}
else
{
Error_code_G = 0;
Network_error_pin = NO_NETWORK_ERROR;
}
// 准备好发送第一个定时消息
Message_byte_G = 1;
First_ack_G = 1;
Current_slave_index_G = 0;
// 启动调度器
EA = 1;
}

/*-----*
SCU_B_MASTER_Update_T2
这是调度器中断服务程序。该程序的调用频率由 SCU_B_MASTER_Init_T2() 函数中的定时器
```

设定决定。本版本是由定时器 2 的中断触发：定时器将自动重新加载

```
-----*/
void SCU_B_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Task_index;
    tByte Previous_slave_index;

    TF2 = 0; // 必须人工清除
    // 刷新看门狗
    SCU_B_MASTER_Watchdog_Refresh();
    // 默认
    Network_error_pin = NO_NETWORK_ERROR;
    // 用于记录当前从节点
    // 第一个值是 0
    Previous_slave_index = Current_slave_index_G;
    // 假设发送和接收 2 字节的消息
    // -每个消息需要两个定时时标传递
    if (Message_byte_G == 0)
    {
        Message_byte_G = 1;
    }
    else
    {
        Message_byte_G = 0;

        if (++Current_slave_index_G >= NUMBER_OF_SLAVES)
        {
            Current_slave_index_G = 0;
        }
    }
    // 检查是否是正确的从节点应答了上述的消息：
    // (如果是的，存储该从节点发送的数据)
    if (SCU_B_MASTER_Process_Ack(Previous_slave_index) == RETURN_ERROR)
    {
        Network_error_pin = NETWORK_ERROR;
        Error_code_G = ERROR_SCH_LOST_SLAVE;
        // 如果和一个从节点失去联系，复位网络后尝试切换到备用节点（如果有的话）
        //
        // 注意：并不在每个定时这样做，否则网络将被不断地复位
        //
        // 选择 SLAVE_RESET_INTERVAL 的值，每 5s 复位一次
        if (++Slave_reset_attempts_G[Previous_slave_index] >=
SLAVE_RESET_INTERVAL)
        {
            // Now reset the network
            SCU_B_MASTER_Reset_the_Network();
        }
    }
    else
```

```

    {
        // 复位计数器
        Slave_Reset_Attempts_G[Previous_Slave_Index] = 0;
    }
    // 发送定时时标至所有联络上的从节点
    // (发送一个数据字节给当前从节点)
    SCU_B_MASTER_Send_Tick_Message(Current_Slave_Index_G);
    // 注意：计算的单位是定时时标数，而非毫秒。
    for (Task_Index = 0; Task_Index < SCH_MAX_TASKS; Task_Index++)
    {
        // 检查是否有一个任务
        if (SCH_tasks_G[Task_Index].pTask)
        {
            if (SCH_tasks_G[Task_Index].Delay == 0)
            {
                // 任务将开始运行
                SCH_tasks_G[Task_Index].RunMe += 1; // 运行标志加1
                if (SCH_tasks_G[Task_Index].Period)
                {
                    //
                    // 再次调度周期性任务运行
                    SCH_tasks_G[Task_Index].Delay
                    = SCH_tasks_G[Task_Index].Period;
                }
            }
            else
            {
                // 还没准备好运行，只递减延迟
                SCH_tasks_G[Task_Index].Delay -= 1;
            }
        }
    }
}

/*
SCU_B_MASTER_Send_Tick_Message()
此函数在 USART (通用同步-异步收发器) 网络上发送一个定时消息。
此消息的接收将在从节点上引发一个中断，进而调用调度器的更新函数。
*/
void SCU_B_MASTER_Send_Tick_Message(const tByte SLAVE_INDEX)
{
    tLong Timeout;
    // 找到该从节点的从节点 ID
    tByte Slave_ID = (tByte) Current_Slave_IDS_G[SLAVE_INDEX];
    // 根据下标，每次发送一个字节的数据
    // 如果 Message_byte_G 是 0，发送第一个字节（从节点的 ID）
    if (Message_byte_G == 0)
    {
        Timeout = 0;
        while ((++Timeout) && (TI == 0));
        if (Timeout == 0)

```

```

    {
        // UART 无应答——出错了
        Error_code_G = ERROR_USART_TI;
        return;
    }
    TI = 0;
    TB8 = 1; // 设置命令位
    SBUF = Slave_ID;
}
else
{
    // Message_byte_G 是 1, 发送数据字节
    Timeout = 0;
    while ((++Timeout) && (TI == 0));
    if (Timeout == 0)
    {
        // USART (通用同步收发器) 无应答——出错了
        Error_code_G = ERROR_USART_TI;
        return;
    }
    TI = 0;
    TB8 = 0;
    SBUF = Tick_message_data_G[SLAVE_INDEX];
}
// 发送数据——返网
}

/*-----*
SCU_B_MASTER_Process_Ack()
确认从节点 (SLAVE_ID) 应答了上面发送的消息。如果有应答，从 USART 读取消息数据；  

如果没有，调用相应的错误处理程序。 Slave_index ——从节点的索引。  

返回： RETURN_NORMAL ——收到了确认 (Ack_message_data_G 中的数据)  

      RETURN_ERROR ——没有收到确认 (没有数据)
*-----*/
bit SCU_B_MASTER_Process_Ack(const tByte Slave_index)
{
    tByte Message_contents;
    tByte Slave_ID;
    // 第一次调用时，没有确认定时消息可供检查
    // -返回“OK”即可
    if (First_ack_G)
    {
        First_ack_G = 0;
        return RETURN_NORMAL;
    }
    // 找到该从节点的从节点 ID
    Slave_ID = (tByte) Current_Slave_IDS_G[Slave_index];
    // 数据应该已经在缓冲区中
    if (RI == 0)
    {

```

```

// 从节点没有应答最后的定时消息
// 设置出错 LED
Network_error_pin = NETWORK_ERROR;
return RETURN_ERROR;
}

// 有数据——读取之
Message_contents = (tByte) SBUF;
RI = 0;
// 这是对上次消息的应答
// -反向解释消息字节
if (Message_byte_G == 1)
{
    // 检查命令位是否设置了
    if (RB8 == 1)
    {
        // 检查 ID 是否正确
        if (Slave_ID == (tByte) Message_contents)
        {
            // 所需的确认消息已经收到
            return RETURN_NORMAL;
        }
    }
    // 什么地方出错了...
    // 设置出错 LED
    Network_error_pin = NETWORK_ERROR;
    return RETURN_ERROR;
}
else
{
    // 有可用的数据
    // 从从节点消息中提取数据
    //
    // 注意，假定数据是正确的
    // -对重要的数据可能需要发送两次
    Ack_message_data_G[Slave_index] = Message_contents;
    return RETURN_NORMAL; // 返回从节点数据
}
}

/*-----*
SCU_B_MASTER_Reset_the_Network()
此函数复位（即重新启动）整个网络。
-*-----*/
void SCU_B_MASTER_Reset_the_Network(void)
{
    EA=0;
    While(1); // 看门狗将超时
}

/*-----*
SUB_B_MASTER_Shut_Down_the_Network()

```



```

此函数关闭整个网络。
-----*/
void SCU_B_MASTER_Shut_Down_the_Network(void)
{
    EA = 0; // 禁止中断
    Network_error_pin = NETWORK_ERROR;
    SCH_Report_Status(); // 调度器尚未运行——人工处理
    while(1)
    {
        SCU_B_MASTER_Watchdog_Refresh();
    }
}
/*-----*/
SCU_B_MASTER_Enter_Safe_State()
当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位
(2) 主节点检测不到一个从节点
(3) 网络出错
在这些情况下，应把系统置于安全状态。
-----*/
void SCU_B_MASTER_Enter_Safe_State(void)
{
    // 由用户根据自己的需要编辑
    TRAFFIC_LIGHTS_Display_Safe_Output();
}
/*-----*/
SCU_B_MASTER_Watchdog_Init()
此函数设置看门狗计时器。
如果主节点失败（或者发生了其他的错误），定时消息没有到达，调度器将停止运行。
为了检测这种情况，从节点上有一个硬件看门狗在运转。这个看门狗设置为大约 100ms 溢出，用于把系统至于安全状态。然后从节点将一直等待，直到问题得到解决。
注意：在这些情况下，从节点不会发送确认消息。所以主节点（如果在运行）将察觉到出了问题。
-----*/
void SCU_B_MASTER_Watchdog_Init(void)
{
    // 1232 外部看门狗不需要初始化
    // 其他的硬件看门狗可能需要
    // 按需要编写相应的初始化程序
}
/*-----*/
SCU_B_MASTER_Watchdog_Refresh()
喂外部 1232 看门狗。
超时时间在 60~250ms 间（取决于硬件）
硬件：假定为外部 1232 看门狗
-----*/
void SCU_B_MASTER_Watchdog_Refresh(void) reentrant
{
    // 改变看门狗引脚的状态
    if (WATCHDOG_state_G == 1)

```

```

    {
        WATCHDOG_state_G = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state_G = 1;
        WATCHDOG_pin = 1;
    }
}
/*-----文件结束-----*/

```

源程序清单 27.3 共享时钟（UART）调度器演示系统的部分软件（主节点）

注意，网络支持 Max489 收发器，并使用 RS-485 协议。

从节点软件

```

/*-----*
 * SCU_Bs.c (v1.00)
 *-----*

这是一个 SCU (使用 UART) 调度器 (RS-485) 的 B051/B052 实现方案。
*** SLAVE / BACKUP NODE ***
*** MASTER CHECKS FOR SLAVE ACKNOWLEDGEMENTS ***
*** Includes support for transceiver enables ***
*** Uses 1232 watchdog timer ***
*** Assumes 12 MHz osc (same as Master) ***
*** Both Master and Slave share the same tick rate (5 ms) ***
*** - See Master code for details ***
*-----*/
#include "Main.h"
#include "Port.h"
#include "SCU_Bs.h"
#include "TLight_B.h"
//-----公有变量定义-----
// 数据从主节点发送至从节点
tByte Tick_message_data_G;
// 从该从节点发送至主节点的数据
// -数据可能被主节点转送给另一个从节点
tByte Ack_message_data_G = '1';
// -----公有变量声明-----
// 任务数组 (参见 Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量数组 (参见 Sch51.c)
extern tByte Error_code_G;
// -----私有函数原型-----
static void SCU_B_SLAVE_Enter_Safe_State(void);
static void SCU_B_SLAVE_Send_Ack_Message_To_Master(void);

```

第 27 章 使用 UART (通用异步收发器) 的共享时钟调度器

```
static tByte SCU_B_SLAVE_Process_Tick_Message(void);
static void SCU_B_SLAVE_Watchdog_Init(void);
static void SCU_B_SLAVE_Watchdog_Refresh(void) reentrant;
// -----私有常数 -----
// 每个从节点都必须有一个独一无二的非零 ID
#define SLAVE_ID 0x31
#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)
// -----私有变量 -----
static bit Message_byte_G;
static bit WATCHDOG_state_G = 0;
static tByte Message_ID_G = 0;
/*-----*
SCU_B_SLAVE_Init_T1()
调度器初始化函数。准备调度器数据结构并设置定时器中断频率。
使用调度器前必须调用此函数。
BAUD_RATE - 所需的波特率
-----*/
void SCU_B_SLAVE_Init_T1(const tWord BAUD_RATE)
{
    tByte i;
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 清除全局错误变量
    // - SCH_Delete_Task() 将生成一个错误代码。
    // (因为任务数组是空的)
    Error_code_G = 0;
    // 设置网络出错引脚 (收到定时消息后清除)
    Network_error_pin = NETWORK_ERROR;
    // 配置 RS-485 收发器
    RS485_Rx_NOT_Enable = 0; // 这里接收器是始终使能的 Receiver is (here)
    constantly enabled
    //
    // (注意——是负逻辑!)
    RS485_Tx_Enable = 0;      // 使能从节点发送器 Transmitter (in slave) is enabled
    // 只在有数据需要传送时才使能
    // 准备接收第一个定时消息
    Message_byte_G = 1;
    // -----设置波特率 (开始) -----
    PCON &= 0x7F; // 设置 SMOD 位为 0 (不加倍波特率)
    // 使能接收
    // *9 位数据*, 1 位起始位, 1 位停止位, 波特率可变 (异步)
    SCON = 0xD2;
    TMOD |= 0x20; // T1 设为模式 2, 8 位自动重新加载
    TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
        / ((tLong) BAUD_RATE * OSC_PER_INST * 1000))));
```

第6篇 多处理器系统的时间触发体系结构

```
TL1 = TH1;
TR1 = 1; // 启动定时器
TI = 1; // 发送第一个字符(哑字符)
// -----设置波特率(结束) -----
// 开放中断
// (接收和发送一个字节都将产生一个串行中断)
// 全局中断还没有开放
ES = 1;
//启动看门狗
SCU_B_SLAVE_Watchdog_Init();
}

/*-----*
SCU_B_SLAVE_Start()
使能中断，启动从节点调度器
注意：通常在所有的常规任务加入后调用，以保持任务同步。
注意：只能使能调度器中断!!!
*-----*/
void SCU_B_SLAVE_Start(void)
{
    tByte Command = 0;
    tByte Message_byte;
    tByte Count = 0;
    bit Slave_started = 0;
    // 禁止中断
    EA = 0;
    // 程序会因为以下原因运行到这里：
    // 1. 网络刚刚上电
    // 2. 主节点出错，停止发送定时消息
    // 3. 网络损坏，从节点收不到定时时标
    //
    // 设法使系统处于安全状态
    SCU_B_SLAVE_Enter_Safe_State();
    // 注意：这里中断被禁止
    Count = 0;
    Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
    SCH_Report_Status(); // 调度器尚未运行——人工处理
    // 现在等待主节点的信号(无限期地)
    do {
        // 等待定时消息(字节1)的接收，所有位设置为0
        do {
            SCU_B_SLAVE_Watchdog_Refresh(); // 必须不间断地喂看门狗
        } while (RI == 0);
        Message_byte = (tByte) SBUF;
        RI = 0;
        // 会连续收到两个ID消息
        // (带命令位)
        // 确认这两个消息
        if ((Message_byte == (tByte) SLAVE_ID) && (RB8 == 1))
        {
```

第 27 章 使用 UART (通用异步收发器) 的共享时钟调度器

```
Count++;
// 这个从节点接收到的消息——发送确认
// 必须使能从节点的 RS-485 发送器 (Max489)
RS485_Tx_Enable = 1;
TI = 0;
TB8 = 1; // Set command bit //置命令位
SBUF = (tByte) SLAVE_ID;
// 当数据发送时等待
// (看门狗会捕捉到 UART 的失效)
while (TI == 0);
// 现在清除发送使能引脚
RS485_Tx_Enable = 0;
}
else
{
Count = 0;
}
} while (Count < 2);
// 启动调度器
EA = 1;
}

/*
SCU_B_SLAVE_Update
这是调度器中断服务程序。该程序的调用频率由 SCU_B_SLAVE_Init() 函数中的定时器设定决定。
从节点由 USART(通用同步-异步收发器) 中断触发。
*/
void SCU_B_SLAVE_Update(void) interrupt INTERRUPT_UART_Rx_Tx
{
tByte Index;
if (RI == 1) // Must check this. //必须检查。
{
// 默认
Network_error_pin = NO_NETWORK_ERROR;
// 发送(确认)和接收(定时时标)2字节的消息
// -每个消息需要两个定时时标的时间处理
//
// 用于跟踪当前字节
if (Message_byte_G == 0)
{
Message_byte_G = 1;
}
else
{
Message_byte_G = 0;
}
// 检查定时时标数据——必要时发送确认消息
// 注意：启动消息只在超时后发送。
if (SCU_B_SLAVE_Process_Tick_Message() == SLAVE_ID)
{
```

```

SCU_B_SLAVE_Send_Ack_Message_To_Master();
// 喂看门狗(收到正确的消息后)
// (总线上的噪音等不会使看门狗停止运转)
// 启动消息不会刷新从节点
// -必须定时与每个从节点通信
SCU_B_SLAVE_Watchdog_Refresh();
}

//注意：计算的单位是定时时标数，而非毫秒
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
    // 检查是否有一个任务
    if (SCH_tasks_G[Index].pTask)
    {
        if (SCH_tasks_G[Index].Delay == 0)
        {
            // 任务将开始运行
            SCH_tasks_G[Index].RunMe = 1; // 置运行标志
            if (SCH_tasks_G[Index].Period)
            {
                // 再次调度周期性任务运行
                SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
            }
        }
        else
        {
            // 还没准备好运行：只递减延迟
            SCH_tasks_G[Index].Delay -= 1;
        }
    }
    RI = 0; // 清除 RI 标志
}
else
{
    // 中断服务程序在最后一个字节发送后产生的 TI 标志触发调用
    // 在这里复位 RS485_Tx_Enable 标志
    RS485_Tx_Enable = 0;
    // 清除 TI 标志
    TI = 0;
}
}

/*
SCU_B_SLAVE_Send_Ack_Message_To_Master()
收到定时消息后，从节点必须向主节点发送确认消息。注意：只有发给此从节点的
定时消息才会被确认。
确认消息起两个作用：
[1]使主节点确认从节点仍然有效并且运行良好。
[2]提供向主节点（进而其他从节点）传输数据的途径。
注意：从节点间的直接数据传输是不可能的。
-----*/

```

第 27 章 使用 UART（通用异步收发器）的共享时钟调度器

```
-----*/
void SCU_B_SLAVE_Send_Ack_Message_To_Master(void)
{
    // 使能从节点的 RS-485 发送器 (Max489)
    // 注意：此标志将在更新中断服务程序中清除
    RS485_Tx_Enable = 1;
    // 根据下标，每次发送一个字节的数据
    // 如果 Message_byte_G 为 0，发送第一个字节（从节点 ID）
    if (Message_byte_G == 0)
    {
        TI = 0;
        TB8 = 1; // 设置命令位
        SBUF = SLAVE_ID;
    }
    else
    {
        // Message_byte_G 为 1，发送数据字节
        TI = 0;
        TB8 = 0;
        SBUF = Ack_message_data_G;
    }
    // 发送数据——返回
}
/*-----*/
SCU_B_SLAVE_Process_Tick_Message()
定时消息对共享时钟调度器的运转至关重要：定时消息的接收将调用更新中断服务程序，进而驱动调度器。
定时消息中可能包含数据，这些数据由此函数提取。
-----*/
tByte SCU_B_SLAVE_Process_Tick_Message(void)
{
    tByte Data;
    // 取出数据字节
    if (RI == 0)
    {
        // 没有数据——什么地方出错了...
        // 设置出错标志位
        Network_error_pin = NETWORK_ERROR;
        // 返回从节点 ID 0
        return 0x00;
    }
    // 有可用的数据
    Data = (tByte) SBUF;
    RI = 0; // 清除 RI 标志
    // 如何处理消息取决于该消息是第一个还是第二个字节
    if (Message_byte_G == 0)
    {
        // 这应该是 ID 字节
```

```

Message_ID_G = Data;
if (RB8 == 0)
{
    Message_ID_G = 0; // 应该设置命令位
}
else
{
    // 这应该是数据字节
    // 应该没有设置命令位
    if ((Message_ID_G == SLAVE_ID) && (RB8 == 0))
    {
        Tick_message_data_G = Data;
    }
    else
    {
        // 什么地方出错了 —— 将 Message_ID 设置为 0
        Message_ID_G = 0;
        // 设置出错标志位
        Network_error_pin = NETWORK_ERROR;
    }
}
return Message_ID_G;
}
*/

```

SCU_B_SLAVE_Watchdog_Init()

此函数设置看门狗计时器。

如果主节点失败（或者发生了其他的错误），定时消息没有到达，调度器将停止运行。

为了检测这种情况，从节点上有一个硬件看门狗在运转。这个看门狗设置为大约 100ms 溢出，用于把系统至于安全状态。然后从节点将一直等待，直到问题得到解决。

注意：在这些情况下，从节点不会发送确认消息。所以主节点（如果在运行）将察觉到出了问题。

```

void SCU_B_SLAVE_Watchdog_Init(void)
{
    // 1232 外部看门狗不需要初始化
    // 其他的硬件看门狗可能需要
    //
    // 按需要编写相应的初始化程序
}
/*

```

SCU_B_SLAVE_Watchdog_Refresh()

喂 1232 外部看门狗

超时时间在 60~250ms 间（取决于硬件）。

硬件：假定为外部 1232 看门狗

```

void SCU_B_SLAVE_Watchdog_Refresh(void) reentrant
{

```

```

// 改变看门狗引脚的状态
if (WATCHDOG_state_G == 1)
{
    WATCHDOG_state_G = 0;
    WATCHDOG_pin = 0;
}
else
{
    WATCHDOG_state_G = 1;
    WATCHDOG_pin = 1;
}
}

/*
SCU_B_SLAVE_Enter_Safe_State()
当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位。
(2) 主节点失效，而且没有可用的工作着的后备节点。
(3) 网络出错。
(4) 因为任何其他的原因，定时消息没有按时收到。
在这些情况下，应把系统置于安全状态。
*/
void SCU_B_SLAVE_Enter_Safe_State(void)
{
    // 由用户根据自己的需要编辑
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*
-----文件结束-----
*/

```

源程序清单 27.4 共享时钟 (UART) 调度器演示系统的部分软件 (从节点)

注意，网络支持 Max489 收发器，并使用 RS-485 协议。

进阶阅读

- Axelson, J. (1998) Serial Port Complete, Lakeview Research.
- Goldie, J. (1991) 'Comparing EIA-485 and EIA-422-A line drivers and receivers in multipoint applications', National Semiconductor Application Note 759. [Available from www.national.com]
- Goldie, J. (1996) Ten ways to bulletproof RS-485 interfaces'. National Semiconductor Application Note 1057. [Available from www.national.com]
- Nelson, T. (1995) 'The practical limits of RS-485', National Semiconductor Application Note 979. [Available from www.national.com]
- Sivasothy, S. (1998) 'Transceivers and repeaters meeting the EIA RS-485 interface standard'. National Semiconductor Application Note 409. [Available from www.national.com]

使用 CAN 的共享时钟调度器

引言

在上一章中介绍了基于 RS-232 和 RS-485 通信协议的设计，这样，在第 3 篇中讨论的合作式调度器就可以扩展应用到分布式多处理机环境中。

使用这种设计的一个重要优点是 UART 是基本的片上外围硬件，所有的 8051 系列微控制器都有 UART，而且大多数微控制器也有片上 UART（包括 8 位机、16 位机、32 位机等等）。因而，这种点对点和多处理器分布式（和本地）系统是一种可以广泛应用的经济有效的系统设计方案。

然而，基于 UART 的通信协议在共享时钟环境中有两个重要的缺点：

- ◎ UART 只支持字节通信：每两个定时时标之间，主节点和从节点之间只能传输 0.5 字节。
- ◎ 硬件检测错误的能力依然很有限：大多数的错误检测/纠正必须由软件进行。

20 世纪 70 年代后期，汽车工业开始解决与之非常类似的问题。主要的动机是因为乘用车内的导线绝缘套管越来越大、越来越重，同时也越来越昂贵。汽车工业试图寻找一种便宜的多节点串行总线，将数目众多的各种系统部件连接起来（详情参见 Lawrenz, 1997）。

这种需求导致了控制器区域网的出现 (CAN)。现在，CAN 已应用在大多数主要汽车制造商生产的车辆中，同时，它也被广泛应用于过程控制和其他工业领域。

CAN 的特点可以概述如下：

- ◎ CAN 是基于消息的，每个消息最长为 8 个字节。用于共享时钟调度器时，每个时标间隔中，主节点和从节点间的可传输最多 7 个字节的数据。这对于大多数的应用来说已经足够了。
- ◎ 许多 8051 微控制器带有片上 CAN 控制器，能以最小的系统开销应用 CAN 协议。
- ◎ CAN 控制器有先进的错误检测（和纠正）功能，从而大大减少了软件的负载。
- ◎ CAN 可用于本地系统和分布式系统。

本章将考虑如何使用 CAN 总线建立高效可靠的共享时钟多处理器系统。

共享时钟 CAN 调度器

适用场合

用多个 8051 系列的微控制器开发一个嵌入式系统。

该系统为基于调度器的时间触发体系结构。

问题

两个或更多的 8051 微控制器通过 CAN 协议通信，如何调度其上的任务（和传输数据）？

背景知识

在本节中，将介绍一些相应的背景材料。

什么共享时钟调度器？

一般的背景材料参见第 25 章的共享时钟调度器。

什么是 CAN？

下面将从一些最显著和重要的特色开始介绍 CAN 协议：

- CAN 支持短距离上（40 米）的高速数据传输（1MB/s）和最多 10 千米距离上的低速数据传输（5kB/s）。
- CAN 是基于消息的。每个消息中的数据长度可以在 0 到 8 个字节间变化。这个数据长度对许多嵌入式系统很理想。
- 消息的接收可产生中断。中断在整个消息全部收到时才产生（最多为 8 个字节的数据），而 UART 每收到一个字节就会产生中断。
- CAN 是共享广播总线，所有的消息会发送给各个节点。不过，每个消息都有一标识符用于接收节点过滤消息。这意味着，如果使用完整的 CAN 控制器（见后文），可以保证某个特定节点只应答相关的消息，即具有某一特定 ID（标识符）的消息。这个特性非常强大。例如，在实际应用中，一个从节点可以完全忽略从不同从节点发给主节点的消息。
- CAN 通常使用简单而廉价的双线差分串行总线实现。其他的物理层介质也可以使用，比如光纤（但是相当少见）。
- CAN 总线的最大节点数为 32 个。
- 每个消息可以有一个单独的优先级。这意味着，例如，时标消息可以赋予比确认消息更高的优先级。
- CAN 的容错能力非常强，CAN 控制器内建了强大的检测和处理机制。
- 许多公司都生产带有片上 CAN 控制器的微控制器。例如，带 CAN 控制器的 8051 微

控制器有 Infineon 的 C505C、C515C，Philips 的 8xC592、8xC598 和 Dallas 的 80C390。

上述内容在本章中都将看到，总的说来，CAN 总线为可靠的分布式调度应用提供了很好的基础。

下面将更详细地考察 CAN 的一些重要特性。

CAN 1.0 vs. CAN 2.0

CAN 协议有两个版本：CAN 1.0 和 CAN 2.0。CAN 2.0 兼容 CAN 1.0，大多数新控制器都符合 CAN 2.0 标准。

另外，CAN 2.0 标准有两个部分：A 和 B，对于 CAN 1.0 和 CAN 2.0A，ID 标识符必须是 11 位。对于 CAN 2.0B，ID 可以是 11 位（标准标识符）或者 29 位（扩展标识符）。

下面是基本的兼容性规则：

- CAN 2.0B 主动控制器能发送和接收标准和扩展消息。
- CAN 2.0B 被动控制器能发送和接收标准消息，除此之外的扩展帧将被丢弃。当收到扩展消息时，CAN 2.0B 被动控制器不会产生错误。
- 因为 CAN 1.0 控制器收到扩展帧时会产生总线错误，所以不能用于使用扩展标识符的网络。

基本 CAN 控制器和完整 CAN 控制器

CAN 控制器主要分两类。注意，这种分类不是标准规定的，所以会存在一些变化。正如前面已经提到过的，两者的差异在于完整 CAN 控制器提供接收过滤器，能够让节点忽略与其不相关的信息。

哪些微控制器支持 CAN

到本书写作时为止，以下的 8051 微控制器都提供片上 CAN 支持：

- Dallas 80C390，两个片上 CAN 模块，每个都支持 CAN 2.0B。
- Infineon C505C，支持 CAN 2.0B。
- Infineon C515C，支持 CAN 2.0B。
- Philips 8xC591，支持 CAN 2.0B。
- Philips 8x592，支持 CAN 2.0A。
- Philips 8x598，支持 CAN 2.0A。
- Temic T89C51CC01，支持 CAN 2.0B。

对于没有片上 CAN 硬件支持的微控制器该怎么办？

使用带片上 CAN 支持的微控制器在大多数情况下更容易、更可靠而且更经济。如果不能使用带片上 CAN 支持的微控制器又要使用 CAN，则需要使用外部 CAN 控制器芯片。下面简要地介绍两种选择。

Scott(1995)详细地讨论了如何使用 8051 系列微控制器外接 Intel 82527 CAN 控制器芯片。

Hank and Jöhnk (1997) 详细地讨论了如何在各种微控制器上外接 Philips SJA1000 CAN 控制器。

解决方案

本节讨论了如何设计基于 CAN 的共享时钟调度器。

方法的基础

本书中的所有共享时钟调度器都有一样的底层结构（如图 28.1 所示）。

在网络上，主节点处有一个精确的时钟。这个时钟按第 3 篇中讲述的方式驱动主节点的调度器。在在网络上，主节点处有一个精确的时钟。这个时钟按第 3 篇中讲述的方式驱动主节点的调度器。

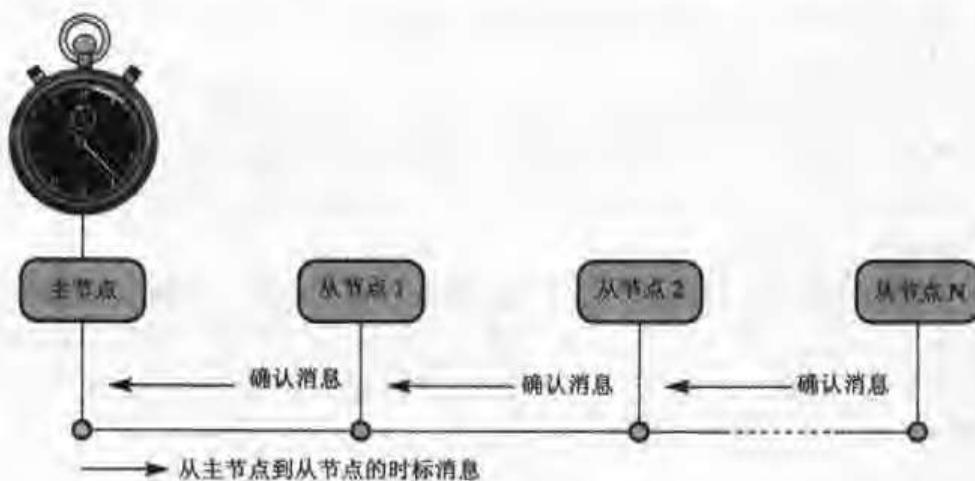


图 28.1 本节中讨论的所有共享时钟调度器的体系结构

从节点也有调度器，而用于驱动这个调度器的中断来源于主节点产生的时标消息。这样，在此基于 CAN 网的模式中，从节点的共享时钟调度器将由接收到主节点数据产生的接收中断驱动。

注意，当从节点的 CAN 模块收到一个数据字节时会产生一个中断，这使得底层调度器能以非常简单的两阶段过程运作：

1. 主节点的定时器溢出使调度器的更新函数被调用，这将发送一个字节的数据到所有的从节点（通过 CAN 总线）：

```
void MASTER_Update_T2(void) interrupt INTERRUPT_Tiraer_2_Overflow {
{
    ...
    MASTER_Send_Tick_Message(...);
    ...
}
```

2. 当所有的从节点收到这个数据时会产生一个中断，这将调用从节点调度器的更新函数，其中一个从节点将向主节点发回一个确认消息（通过 CAN 总线）。

```
void SLAVE_Update(void) interrupt INTERRUPT_CAN {
{
    ...
}
```

```

    SLAVE_Send_Ack_Message_To_Master();
}
...

```

消息结构

正如 SCU 调度器（本地）中所述，基于 UART 网络的消息结构要小心设计，因为数据带宽非常有限（每个定时时标 8 位），而 CAN 总线的一个突出优点是支持最大 8 个字节的消息，这使得消息结构大大简化。

这里采用以下的消息结构设计：

- 在 CAN 网络中最多可以有 31 个从节点（和一个主节点）。每个从节点有一个唯一的标识符（0x01~0xFF）。
- 每个从主节点发来的时标消息长为 1~8 个字节，所有的字节在一个时标间隔内发送。对于所有的消息，第一个字节都是用于寻址的从节点标识符，剩下的字节（如果有）是消息数据——所有的从节点对每个时标消息都会产生中断响应（不论从节点是否是该消息的收件人）。
- 只有当前时标消息寻址的从节点才会回答主节点，答复采取确认消息的形式。
- 每个从节点的确认消息长为 1~8 个字节，所有的字节在收到时标消息的同一个时标间隔内发送。确认消息的第一个字节是发送消息的从节点的标识符，剩下的字节（如果有）是消息数据（如图 28.2 所示）。



图 28.2 基于 CAN 的共享时钟网络中主节点和两个从节点间的通信

确定所需的波特率

主节点定时器的定时应设置得足够长，足以在一个时标间隔内发送时标消息和接收确认消息。显然，定时的长短取决于网络的波特率。

确定 CAN 总线所需的波特率牵涉面更广，因为其消息结构更复杂（如表 28.1 所示）。

每个定时时标传送两个消息，如果时标间隔为 1ms，则至少需要 308 000 波特，若为 350 000 波特则有足够的裕度。对于 CAN 来说，这个波特率下的传输距离大约为 100 米。如果需要更长的传输距离，时标间隔必须延长或者在网络中每隔 100 米加入一个中继节点。

同样需要注意的是，和任何共享时钟网络一样，主节点的定时器和从节点的 CAN 接收中断间存在一个延迟（如图 28.3 所示）。延迟的发生是因为从节点的中断是在时标消息的末尾才产生的。当不存在网络错误时，这个延迟是固定的，而且能由 CAN 总线传送一个消息所需的时间

间大致算出；也就是说，这个延迟只随波特率而变。正如先前讨论过的，350 000 位/秒的波特率对 1ms 的调度器很合适；此时的定时时标延迟（假设消息长 154 位，如表 28.1 所示）大约为 0.5ms。

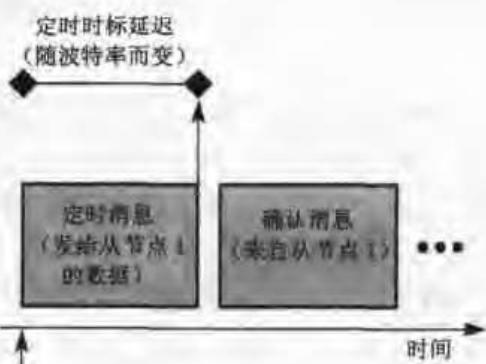


图 28.3 主节点和从节点之间的定时延迟

表 28.1 CAN 消息的结构

描 述	长度 (位)
数据	64
起始位	1
标识符位	11
SRR 位	1
IDE 位	1
标识符位	18
RTR 位	1
控制位	6
循环冗余校验位	15
填充位 (最大)	23
循环冗余校验分隔符	1
确认位	1
确认分隔符	1
EOF 位	7
IFS 位	3
总计	154 位/消息

注意，这是消息最长时的情况，消息包含 8 字节的数据。

如前所述，这个延迟是固定的，而且可以从理论上精确地预计，再通过模拟和测试确认。如果需要在主节点和从节点的进程间精确同步，请注意：

- 所有的从节点应尽可能地保持步调一致。
- 为了让主节点和从节点步调一致，需要在主节点的更新函数中增加一个短延时。

分布式网络的收发器

虽然可用的CAN收发器有几种，但 Philips^①的PCA82C250是使用最广泛的一种。PCA82C250可以很容易地使用类似图28.4所示的电路连接至8051微控制器的片上CAN控制器。

注意，有多种连接方式可供选择，例如，在收发器和微控制器之间加上光耦，以获得更大的隔离程度，避免微控制器的损坏。

有关详情请查阅Philips PCA82C250的数据手册。

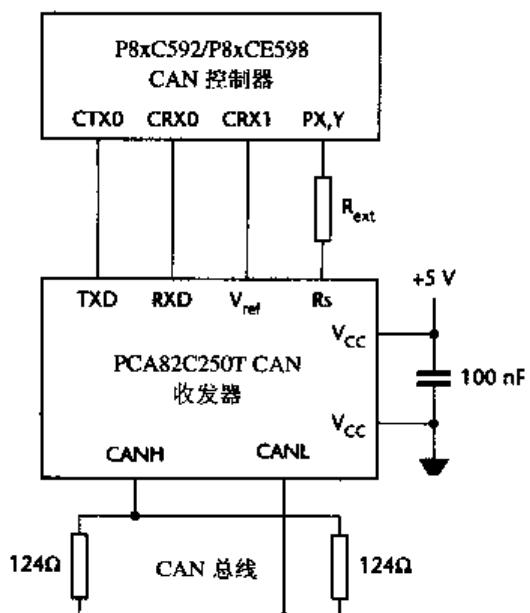


图28.4 将CAN收发器(Philips PCA82C250)连至微控制器的片上CAN控制器
(承蒙Philips Semiconductors允许转载)

分布式网络的节点接线

CAN标准并没有规定必须用某种接线方案。不过，最普遍的连线方式是使用一对双绞线，这和SCU调度器(RS-485)的RS-485接线布置非常类似。

在CAN总线中，两根信号线被称为“CAN高”和“CAN低”。在静态时，两根线的电压都保持在2.5V。传送1时，升高“CAN高”线的电压，这被称为支配位。传送0时，升高“CAN低”线的电压，这被称为隐性位。

因为使用了双绞线，所以CAN的差分输入能够抵消噪音。另外，用这种方式连接的CAN

^① www.philips.com

网络即使在一根线断线时也能够连续工作。

注意，和 RS-485 电缆连接一样，需要在总线的两端并联一个 120Ω 的终端电阻（如图 28.5 所示）。

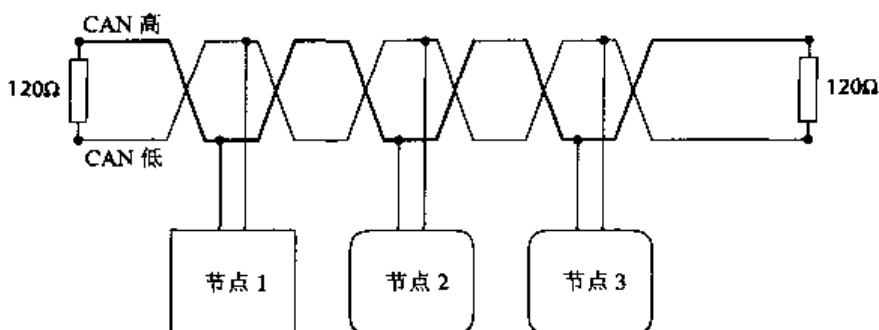


图 28.5 三个节点（带收发器）的 CAN 总线双绞线接线

本地网络的硬件和接线

本地 CAN 网络并不需要使用收发器芯片。对早期的原型 CAN 控制器而言，将各个控制器的 Tx 和 Rx 分别接在一起（即所有的 Tx 线接在一起，所有的 Rx 线接在一起）就能构成网络，并不需要终端电阻。

有一个基于线或结构的更好解决方案（由 Barrenscheen 提出，1996），参见图 28.6。把所有的 Tx 线路经快速二极管连接至数据线（以免输出引脚的短路）。Rx 输入直接连接到这条数据线，数据线由一个上拉电阻拉至 +5V，以产生所需要的被动 1 电平。最大线长限制在 1 米左右。

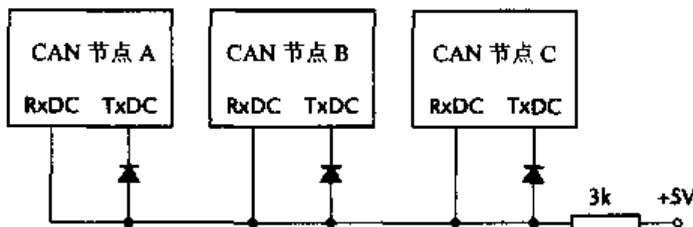


图 28.6 不用收发器连接带 CAN 控制器的微控制器（改编自 Barrenscheen, 1996）

共享时钟 CAN 调度器的软件

共享时钟 CAN 调度器的软件在结构上和前两章的调度器的软件非常相似（实现细节参阅随后的源程序清单）。

这里介绍的调度器和前几章讨论的调度器有一个重要的区别，即错误处理机制。如第 25 章所述，本书共给出了三种不同的共享时钟调度器的错误处理技术。下面将介绍其中的第三种。这种错误处理机制在某个从节点失效时，并不会重启整个网络，而会试着启动相应的备用单元。

这个方法的优点和缺点如下：

- ◎ 可以充分利用后备节点。

◎ 在大多数的情况下，启用备用单元只需花费相对较短的时间。

◎ 所需的编程比本书给出的其他两个方案复杂。

注意，如果网络没能成功地启动备用单元，则可以尝试第 25 章介绍的其他恢复策略。

硬件资源

对于设计 CAN 网络，使用带片上 CAN 支持的微控制器比较经济。注意，一些控制器支持双 CAN 连接。

可靠性和安全性

关于使用多微控制器可能会减少系统可靠性的讨论参见第 434 页。

虽然如此，CAN 的容错能力非常强，CAN 控制器内建了强大的错误检测和处理机制，在随后的例子中将举例说明其中的一些功能。

可移植性

因为只有相对较少的 8051 微控制器具有片上 CAN 支持，所以此模式的移植性比书中介绍的其他一些模式要低。

优缺点小结

- ◎ CAN 是基于消息的，每个消息最长为 8 个字节。用于共享时钟调度器时，每个时标间隔中，主节点和从节点间可传输最多 7 个字节的数据。这对大多数系统而言已经足够了。
- ◎ 许多 8051 微控制器都带有片上 CAN 控制器，能以最小的开销实现 CAN 协议。
- ◎ CAN 控制器有先进的错误检测（和纠正）功能，大大减少了软件开销。
- ◎ CAN 可用于本地系统和分布式系统。
- ◎ 带 CAN 支持的 8051 微控制器一般比标准的 8051 微控制器贵。

相关的模式和替代方案

本书中最相近的替代方案是 SCU 调度器 (RS-485)。

例子：用 Infineon C515C 建立一个基于 CAN 的调度器

这个例子介绍了 Infineon C515C 微控制器的应用，这种流行的微控制器具有片上 CAN 硬件支持（源程序清单 28.1 和源程序清单 28.2）。

配合前面讨论的硬件，这些程序可用于分布式或者本地网络。

主节点软件

```
/*-----  
SCC_M515.c (v1.00)
```

第28章 使用CAN的共享时钟调度器

```
-----  
***这是80C515C的共享时钟(CAN)调度器的主节点程序***  
*** T2用于定时，16位自动重新装入  
***此版本假定515C使用了10MHz晶振 ***  
***时标间隔为6ms  
***主节点和从节点的时标间隔相同***  
-----*/  
#include "Main.h"  
#include "Port.h"  
#include "Delay_T0.h"  
#include "TLight_B.h"  
#include "SCC_M515.h"  
  
// -----公有变量定义-----  
// 发送四个字节的数据(加上标识符)  
tByte Tick_message_data_G[NUMBER_OF_SLAVES][4] = {'M'};  
tByte Ack_message_data_G[NUMBER_OF_SLAVES][4];  
  
// -----公有变量声明-----  
// 任务数组(参见Sch51.c)  
extern sTask SCH_tasks_G[SCH_MAX_TASKS];  
// 错误代码变量数组(参见Sch51.c)(see Sch51.c)  
extern tByte Error_code_G;  
  
// -----私有变量定义-----  
static tByte Slave_index_G = 0;  
static bit First_ack_G = 1;  
  
// -----私有函数原型-----  
static void SCC_A_MASTER_Send_Tick_Message(const tByte);  
static bit SCC_A_MASTER_Process_Ack(const tByte);  
static void SCC_A_MASTER_Shut_Down_the_Network(void);  
static void SCC_A_MASTER_Enter_Safe_State(void);  
static void SCC_A_MASTER_Watchdog_Init(void);  
static void SCC_A_MASTER_Watchdog_Refresh(void) reentrant;  
static tByte SCC_A_MASTER_Start_Slave(const tByte) reentrant;  
  
// -----私有常数-----  
//不使用标识符0x00(已用于启动从节点)  
static const tByte MAIN_SLAVE_IDS[NUMBER_OF_SLAVES] = {0x01};  
static const tByte BACKUP_SLAVE_IDS[NUMBER_OF_SLAVES] = {0x02};  
//复位目标间有数个定时时标  
#define SLAVE_RESET_INTERVAL_OU  
#define NO_NETWORK_ERROR (1)  
#define NETWORK_ERROR (0)  
  
// -----私有变量-----  
static tWord Slave_reset_attempts_G[NUMBER_OF_SLAVES];  
// 从节点ID可以是任何非零tByte值(但必须互不相同)  
static tByte Current_Slave_IDS_G[NUMBER_OF_SLAVES] = {0},
```

```
//对从节点, ID 可任意取值
/*
 *-----*
 * SCC_A_MASTER_Init_T2_CAN()
 * 调度器初始化函数。
 * 准备调度器数据结构并设置定时器中断频率。
 * 使用调度器前必须调用此函数。
 *-----*/
void SCC_A_MASTER_Init_T2_CAN(void)
{
    tByte i;
    tByte Message;
    tByte Slave_index;
    // 尚无中断
    EA = 0;
    // 启动看门狗
    SCC_A_MASTER_Watchdog_Init();
    Network_error_pin = NO_NETWORK_ERROR;
    // -----设置调度器 -----
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // - SCH_Delete_Task()将生成--错误代码
    // (因为任务数组是空的)
    Error_code_G = 0;
    // 对从节点, ID 可任意取值
    for (Slave_index = 0; Slave_index < NUMBER_OF_SLAVES; Slave_index++)
    {
        Slave_reset_attempts_G[Slave_index] = 0;
        Current_Slave_IDs_G[Slave_index] = MAIN_SLAVE_IDS[Slave_index];
    }
    // 准备好发送第一个时标消息
    First_ack_G = 1;
    Slave_index_G = 0;
    //-----设置 CAN (开始) -----
    // -----SYSCON 寄存器 -----
    // 使能 XRAM 和 CAN 控制器读写
    // 读写时无!RD 和 !WR 信号
    // 至 XRAM/CAN 控制器
    // ALE (地址锁存使能) 有效
    SYSCON = 0x20;
    // -----CAN 控制/状态寄存器-----
    // 开始初始化 CAN 模块
    CAN_cr = 0x41; // INIT 和 CCE
    // -----位定时寄存器-----
    // 波特率 = 333.333 kbaud
    // 对 8 字节数据, 1ms 定时需要高于 308kbaud 的波特率
```

```

// 详见正文
//
// 取样点前有 5 个时间段
// 取样点后有 4 个时间段
// 再同步跨越宽度是 2 个时间段
CAN_btr1 = 0x34;      // 位定时寄存器
CAN_btr0 = 0x42;
CAN_gms1 = 0xFF; // 全局掩码短寄存器 1
CAN_gms0 = 0xFF; // 全局掩码短寄存器 0
CAN_ugm11 = 0xFF; // 高全局掩码长寄存器 1
CAN_ugm10 = 0xFF; // 高全局掩码长寄存器 0
CAN_lgm11 = 0xF8; // 低全局掩码长寄存器 1
CAN_lgm10 = 0xFF; // 低全局掩码长寄存器 0
// ---配置定时时标消息对象 ---
// 消息对象 1 有效
CAN_messages[0].MCR1 = 0x55; // 消息控制寄存器 1
CAN_messages[0].MCR0 = 0x95; // 消息控制寄存器 0
// 消息方向是发出
// 扩展 29 位标识符
// 有 0x000000 标识符和 5 字节有效数据
CAN_messages[0].MCFG = 0x5C; // 消息配置寄存器 0
CAN_messages[0].UAR1 = 0x00; // 高位优先级寄存器 1
CAN_messages[0].UAR0 = 0x00; // 高位优先级寄存器 0
CAN_messages[0].LAR1 = 0x00; // 低位优先级寄存器 1
CAN_messages[0].LAR0 = 0x00; // 低位优先级寄存器 0
CAN_messages[0].Data[0] = 0x00; // 数据字节 0
CAN_messages[0].Data[1] = 0x00; // 数据字节 1
CAN_messages[0].Data[2] = 0x00; // 数据字节 2
CAN_messages[0].Data[3] = 0x00; // 数据字节 3
CAN_messages[0].Data[4] = 0x00; // 数据字节 4
// ---配置确认消息对象---
// 消息对象 2 有效
// 注意：对象 2 接收所有的确认消息
CAN_messages[1].MCR1 = 0x55; // 消息控制寄存器 1
CAN_messages[1].MCR0 = 0x95; // 消息控制寄存器 0
// 消息方向是接收
// 扩展 29 位标识符
// 都有标识符：0x000000FF (5 字节有效数据)
CAN_messages[1].MCFG = 0x04; // 消息配置寄存器
CAN_messages[1].UAR1 = 0x00; // 高位优先级寄存器 1
CAN_messages[1].UAR0 = 0x00; // 高位优先级寄存器 0
CAN_messages[1].LAR1 = 0xF8; // 低位优先级寄存器 1
CAN_messages[1].LAR0 = 0x07; // 低位优先级寄存器 0
// 配置其余的消息对象——全部无效
for (Message = 2; Message <= 14; ++Message)
{
    CAN_messages[Message].MCR1 = 0x55; // 消息控制寄存器 1
    CAN_messages[Message].MCR0 = 0x55; // 消息控制寄存器 0
}

```

```
// ----- CAN 控制寄存器 -----
// 复位 CCE 和 INIT
CAN_cr = 0x00;
// -----设置 CAN (结束) -----
// -----设置定时器 2 (开始) -----
// 80c515c, 10 MHz
// 定时器 2 设置为每 6ms 溢出——参见正文
// 方式 1 =定时器功能
// 预分频: Fcpu/12
T2PS = 1;
// 方式 0 =定时器溢出时自动重新加载
// 将定时器寄存器预置为自动重装值
// 注意: 定时和标准 (8052) T2 定时一样
// -如果 T2PS = 1 (否则比 8052 快一倍)
TL2 = 0x78;
TH2 = 0xEC;
// 所有通道均设为方式 0
T2CON |= 0x11;
// 使能定时器 2 中断
ET2 = 1;
// 禁止定时器 2 外部重新加载中断
EXEN2 = 0;
// 比较/捕捉通道 0
// 禁止
// 比较寄存器 CRC 打开: 0x0000
CRCL = 0x78;
CRCH = 0xEC;
// 禁止 CC0/ext3 中断
EX3 = 0;
// 比较/捕捉通道 1~3
// 禁止
CCL1 = 0x00;
CCH1 = 0x00;
CCL2 = 0x00;
CCH2 = 0x00;
CCL3 = 0x00;
CCH3 = 0x00;
// 中断通道 1~通道 3
// 禁止
EX4 = 0;
EX5 = 0;
EX6 = 0;
// 通道 0~通道 3 的上述所有方式
CCEN = 0x00;
// -----设置定时器 2 (结束) -----
}

/*
SCC_A_MASTER_Start()
使能中断, 启动调度器。

```

第 28 章 使用 CAN 的共享时钟调度器

注意：通常在所有的常规任务加入后调用，以保持任务同步。

注意：只能使能调度器中断！！！

```
-----*/  
void SCC_A_MASTER_Start(void)  
{  
    tByte Num_active_slaves;  
    tByte i;  
    bit Slave_replied_correctly;  
    tByte Slave_index, Slave_ID;  
    // 刷新看门狗  
    SCC_A_MASTER_Watchdog_Refresh();  
    // 将系统置于安全状态  
    SCC_A_MASTER_Enter_Safe_State();  
    // 等待启动时报告错误  
    Network_error_pin = NETWORK_ERROR;  
    Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;  
    SCH_Report_Status(); // 调度器尚未运行——人工处理  
    // 暂停 300ms，以便所有的从节点超时  
    // (这么做是为了同步整个网络)  
    for (i = 0; i < 10; i++)  
    {  
        Hardware_Delay_T0(30);  
        SCC_A_MASTER_Watchdog_Refresh();  
    }  
    // 此时断开和所有的从节点的连接  
    Num_active_slaves = 0;  
    // 在初始的长延迟之后，所有运行的从节点将已经超时  
    // 所有的工作从节点现在将处于准备启动状态  
    // 向从节点发送一个从节点标识符消息，以启动从节点  
    Slave_index = 0;  
    do {  
        // 刷新看门狗  
        SCC_A_MASTER_Watchdog_Refresh();  
        // 找到该从节点的从节点 ID  
        Slave_ID = (tByte) Current_Slave_IDS_G[Slave_index];  
        Slave_replied_correctly = SCC_A_MASTER_Start_Slave(Slave_ID);  
        if (Slave_replied_correctly)  
        {  
            Num_active_slaves++;  
            Slave_index++;  
        }  
        else  
        {  
            // 从节点没有正确应答  
            // -设法切换到备用节点（如果有的话）  
            if (Current_Slave_IDS_G[Slave_index] !=  
                BACKUP_SLAVE_IDS[Slave_index])  
            {  
                // 有后备节点可用：切换到后备节点，再试一次  
            }  
        }  
    } while (Slave_index < 10);  
}
```

```

    Current_Slave_IDs_G[Slave_index]
        = BACKUP_SLAVE_IDS[Slave_index];
    }
else
{
// 没有可用的后备节点（或者后备节点也失效了）——不得不继续
    Slave_index++;
}
}

} while (Slave_index < NUMBER_OF_SLAVES);
// 处理无响应的从节点...
if (Num_active_slaves < NUMBER_OF_SLAVES)
{
// 用户自定义的错误处理...
// 一个或更多的从节点没有应答
// 注意：在某些情况下，如果从节点无响应，可能需要退出
// -或者重新配置网络。
// 最简单的处理方法是显示出错并继续
// （此处即如此处理）
Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
Network_error_pin = NETWORK_ERROR;
{
}
else
{
Error_code_G = 0;
Network_error_pin = NO_NETWORK_ERROR;
}
// 启动调度器
IRCON = 0;
EA = 1;
}
/*-----*
SCC_A_MASTER_Update_T2
这是调度器中断服务程序。该程序的调用频率由 SCC_A_MASTER_Init_T2() 函数中的定时器设定决定。本版本是由定时器 2 的中断触发：定时器将自动重新加载。
*-----*/
void SCC_A_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
tByte Index;
tByte Previous_slave_index;
bit Slave_replied_correctly;
TF2 = 0; // 必须清除
// 刷新看门狗
SCC_A_MASTER_Watchdog_Refresh();
// 默认
Network_error_pin = NO_NETWORK_ERROR;
// 用于记录当前从节点
Previous_slave_index = Slave_index_G; // 第一个前一从节点的值是 0...
if (++Slave_index_G >= NUMBER_OF_SLAVES)

```

```
{  
    Slave_index_G = 0;  
}  
// 检查是否是正确的从节点应答了上述的消息:  
// (如果是的, 存储该从节点发送的数据)  
if (SCC_A_MASTER_Process_Ack(Previous_slave_index) == RETURN_ERROR)  
{  
    Error_code_G = ERROR_SCH_LOST_SLAVE;  
    Network_error_pin = NETWORK_ERROR;  
    // 如果和一个从节点失去联系, 尝试切换到备用节点 (如果有的话)  
    if (Current_Slave_IDs_G[Slave_index_G] !=  
        BACKUP_SLAVE_IDS[Slave_index_G])  
    {  
        // 有后备节点可用: 切换到后备节点, 再试一次  
        Current_Slave_IDs_G[Slave_index_G] =  
        BACKUP_SLAVE_IDS[Slave_index_G];  
    }  
    else  
    {  
        // 没有后备节点可用 (或者已在使用中)  
        // 试试工作从节点  
        Current_Slave_IDs_G[Slave_index_G]  
        = MAIN_SLAVE_IDS[Slave_index_G];  
    }  
    // 设法连接从节点  
    Slave_replied_correctly  
    SCC_A_MASTER_Start_Slave(Current_Slave_IDs_G[Slave_index_G]);  
    if (!Slave_replied_correctly)  
    {  
        // 没有可用的后备节点 (或者后备节点也失效了) ——关闭  
        // 对于具体的应用, 其他的处理方法可能更合适  
        SCC_A_MASTER_Shut_Down_the_Network();  
    }  
}  
// 发送定时时标至所有联络上的从节点  
// (发送一个数据字节给当前从节点)  
SCC_A_MASTER_Send_Tick_Message(Slave_index_G);  
// 通过状态寄存器检查CAN总线上次的错误码  
if ((CAN_sr & 0x07) != 0)  
{  
    Error_code_G = ERROR_SCH_CAN_BUS_ERROR;  
    Network_error_pin = NETWORK_ERROR;  
  
    // 错误码的细节参见 Infineon c515c 手册  
  
    CAN_error_pin0 = ((CAN_sr & 0x01) == 0);  
    CAN_error_pin1 = ((CAN_sr & 0x02) == 0);  
    CAN_error_pin2 = ((CAN_sr & 0x04) == 0);  
}
```

```

else
{
    CAN_error_pin0 = 1;
    CAN_error_pin1 = 1;
    CAN_error_pin2 = 1;
}
// 注意：计算的单位是定时时标数，而非毫秒。
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
// 检查是否有一个任务
if (SCH_tasks_G[Index].pTask)
{
    if (SCH_tasks_G[Index].Delay == 0)
    {
// 任务将开始运行
SCH_tasks_G[Index].RunMe += 1; // 运行标志加 1
        if (SCH_tasks_G[Index].Period)
        {
// 再次调度周期性任务运行
            SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
        }
    }
    else
    {
// 还没准备好运行：只递减延迟
        SCH_tasks_G[Index].Delay -= 1;
    }
}
}
/*-----
SCC_A_MASTER_Send_Tick_Message()
此函数在 CAN 网络上发送一个时标消息。
此消息的接收将在从节点上引发一个中断：会调用从节点调度器的更新函数。
-----*/
void SCC_A_MASTER_Send_Tick_Message(const tByte SLAVE_INDEX)
{
// 找到该从节点的 ID
// 所有的从节点都必须有一个独一无二的非零 ID
tByte Slave_ID = (tByte) Current_Slave_IDS_G[SLAVE_INDEX];
CAN_messages[0].Data[0] = Slave_ID;
// 填充数据域
CAN_messages[0].Data[1] = Tick_message_data_G[SLAVE_INDEX][0];
CAN_messages[0].Data[2] = Tick_message_data_G[SLAVE_INDEX][1];
CAN_messages[0].Data[3] = Tick_message_data_G[SLAVE_INDEX][2];
CAN_messages[0].Data[4] = Tick_message_data_G[SLAVE_INDEX][3];
// 在 CAN 总线上发送消息
CAN_messages[0].MCR1 = 0xE7; // TXRQ, 复位 CPUUPD
}

```

```

/*-----*
SCC_A_MASTER_Process_Ack()
确认从节点(SLAVE_ID)应答了前面发送的消息。如果有应答，从USART读取消息数据；如果没有，调用相应的错误处理程序。
参数：从节点的下标。
返回：RETURN_NORMAL -收到了确认(Ack_message_data_G中的数据)
      RETURN_ERROR -没有收到确认(->没有数据)
-----*/
bit SCC_A_MASTER_Process_Ack(const tByte SLAVE_INDEX)
{
    tByte Ack_ID, Slave_ID;
    // 第一次调用时，没有确认时标消息可供检查
    // -返回“OK”即可
    if (First_ack_G)
    {
        First_ack_G = 0;
        return RETURN_NORMAL;
    }
    if ((CAN_messages[1].MCR1 & 0x03) == 0x02) // 如果NEWDAT
    {
        // 收到了一个确认消息
        //
        // 提取数据
        Ack_ID = CAN_messages[1].Data[0]; // 取出数据字节0
        Ack_message_data_G[SLAVE_INDEX][0] = CAN_messages[1].Data[1];
        Ack_message_data_G[SLAVE_INDEX][1] = CAN_messages[1].Data[2];
        Ack_message_data_G[SLAVE_INDEX][2] = CAN_messages[1].Data[3];
        Ack_message_data_G[SLAVE_INDEX][3] = CAN_messages[1].Data[4];
        CAN_messages[1].MCR0 = 0xfd; // 复位NEWDAT和INTPND
        CAN_messages[1].MCR1 = 0xfd;
        // 找到该从节点的ID
        Slave_ID = (tByte) Current_Slave_IDS_G[SLAVE_INDEX];
        if (Ack_ID == Slave_ID)
        {
            return RETURN_NORMAL;
        }
    }
    // 无消息，或者标识符不正确
    return RETURN_ERROR;
}
/*-----*
SCC_A_MASTER_Shut_Down_the_Network()
此函数在一个从节点没能确认时标消息时调用。
-----*/
void SCC_A_MASTER_Shut_Down_the_Network(void)
{
    EA = 0;
    while(1)
    {

```

```

        SCC_A_MASTER_Watchdog_Refresh();
    }
}

/*-----*
 * SCC_A_MASTER_Enter_Safe_State()
 * 当系统发生下列情况时, 进入此状态:
 *   (1) 节点上电或者复位
 *   (2) 主节点检测不到一个从节点
 *   (3) 网络出错
 * 在这些情况下, 应把系统置于安全状态。
 *-----*/
void SCC_A_MASTER_Enter_Safe_State(void)
{
    // 由用户根据自己的需要编辑
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*-----*
 * SCC_A_MASTER_Watchdog_Init()
 * 此函数设置看门狗计时器
 *-----*/
void SCC_A_MASTER_Watchdog_Init(void)
{
    // 看门狗计时器预分频使能 (1/16)
    // 看门狗计时器重新加载值是 0x6B
    // 振荡器是 10 MHz ->看门狗周期是 103ms 左右
    WDTREL = 0xEB;
    // 启动看门狗计时器
    WDT = 1;
    SWDT = 1;
}

/*-----*
 * SCC_A_MASTER_Watchdog_Refesh()
 * 喂 c515c 的内部看门狗。
 *-----*/
void SCC_A_MASTER_Watchdog_Refesh(void) reentrant
{
    WDT = 1;
    SWDT = 1;
}

/*-----*
 * SCC_A_MASTER_Start_Slave()
 * 设法连接从节点。
 *-----*/
tByte SCC_A_MASTER_Start_Slave(const tByte SLAVE_ID) reentrant
{
    tByte Slave_replied_correctly = 0;
    // tByte Slave_ID;
    tByte Ack_ID, Ack_00;
    // 发送一个从节点 ID 消息
}

```

```

CAN_messages[0].Data[0] = 0x00; // 不是一个有效的从节点标识符
CAN_messages[0].Data[1] = SLAVE_ID;
CAN_messages[0].MCR1 = 0xE7; // 发送
// 等待从节点回答
Hardware_Delay_T0(5);
// 检查是否有了回答
if ((CAN_messages[1].MCR1 & 0x03) == 0x02) // 如果 NEWDAT
{
    // 收到了一个确认消息——提取数据
    Ack_00 = (tByte) CAN_messages[1].Data[0]; // 取出数据字节 0
    Ack_ID = (tByte) CAN_messages[1].Data[1]; // 取出数据字节 1
    CAN_messages[1].MCR0 = 0xfd; // 复位 NEWDAT 和 INTPND
    CAN_messages[1].MCR1 = 0xfd;
    if ((Ack_00 == 0x00) && (Ack_ID == SLAVE_ID))
    {
        Slave_replied_correctly = 1;
    }
}
return Slave_replied_correctly;
}
/*-----*
 *-----文件结束-----*
 */

```

源程序清单 28.1 共享时钟 CAN 调度器的部分软件（主节点）

从节点软件

```

/*-----*
 *-----SCC_S515.c (v1.00)
-----*
这是基于 CAN 的 80C515C 共享时钟调度器程序
***主节点和从节点的时标间隔相同***
*** - 详情参见主节点的代码 ***
-----*
#include "Main.h"
#include "Port.h"
#include "SCC_S515.h"
#include "TLight_B.h"
// -----公有变量定义-----
// 数据从主节点发送至从节点
tByte Tick_message_data_G[4];
// 从该从节点发送至主节点的数据
// -数据可能被主节点转送给另一个从节点
tByte Ack_message_data_G[4] = ;
// -----公有变量声明-----
// 任务数组 (参见 Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量数组 (参见 Sch51.c)

```

```

extern tByte Error_code_G;
// -----私有函数原型-----
static void SCC_A_SLAVE_Enter_Safe_State(void);
static void SCC_A_SLAVE_Send_Ack_Message_To_Master(void);
static tByte SCC_A_SLAVE_Process_Tick_Message(void);
static bit SCC_A_SLAVE_Read_Command_Bit(const tByte);
static tByte SCC_A_SLAVE_Set_Command_Bit(const tByte);
static tByte SCC_A_SLAVE_Read_Message_ID(const tByte);
static void SCC_A_SLAVE_Watchdog_Init(void);
static void SCC_A_SLAVE_Watchdog_Refresh(void) reentrant;
// -----私有常数-----
// 每个从节点(和后备节点)都必须有一个独一无二的非零标识符
#define SLAVE_ID 0x01
#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)
/*-----*
 * SCC_A_SLAVE_Init_CAN()
 * 调度器初始化函数。准备调度器数据结构并设置定时器中断频率。
 * 使用调度器前必须调用此函数。
 *-----*/
void SCC_A_SLAVE_Init_CAN(void)
{
    tByte i;
    tByte Message;
    // 清理所有任务
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 清全局错误变量
    // - SCH_Delete_Task()将生成一个错误代码
    // (因为任务数组是空的)
    Error_code_G = 0;
    // 设置网络出错引脚(收到时标消息后清除)
    Network_error_pin = NETWORK_ERROR;
    // ----- SYSCON 寄存器
    // 使能 XRAM 和 CAN 控制器读写。
    // 读写 XRAM/CAN 控制器时!RD 和!WR 信号无效
    // 到 XRAM/CAN 控制器
    // ALE(地址锁存使能)
    SYSCON = 0x20;
    // ----- CAN 控制/状态寄存器-----
    // 开始初始化 CAN 模块
    CAN_cr = 0x41; // INIT 和 CCE
    // -----位定时寄存器-----
    // 波特率 = 333.333 kbaud
    // 对 8 字节数据, 1ms 定时需要高于 308kbaud 的波特率
    // 详见正文
    //
}

```

```

// 取样点前有 5 个时间段
// 取样点后有 4 个时间段
// 再同步跨越宽度是 2 个时间段
CAN_btr1 = 0x34; // 位定时寄存器
CAN_btr0 = 0x42;
CAN_gms1 = 0xFF; // 全局掩码短寄存器 1
CAN_gms0 = 0xFF; // 全局掩码短寄存器 0
CAN_ugml1 = 0xFF; // 高位全局掩码长寄存器 1
CAN_ugml0 = 0xFF; // 高位全局掩码长寄存器 0
CAN_lgml1 = 0xF8; // 低位全局掩码长寄存器 1
CAN_lgml0 = 0xFF; // 低位全局掩码长寄存器 0
// -----配置定时消息对象-----
// 消息对象 1 有效
// 使能接收中断
CAN_messages[0].MCR1 = 0x55; // 消息控制寄存器 1
CAN_messages[0].MCR0 = 0x99; // 消息控制寄存器 0
// 消息方向是接收
// 扩展 29 位标识符
CAN_messages[0].MCFG = 0x04; // 消息配置寄存器
CAN_messages[0].UAR1 = 0x00; // 高位优先级寄存器 1
CAN_messages[0].UAR0 = 0x00; // 高位优先级寄存器 0
CAN_messages[0].LAR1 = 0x00; // 低位优先级寄存器 1
CAN_messages[0].LAR0 = 0x00; // 低位优先级寄存器 0
// -----配置确认消息对象 ---
CAN_messages[1].MCR1 = 0x55; // 消息控制寄存器 1
CAN_messages[1].MCR0 = 0x95; // 消息控制寄存器 0
// 消息方向是发出
// 扩展 29 位标识符
// 5 字节有效数据
CAN_messages[1].MCFG = 0x5C; // 消息配置寄存器
CAN_messages[1].UAR1 = 0x00; // 高位优先级寄存器 1
CAN_messages[1].UAR0 = 0x00; // 高位优先级寄存器 0
CAN_messages[1].LAR1 = 0xF8; // 低位优先级寄存器 1
CAN_messages[1].LAR0 = 0x07; // 低位优先级寄存器 0
CAN_messages[1].Data[0] = 0x00; // 数据字节 0
CAN_messages[1].Data[1] = 0x00; // 数据字节 1
CAN_messages[1].Data[2] = 0x00; // 数据字节 2
CAN_messages[1].Data[3] = 0x00; // 数据字节 3
CAN_messages[1].Data[4] = 0x00; // 数据字节 4
// -----配置其他的对象-----
// 配置其余消息对象 (2~14) ——全部无效
for (Message = 2; Message <= 14; ++Message)
{
    CAN_messages[Message].MCR1 = 0x55; // Message Ctrl. Reg. 1
    CAN_messages[Message].MCR0 = 0x55; // Message Ctrl. Reg. 0
}
// ----- CAN 控制寄存器-----
// 复位 CCE 和 INIT
// 使能 CAN 模块的中断

```

```

// 使能控制器的 CAN 中断
CAN_cr = 0x02;
IEN2 |= 0x02;
// 启动看门狗
SCC_A_SLAVE_Watchdog_Init();
}

/*
SCC_A_SLAVE_Start()
使能中断，启动从节点调度器
注意：通常在所有的常规任务加入后调用，以保持任务同步。
注意：只能使能调度器中断！！！
*/
void SCC_A_SLAVE_Start(void)
{
    tByte Tick_00, Tick_ID;
    bit Start_slave;
    // 禁止中断
    EAL = 0;
    // 程序会因为以下原因运行到这里：
    // 1. 网络刚刚上电
    // 2. 主节点出错，停止发送时标
    // 3. 网络损坏，从节点收不到时标
    //
    // 设法使系统处于安全状态
    // 注意：这里中断被禁止
    SCC_A_SLAVE_Enter_Safe_State();
    Start_slave = 0;
    Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
    SCH_Report_Status(); // 调度器尚未运行——人工处理
    // 现在等待主节点的信号（无限期地）
    do {
        // 等待接收从节点标识符消息
        do {
            SCC_A_SLAVE_Watchdog_Refresh(); // 必须不间断地喂看门狗
        } while ((CAN_messages[0].MCR1 & 0x03) != 0x02);
        // Got a message - extract the data // 收到了一个消息——提取数据
        if ((CAN_messages[0].MCR1 & 0x0c) == 0x08) // if MSGLST set
        {
            // 忽略丢失的消息
            CAN_messages[0].MCR1 = 0xf7; // 复位 MSGLST
        }
        Tick_00 = (tByte) CAN_messages[0].Data[0]; // 取出数据字节 0
        Tick_ID = (tByte) CAN_messages[0].Data[1]; // 取出数据字节 1
        CAN_messages[0].MCR0 = 0xfd; // 复位 NEWDAT 和 INTPND
        CAN_messages[0].MCR1 = 0xfd;
        if ((Tick_00 == 0x00) && (Tick_ID == SLAVE_ID))
        {
            // 消息正确
            Start_slave = 1;
        }
    }
}

```

```
// 发送确认
CAN_messages[1].Data[0] = 0x00;      // 设置数据字节0
CAN_messages[1].Data[1] = SLAVE_ID; // 设置数据字节1
CAN_messages[1].MCR1 = 0xE7; // 发送
}
else
{
// 还没有收到正确的消息——等待
Start_slave = 0;
}
} while (!Start_slave);
// 启动调度器
IRCON = 0;
EAL = 1;
}

/*-----*
SCC_A_SLAVE_Update
这是调度器中断服务程序。该程序的调用频率由 SCC_A_SLAVE_Init() 函数中的定时器设定决定。
从节点由 USART(通用同步-异步收发器) 中断触发。
-----*/
void SCC_A_SLAVE_Update(void) interrupt INTERRUPT_CAN_c515c
{
tByte Index;
// 当收到定时时标时复位
Network_error_pin = NO_NETWORK_ERROR;
// 检查定时时标数据——必要时发送确认消息
// 注意：只在超时后发送启动消息
if (SCC_A_SLAVE_Process_Tick_Message() == SLAVE_ID)
{
    SCC_A_SLAVE_Send_Ack_Message_To_Master();
// 喂看门狗(收到正确的消息后)
// (总线上的噪音等不会使看门狗停止运转)
//
// 启动消息不会刷新从节点
// -必须定时与每个从节点通信
    SCC_A_SLAVE_Watchdog_Refresh();
}
// 通过状态寄存器检查 CAN 总线上次的错误码
if ((CAN_sr & 0x07) != 0)
{
    Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
    Network_error_pin = NETWORK_ERROR;

// 错误码的细节参见 Infineon c515c 手册
    CAN_error_pin0 = ((CAN_sr & 0x01) == 0);
    CAN_error_pin1 = ((CAN_sr & 0x02) == 0);
    CAN_error_pin2 = ((CAN_sr & 0x04) == 0);
}
else
```

```

{
CAN_error_pin0 = 1;
CAN_error_pin1 = 1;
CAN_error_pin2 = 1;
}
// 注意：计算的单位是定时时标数，而非毫秒
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
// 检查是否有任务
if (SCH_tasks_G[Index].pTask)
{
if (SCH_tasks_G[Index].Delay == 0)
{
// 任务将开始运行
SCH_tasks_G[Index].RunMe = 1; // 设置运行标志
if (SCH_tasks_G[Index].Period)
{
//
// 再次调度周期性任务运行
SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
}
}
else
{
// 还没准备好运行：只递减延迟
SCH_tasks_G[Index].Delay -= 1;
}
}
}
}

/*-----*
SCC_A_SLAVE_Process_Tick_Message()
时标消息对共享时钟调度器的运转至关重要：时标消息的接收将调用更新中断服务程序，进而驱动调度器。
时标消息中可能包含数据。这些数据由此函数提取。
*-----*/
tByte SCC_A_SLAVE_Process_Tick_Message(void)
{
tByte Tick_ID;
if ((CAN_messages[0].MCR1 & 0x0c) == 0x08) // 如果设置了MSGLST
{
// 表示当 NEWDAT 仍然置 1 时，CAN 控制器已经在此对象中存储了一个新的消息，即前面存储的消息丢失了
// 这里简单地忽略这种情况，并清除标志
CAN_messages[0].MCR1 = 0xf7; // 复位 MSGLST
}
// 第一个字节是将接收数据的从节点标识符
// intended
Tick_ID = CAN_messages[0].Data[0]; // 取出数据字节 0（从节点标识符）
if (Tick_ID == SLAVE_ID)

```

```

{
    // 只有匹配时才需要复制这些域
    Tick_message_data_G[0] = CAN_messages[0].Data[1];
    Tick_message_data_G[1] = CAN_messages[0].Data[2];
    Tick_message_data_G[2] = CAN_messages[0].Data[3];
    Tick_message_data_G[3] = CAN_messages[0].Data[4];
}
CAN_messages[0].MCR0 = 0xfd; // 复位 NEWDAT 和 INTPND
CAN_messages[0].MCR1 = 0xfd;
return Tick_ID;
}

/*
SCC_A_SLAVE_Send_Ack_Message_To_Master()
收到时标消息后，从节点必须向主节点发送确认消息。注意：只有发给此从节点的时标消息
才会被确认。
确认消息起两个作用：
[1]使主节点确认从节点仍然有效并且运转正常
[2]提供向主节点（进而从节点）传输数据的途径
注意：在从节点间传输数据是不允许的！
*/
void SCC_A_SLAVE_Send_Ack_Message_To_Master(void)
{
    // 消息的第一个字节必须是从节点标识符
    CAN_messages[1].Data[0] = SLAVE_ID; // 数据字节 0
    CAN_messages[1].Data[1] = Ack_message_data_G[0];
    CAN_messages[1].Data[2] = Ack_message_data_G[1];
    CAN_messages[1].Data[3] = Ack_message_data_G[2];
    CAN_messages[1].Data[4] = Ack_message_data_G[3];
    // 在 CAN 总线上发送消息
    CAN_messages[1].MCR1 = 0xE7; // TXRQ, 复位 CPUUPD
}
/*
SCC_A_SLAVE_Watchdog_Init()
此函数设置看门狗计时器
如果主节点失败（或者发生了其他的错误），时标消息没有到达，调度器将停止运行。
为了检测这种情况，从节点上有一个硬件看门狗在运转。这个看门狗设置为大约 100ms 溢出，
溢出后将把系统至于安全状态。然后从节点将一直等待，直到问题得到解决。
注意：在这些情况下，从节点不会发送确认消息。所以主节点（如果在运行）将察觉到出了问题
*/
void SCC_A_SLAVE_Watchdog_Init(void)
{
    // 看门狗计时器预分频使能 (1/16)
    // 看门狗计时器重新加载值是 0x6B
    // 看门狗周期是 103.2ms (10.0MHz 晶振, c515c)
    WDTREL = 0xEB;
    // 启动看门狗计时器
    WDT = 1;
    SWDT = 1;
}

```

```

/*-----*/
SCC_A_SLAVE_Watchdog_Refresh()
// 喂看门狗
/*-----*/
void SCC_A_SLAVE_Watchdog_Refresh(void) reentrant
{
    WDT = 1;
    SWDT = 1;
}
/*-----*/
SCC_A_SLAVE_Enter_Safe_State()
当系统发生下列情况时，进入此状态：
(1) 节点上电或者复位
(2) 主节点失效，而且没有可用的工作着的后备节点
(3) 网络出错
(4) 因为任何其他的原因，时标消息没有按时收到
在这些情况下，应把系统置于安全状态。
/*-----*/
void SCC_A_SLAVE_Enter_Safe_State(void)
{
    // 根据具体情况编写
    TRAFFIC_LIGHTS_Display_Safe_Output();
}
/*-----*/
---文件结束---
/*-----*/

```

源程序清单 28.2 共享时钟 CAN 调度器的部分软件（从节点）

后备从节点软件

后备从节点的源程序参见本书所附 CD。

进阶阅读

Barrenscheen, J. (1996) 'On-board communication via CAN without transceiver', Infineon (Siemens) Application Note AP2921. [Available from the Infineon WWW site].

Gergeleit, M. and Streich, H. (1994) 'Implementing a distributed high-resolution real-time clock using the CAN-bus', Proceedings 1st International CAN Conference, Mainz, Germany, September 1994.

Hank, P. and Jöhnk, E. (1997) 'SJA1000 stand-alone CAN controller'. Philips Application Note AN97076. [Available from the Philips WWW site].

Lawrenz, W. (1997) CAN System Engineering, Springer.

Scott, G. (1995) 'Interfacing an MCS 51 microcontroller to an 82527 CAN controller', Intel Application Note AP-724. [Available from the Intel WWW site]

多处理器系统的设计

引言

共享时钟调度器的应用中有许多需要阐明的重要设计问题，本章将讨论其中的一些关键问题。具体而言，将讨论以下几个方面的内容：

- 在数据联合一节中，将讨论如何通过带宽有限的通信信道在主节点和从节点间传输数据（可能是大量的数据）。
- 在长任务一节中，将讨论调度系统中由来已久的一个问题：需要同时处理长而很少调用的任务和短而经常要调用的任务。这里考虑的技术是将一些长任务迁移至从节点，从而使主节点可以解脱出来处理短任务。
- 在多米诺骨牌任务一节中，将讨论在两个或更多网络节点上运行的任务间的分布式处理问题。具体地说，就是考虑任务持续时间近似而且必须依次调用的情况。

数据联合

适用场合

- 用多个 8051 系列的微控制器开发一个嵌入式系统。
- 该系统为基于共享时钟调度器的时间触发体系结构。

问题

如何通过一个带宽有限的通信信道在主节点和从节点间传输数据？

背景知识

假设使用基于 UART 的共享时钟调度器。通信信道为 8 位带宽，也就是说，任何网络报文只能在主节点和从节点间传输 8 位数据——如何才能在这个有限带宽的信道上传输整数或者浮点数据？这正是此处给出的模式想要解决的问题。

解决方案

为了实现数据联合，将使用 C 语言的 union 关键字。在 C 语言中，定义为 union 的变量能够在不同的时候存储不同类型和大小的值，由编译器保证变量的大小和对齐。因此，不同类型的数据可以储存在单个变量中。

定义 union 变量的语法和广泛使用的 struct 一样，但是 struct 变量可以同时存储一个整型变量、一个双精度变量和一个字符变量，而 union 只能存储一个整型变量、一个双精度变量，或一个字符变量。

例如，看一下源程序清单 29.1 中的代码。

```
// 使用 C/C++ union 关键字的一个例子
#include <stdio.h>
// 基于 union 的用户自定义的数据类型
typedef union
{
    int Integer;
    float Float;
} uNumber;
int main(void)
{
    uNumber Value
    Value.Integer = 100;
    printf("%s\n%s\n%d\n%f\n\n",
           "Put a value in the integer member.",
           "and print both members.",
           "int: ", Value.Integer,
           "float: ", Value.Float);
    Value.Float = 100.0f;
    printf("%s\n%s\n%d\n%f\n",
           "Put a value in the floating member",
           "and print both members.",
           "int: ", Value.Integer,
           "float: ", Value.Float);
    return 0;
}
```

源程序清单 29.1 使用 C/C++ union 关键字的一个例子

源程序列表 29.1 中的程序输出见图 29.1。

周密地使用 union 关键字能够高效地在共享时钟网络的节点间传输数据，这将在后面的例子中说明。

硬件资源

并不需要特别的硬件资源。

```
在整型数据成员中放入一个数值，  
并打印两个数据成员。
```

```
int: 100  
float: 0.000000
```

```
在浮点数据成员中放入一个数值，  
并打印两个数据成员。
```

```
int: 0  
float: 100.000000
```

图 29.1 源程序清单 29.1 中的程序输出

可靠性和安全性

如果相互传输数据的两个 8051 微控制器上的程序代码是用相同的编译器编译的，本模式的方法就是简单而安全可靠的。

然而，如果想用这种方法将数据传送到一个不同系列的微控制器或者微处理器，就有可能会出问题，详细情形参见可移植性一节。

可移植性

这种方法是可移植的，能够被用于任何微控制器或者微处理器。

但另一方面，数据本身可能是不可移植的。例如，如果用这种方法在 8 位和 16 位微控制器或者 8 位微控制器和 32 位个人计算机间传输数据，必须小心从事。因为程序代码假定数据（特别是浮点数据）大小一定，并按一定的次序存储，而数据的大小和存储次序对不同的编译器和运行环境是不同的。

例如，浮点数在 8051 上可能表示为 4 个字节，而在其他环境中，却可能是用 8 字节或者 16 字节表示的。如果试图直接在 8 字节环境中解释 4 字节浮点数，结果将是毫无意义的。

因此，在不同的环境之间用这种方法传输数据时必须小心。

优缺点小结

- ◎ 简单而有效地通过单字节通信信道在 8051 微控制器间传输数据的方法。
- ◎ 当用于 8 位和 16 位或者 32 位环境间传输数据时，必须小心操作。

相关的模式和替代方案

数据联合是多级任务的一种形式。

例子：在微控制器间传输浮点数

这里有一个在 8 位通信网络上传输 32 位浮点数的简单例子。

```

// 为了在串行连接或者并口上传输，分解浮点数
// 假定浮点数是4字节
typedef union
{
    float Float;
    unsigned char Bytes[4];
} uTransfer;

void main()
{
    uTransfer X, Y;
    X.Float = 3.1415f;
    printf("Original data is %f\n", X.Float);
    // 模拟通过单字节宽的通信链路传输浮点数
    for (byte = 0; byte < 4; byte++)
    {
        Y.Bytes[byte] = X.Bytes[byte];
    }
    // 现在显示传输数据
    printf("Original" " data is %f\n", X.Float);
}

```

进阶阅读

Schildt, H. (1997) Teach Yourself C, 3rd edn, McGraw-Hill, Maidenhead.

长任务

适用场合

- 用多个8051系列的微控制器开发一个嵌入式系统。
- 该系统为基于共享时钟调度器的时间触发体系结构。

问题

如何在多处理器系统中同时处理长任务（不频繁）和短任务（频繁）？

背景知识

合作式、时间触发的体系结构在开发嵌入式系统方面有许多优点。实际上，本书主张，只要实现没有困难，通常应该尽量使用这种体系结构。但是，对合作式调度器的限制，本书也采取面对实际的态度。特别是承认在一些情况下，可能需要同时运行长而不频繁调度的任务（例如，每1000ms持续运行100ms）和短而频繁调度的任务（每毫秒运行0.1ms）。这两个要求在合作式系统中可能相互矛盾，因为合作式系统要求所有的任务，在任何条件下，总的任务最长

执行时间 (WCET)，即最大任务持续时间必须满足条件：

$$WCET_{task} < \text{时标间隔}$$

在本书前面的章节中，已经考虑了满足频繁任务和长任务要求的各种方法。例如，使用更快的处理器（参见第3章）或者更快的晶振（第4章），这样能够缩短任务执行时间。另外，可以利用超时（参见第15章）、多级任务（参见第16章）或者使用混合式调度器（参见第17章）。

本模式通过多处理器方案解决这个问题。

解决方案

本模式中，讨论基于共享时钟调度器的系统体系结构的设计和运用，其最简单的形式如下（如图29.2和图29.3所示）：

- 频繁运行（例如每毫秒调度一次）的单个短任务（例如 WCET 0.1ms）在主节点上运行——这个任务可以用于检查错误或者处理其他需要快速响应的动作。



图 29.2 长任务体系结构

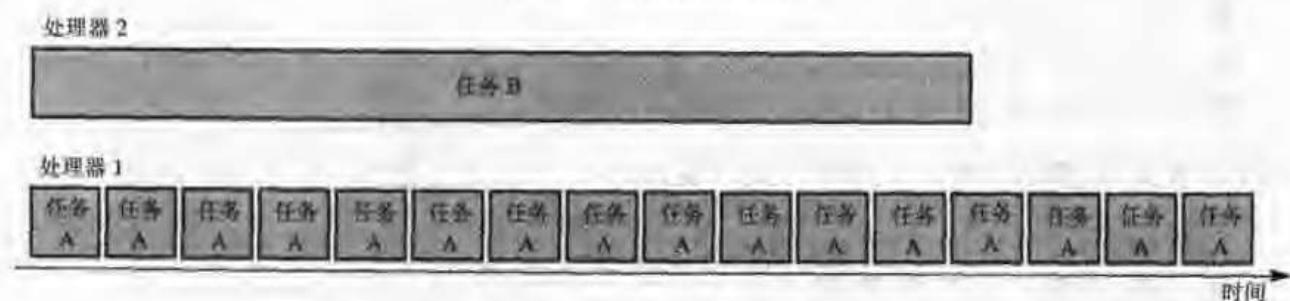


图 29.3 长任务体系结构的任务结构

- 不频繁运行（例如每秒1次）的单个长任务（例如 WCET 500ms）在单个从节点上运行，可用于实现诸如数据处理或者数据压缩算法之类的任务。

- 在大多数情况下，主节点和从节点的时标间隔不同。具体而言，主节点发送定时报文的频率要和长任务的持续时间相配。例如，在本例中，主节点的时标间隔是 1ms 而从节点的时标间隔是 1000ms。这样，主节点每调用 1000 次主节点更新函数时将向从节点发送定时报文。

注意：

- 可能使用多个从节点。
- 所有从节点的时标间隔通常一样（都比较长）。
- 主节点无法直接中止从节点进程。

硬件资源

本模式需要使用至少两个微控制器和相关的硬件（例如，复位、晶振、存储器和电源回路）。

可靠性和安全性

长任务的正确运行有赖于所有相关微控制器的正常运行。这个要求在某些情况下可能会降低系统的总可靠性，参见第 434 页上关于这个问题的讨论。

可移植性

这个模式可以用于任何微控制器或者微处理器家族。

优缺点小结

- ☺ 同时支持长而不频繁调度任务和短而频繁调度任务运行的有效方法。
- ☹ 和任何多处理器设计一样，如果不允许降低整个系统的可靠性，则必须小心设计和使用。

相关的模式和替代方案

- 参见混合式调度器。
- 参见数据联合。
- 参见多米诺骨牌任务。

例子：数据采集和 FFT（快速傅里叶变换）

在许多情况下（如监测系统），需要：

- 通过模数转换器采集数据。
- 在后续处理前，对数据先进行快速傅里叶变换（FFT）。

数据采集进程非常短，一般只要不到一个毫秒就能完成。相比之下，快速傅里叶变换要等到积累了（比方说）256 个采样时才进行。FFT 是计算密集的处理（参见 Press 等, 1992; Lynn 和 Fuerst, 1998; Smith, 1999），通常必须在下一组 256 次采样前完成。

对于这种情况，使用长任务是理想的。

进阶阅读

Lynn, P. and Fuerst, W. (1998) *Introductory Digital Signal Processing with Computer Applications*, Wiley, Chichester.

Press, W.H., Teulolsky, S.A., Vettering, W.T. and Flannery, B.P. (1992) *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, Cambridge.

Smith, S.W. (1999) *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd edn, California Technical Publishing. [Available electronically at www.DSPguide.com]

多米诺骨牌任务

适用场合

- 用多个8051系列的微控制器开发一种嵌入式系统。
- 该系统为基于共享时钟调度器的时间触发体系结构。

问题

当使用多处理器体系结构时，如何最好地调度依次相继运行的任务？

背景知识

表面上看，使用多个微控制器带来的性能改善是明显的。如果一个微控制器的性能是1MIPS而需要达到2MIPS的性能，使用两个处理器就行了。

但实际上，情况却复杂得多。因为要在两个处理器间通信，所以性能不一定会加倍。实际上，在某些情况下，性能还会下降。

例如，假设要顺序执行三个任务（任务A、任务B和任务C），处理从一个数据源处获得的一些数据，如图29.4所示。

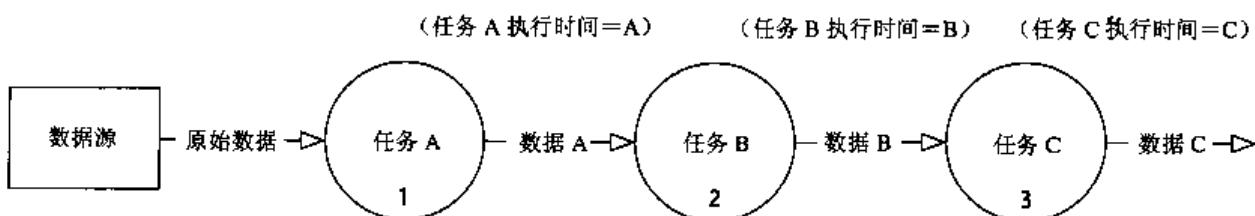


图29.4 三个依次相连任务的执行

如果使用一个处理器执行这些任务，那么执行三个任务并生成数据C所需的时间是：

$$\text{Duration}_1 = A + B + C + \text{Transfer}_{A-B} + \text{Transfer}_{B-C}$$

也就是说，总的时间是任务 A 的执行时间，加上任务 B 和任务 C 的执行时间，加上在任务间传输中间数据所需的时间。

现在，假设用三个不同的微处理器执行一样的任务，每个微处理器的性能指标和第一个例子中的相同。所需的时间为：

$$\text{Duration}_3 = A + B + C + \text{Transfer}_{A,B} + \text{Transfer}_{B,C}$$

在这种情况下，第二个处理器在开始处理前必须等待数据 A，第二个处理器在开始处理前必须等待数据 B，所以，性能没有改善。更糟的是，两个任务在不同的处理器上运行时，数据传输一般需要更长的时间，因而总的处理时间反而会增加。

可是，这不是评定性能的惟一方法。如果不看任务执行时间，而着重于数据重叠，情况将很不一样，见图 29.5。

图 29.5 中，以一个简化形式说明了三任务系统的单处理器版本。这里，在执行任务 B 时，无法同时获取数据。因而，如果任务 A 和任务 B 的执行时间近似，那么大约有一半的时间，数据捕捉不在进行。

和图 29.6 比较一下。



图 29.5 在单个处理器上运行三个依次相连的任务可能导致数据丢失

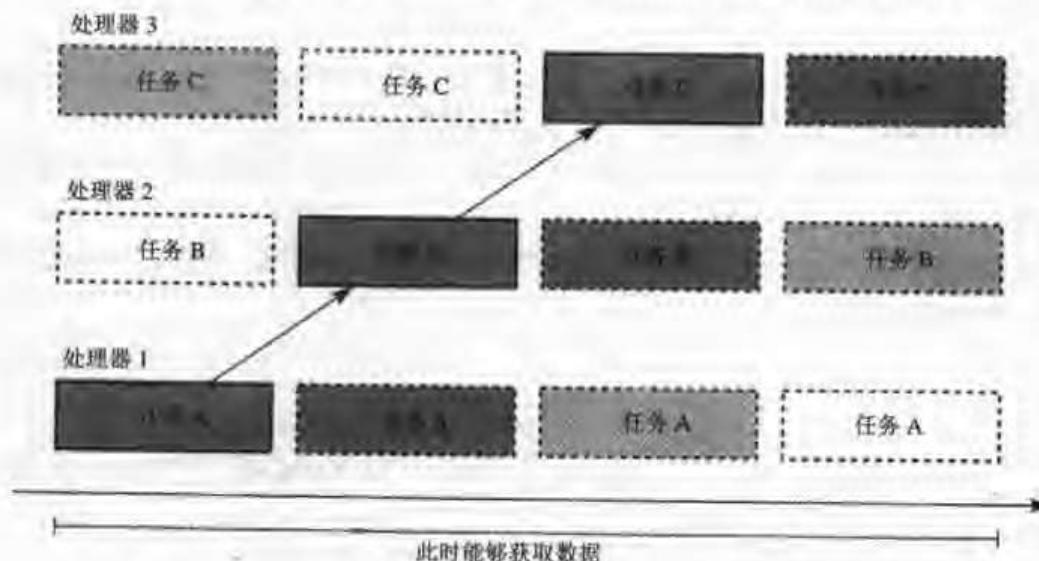


图 29.6 在多处理器体系结构上运行三个依次相连的任务能够避免数据丢失

图 29.6 中，任务 A、任务 B 和任务 C 以多米诺骨牌（或者流水线）方式调度。每个时间间隔，任务 A 处理新数据，而任务 B 处理上次任务 A 运行的处理结果。

在这种情况下，任务 B 和任务 C 被称作多米诺骨牌任务。

解决方案

决定多米诺骨牌任务的体系结构是不是适合某个系统的关键因素是各种任务的最长执行时间。

如图 29.6 所示，为了达到最高效率，任务 A、任务 B 和任务 C 必须具有近似的执行时间。如果这个条件不满足，多米诺骨牌体系结构就不适用，通过使用多处理器获得的改进将大大减少。

如果执行时间长短不同的任务，想要获得使用的多处理器的益处，就需要使用不同的软件体系结构，详情见长任务。

注意，如果需要完全的数据重叠，任务 A 的执行时间必须大于或等于任务 B 的执行时间，而任务 B 的执行时间必须大于或等于任务 C 的执行时间。如果这些条件不满足，则有三种选择：

1. 使用更快的处理器，减少任务 B 及任务 C 的执行时间。
2. 将一个或多个任务劈分成两个连续的短任务。例如，将任务 B 劈分成任务 B1 和任务 B2，并在两个处理器上依次运行每个任务。
3. 将一个或多个任务劈分成两个平行短任务。例如，将任务 B 劈分成任务 B1 和任务 B2，并在两个处理器上依次运行每个任务。

硬件资源

本模式需要使用至少两个微控制器和相关的硬件（例如，复位、晶振，存储器和电源回路）。

可靠性和安全性

多米诺骨牌任务的正确运行有赖于所有相关微控制器的正常运行。这个要求在某些情况下可能会降低系统的总可靠性，参见第 434 页上关于这个问题的讨论。

可移植性

这个模式能用于任何微控制器或者微处理器家族。

优缺点小结

- ☺ 流水线处理任务的有效方法。
- ☹ 和任何多处理器设计一样，如果不允许降低整个系统的可靠性，则必须小心设计和使用。

相关的模式和替代方案

参见混合式调度器。

参见长任务。

参见数据联合。

例子：状态监视和控制

在状态监视与控制中，对第7篇中讨论的那类应用，多米诺骨牌任务的使用特别普遍。

例如，曾讨论过的各种各样的状态监视系统大都采取了图29.7的形式。

这些应用有三个主要阶段：

- 数据采集和预处理阶段，一般包含模数转换器的操作。
- 时域-频域变换阶段（一般包含一个快速傅里叶变换：FFT）。
- 分类阶段，可能会包含一个神经网络。

在所有情况下，每个阶段的执行时间相近，因此多米诺骨牌任务体系结构非常合适。

这类应用的详情参见Li等(1999, 2000)和Parikh等(2001)。

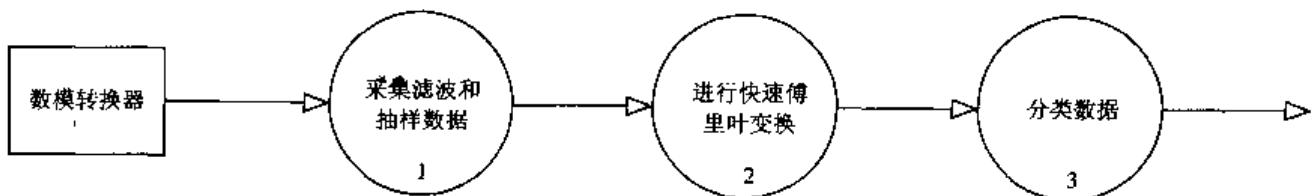


图29.7 一个典型状态监视应用的体系结构

进阶阅读

Li, Y., Pont, M.J., and Jones, N.B. (1999) 'A comparison of the performance of radial basis function and multi-layer Perceptron networks in a practical condition monitoring application'. Proceedings of Condition Monitoring, Swansea, UK, 12-15 April, 1999.

Li, Y., Pont, M.J., Parikh, C.R. and Jones, N.B. (2000) 'Using a combination of RBFN, MLP and kNN classifiers for engine misfire detection', R. John, and R. Birkenhead(eds) Advances in Soft Computing: Soft Computing Techniques and Applications, Springer-Verlag, Berlin.

Parikh, C.R., Pont, M.J., Li, Y. and Jones, N.B. (1999) 'Improving the performance of multi-layer Perceptrons where limited training data are available for some classes', Proceedings of the IEE International Conference on Neural Networks, Edinburgh, September 1999.

Parikh C.R., M.J. Font and N-B. Jones (2001) 'Application of Dempster-Shafer theory in condition monitoring applications – a case study', Pattern Recognition Letters, 22 (6-7): 777-85.

Part 7

监视与控制组件

本篇主要讲述数据采集、状态监视和控制系统的相关模式。在上几篇中，主要讲述的是时间触发软件体系结构的相关技术。

在第 30 章中，将研究脉冲的计数。该技术广泛应用于许多工业领域，特别是汽车工业。硬件脉冲计数模式和软件脉冲计数模式分别介绍了基于硬件和软件的脉冲计数技术。

在第 31 章中，将讲述脉冲频率调制，即，按某个指定的频率产生方波信号。该技术也有两种实现模式：硬件 (PRM) 脉冲频率调制和软件 PRM。

在第 32 章中，将考虑如何用 8051 微控制器测量模拟电压或电流信号。例如，被测量的信号可能是充电器中电池的电压；就一般而言，这些信号来自于类型多样的各种传感器，诸如电位计或者温度探头。单次模数转换 (ADC) 和连续模数转换模式讨论了如何有效地使用片上模拟-数字转换器和独立模拟-数字转换器。本章也讨论了 ADC 前置放大器和滤波器模式，在测量模拟信号前可能需要先经过这些预处理级。最后，电流传感器模式考虑了如何在诸如检测烧坏的灯泡或者堵转直流电动机的系统中测量模拟电流。

在第 33 章中，将探讨如何用微控制器产生脉冲宽度调制信号，从而获得模拟输出，该技术可用于电动机转速控制或者电灯亮度控制等系统。同样，脉宽调制可以用硬件或者软件方法实现，参见硬件脉宽调制模式和软件脉宽调制模式。本章还将考虑脉宽调制信号的滤波后处理，包括脉宽调制平滑滤波器模式和不用专门硬件的高频率 3 级脉宽调制模式。

在第 34 章中，将考虑如何用数模转换器 (DAC) 在微控制器上输出模拟信号（参见数模转换器输出）。同时，也将考虑数模转换器所需的后处理步骤：滤波和放大。所用模式是数模转换器平滑滤波器和数模转换器驱动器。

最后，在第 35 章中，将把注意力转向比例 - 积分 - 微分 (PID) 控制。PID 简单而有效，因此被广泛应用于控制算法。本章将探讨应用于嵌入式时间触发体系的 PID 控制器的设计和实现技术。



Chapter 30

脉冲频率检测

引言

假设需要测量一根旋转轴的转速，这经常会在汽车或者其他工业领域中用到。

测量转速的一个有效方法是在轴上安装一个光电或者磁性旋转编码器（如图 30.1 所示），然后测量在一个给定的周期内（比如说 100ms 或者 1s）的脉冲计数。根据脉冲计数和旋转编码器的参数，能够计算出旋转的平均转速。

本章将考虑如何进行脉冲计数。这一技术不仅能用于转速测量，还能用在流体流速测量和振动感应等方面。

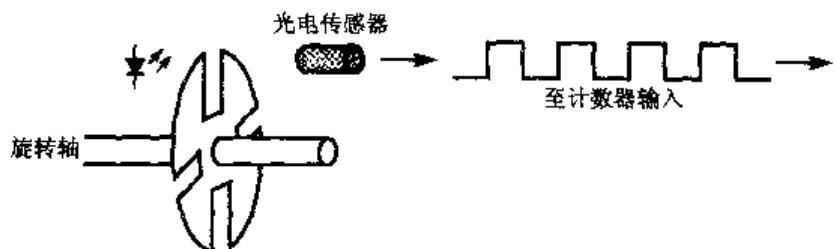


图 30.1 计数光电编码器的脉冲以测量轴的转速

硬件脉冲计数

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何对外围设备送来的脉冲进行计数？

背景知识

解决方案

考虑一下本章引言中讨论过的从旋转编码器处得到的脉冲序列。

如果有足够的硬件资源可用，通常不需要软件的介入就能够从这一脉冲序列算出平均的脉冲频率。由定时器 0 或者定时器 1^①对引脚上的脉冲（准确地说，脉冲的下降沿）进行计数，不会产生中断或影响其他的处理。

硬件脉冲计数的关键是 TMOD 特殊功能寄存器的设置（如表 30.1 所示）。

表 30.1 TMOD 特殊功能寄存器，用于控制定时器 0 和定时器 1

位	7 (MSB)	6	5	4	3	2	1	0 (LSB)
名称	Gate	C/T	M1	M0	Gate	C/T	M1	M0
定时器 1					定时器 0			

在表 30.1 中，TMOD 的大部分位已经在前面讨论过了（参见第 11 章）。下面会用到计数器位（第 6 位和第 2 位）。如果其中任何一位为 0，则对应的定时器就会被设置为计数器方式。

例如：

```
// 定时器 0 用作 16 位定时器。清除所有的 T0 位 (T1 不动) 对 P3.4 (T0 引脚) 的下降沿进行脉冲计数
TMOD &= 0xF0; // 清除所有 T0 位
TMOD |= 0x05; // 置所有需要的 T0 位 (T1 不动)
```

在这种情况下，每当对应的外部输入引脚（如图 30.2 所示）上发生一次 1 到 0 的跃迁，定时器 0 寄存器 (TL0 和 TH0) 加 1。每个机器周期引脚的电平都会被采样。当采样显示一个周期为高，而下一个周期为低，计数就加 1。接着，检测到电平变化的机器周期的随后一个周期中，寄存器将更新为新的计数值。

这种方法对被计数波形没有占空比的限制，但是为了确保测量的准确，高和低电平至少应保持一个完整的机器周期（对 8051 而言是 12 个振荡器周期；对其他较新的 8051 改进型微控制器可能是 6、4 或 1 个振荡器周期）。因而，检测 1 到 0 的电平变化至少需要两个机器周期（24、12、8 或者 2 个振荡器周期）。这决定了能测量的脉冲频率的极限。例如，对一个基本的 12 MHz/12 振荡器周期的 8051，能够测量的最大脉冲频率（方波）是 500kHz，周期为 24 个振荡器周期。

^① 在许多情况下，正如第 3 章中所述，现代 8051 系列微控制器都是基于稍微新一些的 8052 体系结构，包含了新增的更强大的定时器（定时器 2）。定时器 2 还可以用于脉冲计数，可是，正如在第 13 章和第 14 章讨论的，定时器 2 一般被用作调度器的定时。所以这里不讨论利用定时器 2 作脉冲计数。

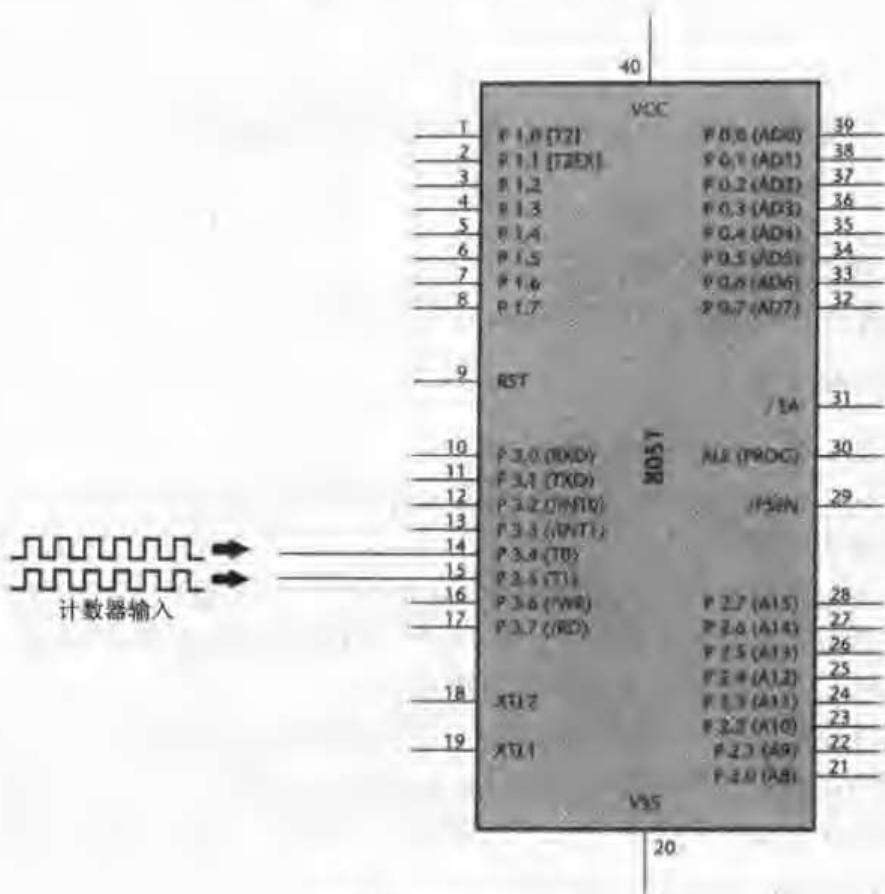


图 30.2 读取脉冲计数

建立一个按以下步骤运行的任务就能有效地测量上述例子中的转速：

1. 读当前的硬件脉冲计数。
2. 将读出的计数存储在一个全局变量中。
3. 将计数复位至 0。

这个任务需要反复调度执行（比方说每 100ms 执行一次）。在下面的例子中，给出了一段代码具体说明这个方法。

硬件资源

硬件脉冲计数需要独占一个定时器，多数情况下定时器 2 被用于调度器，而定时器 1 可能被用作波特率发生器。只剩下定时器 0 可用，如果定时器 0 也被用于其他用途（或者需要测量多个脉冲序列），则可以考虑基于软件的方法，参见软件脉冲计数模式或者增加一个微控制器，利用这个附加的微控制器进行计数（参见第 6 篇）。

可靠性和安全性

如果不打算对超过硬件限制的脉冲进行计数，则没有其他特别的可靠性和安全性要求。

可移植性

因为此模式只使用了 8051 内核特性，所以没有其他的移植限制。

优缺点小结

- ◎ 硬件脉冲计数有最小的软件开销。
- ◎ 每个需要计数的脉冲序列都要独占一个定时器。

相关的模式和替代方案

硬件脉冲计数主要有两个替代方案。

最接近的是软件脉冲计数，测量方法类似，但是不需要使用定时器硬件。然而不可避免的，其软件负荷比本模式大得多。

另外，可以使用外部硬件将脉冲序列转换模拟电压，然后用片上或外部模拟-数字转换器测量。有多种频率电压转换芯片可用于这种方案，例如 National Semiconductor[®] 的 LM2907 和 LM2917。关于 ADC 的使用将在单次模数转换中讨论。

注意，使用模数转换器（和相关的硬件）无法精确计数脉冲的数目，这个方法对于某些系统可能是不适用的，例如计数通过十字转门的来宾数量。但是，如果只要测量平均脉冲计数，比如某种形式的测速，这种解决方案就完全够用了。注意，这个方案的硬件成本可能较高。

例子：通用脉冲计数（硬件）程序库

源程序清单 30.1~源程序清单 30.3 给出了一个简单的脉冲计数（硬件）程序库。

这个程序库对引脚 3.4 上的脉冲进行计数，并将结果以柱状图的形式显示在端口 1 上。

```
/*
Port.H (v1.00)

项目 PC_Hard (参见第 30 章) 的端口头文件 (参见第 10 章)
*/
// ----- Sch51.C -----
// 如果不需要错误报告，注释掉此行。
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P2
#endif
// ----- Bargraph.C -----
// 将 LED 经过合适的电阻从 +5V 连至这些引脚
// [详见第 7 章]
```

```

// 如果需要，8根端口引线可以分布在几个端口上
sbit Pin0 = P1^0;
sbit Pin1 = P1^1;
sbit Pin2 = P1^2;
sbit Pin3 = P1^3;
sbit Pin4 = P1^4;
sbit Pin5 = P1^5;
sbit Pin6 = P1^6;
sbit Pin7 = P1^7;
// ----- PulCnt_H.c -----
// 用定时器0对P3.4上的脉冲进行计数
/*-----文件结束-----*/

```

源程序清单 30.1 通用脉冲计数（硬件）例子的部分代码

```

/*-----*
Main.c (v1.00)
-----*
硬件脉冲计数的演示软件（第30章）。
所需的链接程序选项（详情参见第14章）：
OVERLAY
(main ~ (PC_HARD_Get_Count_T0, BARGRAPH_Update),
SCH_Dispatch_Tasks ! (PC_HARD_Get_Count_T0, BARGRAPH_Update))
*-----*/
#include "Main.h"
#include "2_01_12g.h"
#include "PulCnt_H.h"
#include "BarGraph.h"
/* ..... */
/* ..... */
void main(void)
{
    SCH_Init_T2();           // 设置调度器
    PC_HARD_Init_T0();       // 准备脉冲计数
    BARGRAPH_Init();         // 准备柱状图显示设置
    // 增加一个“脉冲计数”任务
    SCH_Add_Task(PC_HARD_Get_Count_T0, 1000, 1000);
    // 这里简单地显示计数（柱状图显示）
    SCH_Add_Task(BARGRAPH_Update, 1200, 1000);
    // 所有的任务都已加入：启动调度器
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

```

```

    }
/*
-----文件结束-----
*/

```

源程序清单 30.2 通用脉冲计数（硬件）例子的部分代码

```

/*
  PulCnt_H.C (v1.10)

  硬件脉冲计数程序库 (参见第 30 章)
*/
#include "Main.h"
#include "Port.h"
#include "Bargraph.h"
#include "PulCnt_H.h"
// -----公有变量声明 -----
// 计数全局变量——存储最近的计数值
extern tBargraph Count_G;
/*
PC_HARD_Init_T0()
准备用定时器 0 进行硬件脉冲计数
*/
void PC_HARD_Init_T0(void)
{
    // 定时器 0 用作 16 位定时器，在引脚 3.4 (T0 引脚) 的下降沿进行脉冲计数
    TMOD &= 0xF0;           // 清除所有的 T0 位 (T1 不动)
    TMOD |= 0x05;           // 置所有需要的 T0 位 (T1 不动)
    TH0 = 0; TL0 = 0;        // 置定时器计数为 0
    Count_G = 0;             // 置全局计数为 0
    TR0 = 1;                 // 启动计数器
}
/*
PC_HARD_Get_Count_T0()
每隔一定时间，调度此函数
记住：最大计数值是 65536 (16 位计数器) ——必须确保计数不溢出。应选择合适的调度间隔
并留有一定裕度。
对高频率脉冲，需要考虑到在读计数器时计数会停止非常短的时间。
注意：第一个计数前的延迟通常应该和计数的间隔时间相同，以保证第一个计数尽可能的精确。
例如，下面这是正确的：
    Sch_Add_Task(PC_HARD_Get_Count_T0, 1000, 1000);
而这样会得到一个非常小的首次计数：
    Sch_Add_Task(PC_HARD_Get_Count_T0, 10, 1000);
*/
void PC_HARD_Get_Count_T0(void)
{
    TR0 = 0; // 停止计数器
    Count_G = (TH0 << 8) + TL0; // 读取计数
    TH0 = 0; TL0 = 0;           // 复位计数
}
```

```

if (TF0 == 1)
{
    // 定时器已溢出
    // -脉冲频率过高
    // -或者调度频率过低
    // 这个错误的计数设为最大
    // -如果需要的话还可以设置一个全局错误标志
    Count_G = 65535;
    TF0 = 0;
}
TR0 = 1; // 重新启动计数器
}
/*-----*-
-----文件结束 -----
-*-----*/

```

源程序清单 30.3 通用脉冲计数（硬件）例子的部分代码

进阶阅读

软件脉冲计数

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何对外围设备送来的脉冲进行计数？

背景知识

背景知识详见硬件脉冲计数。

解决方案

硬件脉冲计数需要独占一个定时器。

如果没有可用的定时器而且脉冲频率比较低，则可以采用软件脉冲计数方法。

基于软件的脉冲计数用一个周期性的任务跟踪一个或多个输入引脚的电平。如果在任务的两次调用间引脚的状态变化了，相应的计数加 1。这个方法的主要缺点是可测量的最大脉冲频率较低：对于 1ms 的调度器，能够测量的最大脉冲频率是 500Hz，大约比硬件脉冲计数小 1 000 倍。

硬件资源

此模式没有使用片上外围硬件。

可靠性和安全性

此模式没有特别的可靠性和安全性要求。

可移植性

此程序并没有特殊的硬件要求，如果目标环境调度器的定时相同，则可以很容易地移植。

优缺点小结

- ◎ 软件脉冲计数不需要定时器。
- ◎ 主要缺点是可测量的最大脉冲频率较低。对于 1ms 的调度器，能够测量的最大脉冲频率是 500Hz，这比用硬件测量小大约 1000 倍。

相关的模式和替代方案

软件脉冲计数的主要替代方案是硬件脉冲计数。

例子：通用脉冲计数（软件）程序库

源程序清单 30.4~源程序清单 30.6 给出了一个简单的脉冲计数（软件）程序库。

这个程序库对引脚 3.0 上的脉冲进行计数，并将结果以柱状图的形式显示在端口 1 上。

```
/*-----*
Port.H (v1.00)

项目 PC_Soft (参见第 30 章) 的端口头文件 (参见第 10 章)
*-----*/
// ----- Sch51.C -----
// 如果不需要错误报告，注释掉此行。
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P2
#endif
// ----- Bargraph.C -----
// 将 LED 经过合适的电阻从 +5V 连至这些引脚
// [详见第 7 章]
// 如果需要，8 根端口引线可以分布在几个端口上。
sbit Pin0 = P1^0;
sbit Pin1 = P1^1;
sbit Pin2 = P1^2;
sbit Pin3 = P1^3;
```

```

sbit Pin4 = P1^4;
sbit Pin5 = P1^5;
sbit Pin6 = P1^6;
sbit Pin7 = P1^7;
// ----- PulCnt_S.c -----
sbit Count_pin = P3^0;
/*-----文件结束-----*/
*/

```

源程序清单 30.4 通用脉冲计数（软件）例子的部分代码

```

/*-----*
 Main.c (v1.00)
-----*
 软件脉冲计数的演示软件（第30章）。
 所需的链接程序选项（详情参见第14章）：
 OVERLAY
 (main ~ (PC_SOFT_Poll_Count, PC_SOFT_Get_Count, BARGRAPH_Update),
 SCH_Dispatch_Tasks ! (PC_SOFT_Poll_Count, PC_SOFT_Get_Count,
 BARGRAPH_Update))
 *-----*/
#include "Main.h"
#include "2_01_12g.h"
#include "PulCnt_S.h"
#include "BarGraph.h"
/* ..... */
/* ..... */
void main(void)
{
    SCH_Init_T2(); // 设置调度器
    PC_SOFT_Init(); // 脉冲计数设置
    BARGRAPH_Init(); // 柱状图显示(P4)设置
    // 时间单位是时标间隔数(1ms 时标间隔)
    // 增加一个“脉冲计数轮询”任务。
    // 每40ms
    SCH_Add_Task(PC_SOFT_Poll_Count, 0, 20);
    // 增加一个“取脉冲计数”任务。
    // 每20s
    SCH_Add_Task(PC_SOFT_Get_Count, 0, 20000);
    // Simply display the count here (bargraph display)
    // 这里简单地显示计数（柱状图显示）。
    // 最大计数为250
    SCH_Add_Task(BARGRAPH_Update, 50, 10000);
    // 所有的任务已加入：启动调度器
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

```

```

    }
/*-----*
---文件结束 -----
-*-----*/

```

源程序清单 30.5 通用脉冲计数（软件）例子的部分代码

```

/*-----*
PulCnt_S.C (v1.00)

软件脉冲计数程序库（参见第 30 章）。
-*-----*/
#include "Main.h"
#include "Port.h"
#include "Bargraph.h"
#include "PulCnt_S.h"
// -----公有变量定义 -----
// 存储平均计数值
extern tBarGraph Count_G;
//-----私有常数-----
#define PULSE_HIGH (1)
#define PULSE_LOW (0)
// -----私有变量定义 -----
// 存储瞬时计数值
static tWord Count_local_G;
// 上次的脉冲计数引脚状态
static bit Previous_state_G;
/*-----*
PC_SOFT_Init()
软件脉冲计数设置。
-*-----*/
void PC_SOFT_Init(void)
{
    Count_local_G = 0;
    Count_G = 0;
}
/*-----*
PC_SOFT_Poll_Count()
使用软件对指定引脚的下降沿进行计数
-此处没有使用 T0。
-*-----*/
void PC_SOFT_Poll_Count(void)
{
    bit State = Count_pin;
    if ((Previous_state_G == PULSE_HIGH) && (State == PULSE_LOW))
    {
        Count_Local_G++;
    }
}

```

```
Previous_state_G = State;
}

/*
PC_SOFT_Get_Count()
每隔一定时间调度此函数。
复制轮询计数至全局变量。
*/
void PC_SOFT_Get_Count(void)
{
    Count_G = Count_local_G;
    Count_local_G = 0;
}
/*
----文件结束-----
*/

```

源程序清单 30.6 通用脉冲计数（软件）例子的部分代码

进阶阅读

Chapter 31

脉冲频率调制

引言

本章将探讨如何输出各种频率的脉冲序列（有时也叫做时钟输出）。

这种脉冲序列有很多种应用，从驱动简单的闪光报警器（如图 31.1 所示）、控制电灯的明暗和直流电动机的速度，到脉冲频率传感器测试电路等等。



图 31.1 用脉冲频率调制输出控制报警器的频率

硬件脉冲频率调制

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

怎样用最小的软件负荷建立一个频率可变的脉冲频率调制输出（时钟输出）？

背景知识

在本节中，将介绍一些基本背景材料。

占空比

脉冲频率调制就是将一个引脚的电平在一个时段内 (x) 设为逻辑 1, 然后在另一个时段内 (y) 设为逻辑 0, 并重复这一过程 (如图 31.2 所示)。

占空比的定义如下:

$$\text{占空比 (\%)} = \frac{x}{x+y} \times 100$$

本模式中只考虑占空比为 50% 的信号; 也就是说, x 和 y 相等。

在这种情况下, 脉冲频率由下式决定:

$$\text{频率} = \frac{1}{2x} \quad (x \text{ 的单位是秒})$$

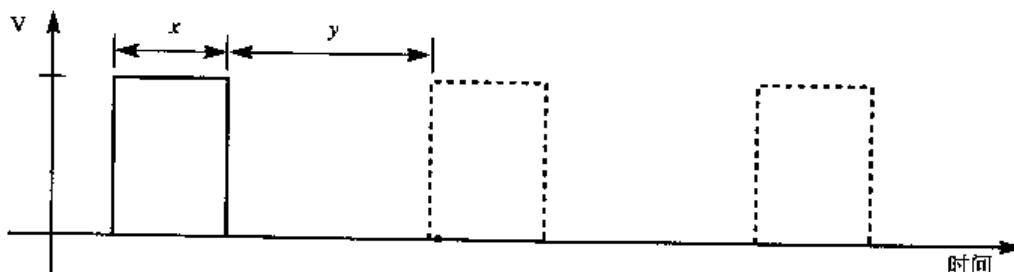


图 31.2 脉冲频率调制的基本定律

定时器 2

正如在第 3 章中所看到的, 原型 8052 和大多数较新的 8051 微控制器都包含一个额外的定时器——定时器 2, 本模式需要利用定时器 2。

定时器 2 的基本情况参见第 3 章, 定时器的基本情况参见第 11 章。

解决方案

利用定时器 2 (参见背景知识), 引脚 1.0 能够被编程为占空比 50% 的时钟输出。

如果要配置定时器^①为时钟发生器, 位 C/T2 (T2CON.1) 必须清为 0 而位 T2OE (T2MOD.1) 必须置 1。位 TR2 (T2CON.2) 用于定时器的启动和停止。时钟输出的频率决定于振荡器频率和定时器 2 捕捉寄存器的重新加载值 (RCAP2H、RCAP2L), 如下式所示:

$$\text{频率}_{\text{pulse}} = \frac{\text{频率}_{\text{Oscillator}}}{4 \times (65536 - \text{RCAP2H}, \text{RCAP2L})}$$

^① 正如第 3 章和第 11 章讨论的, 各种 8051 族的微控制器使用了不同的定时器 2 实现, 某些寄存器的名称可能变化。这段描述针对的是 Atmel 89x52/89x53, 但只要做很少的修改就能应用于其他 8051 微控制器。

例如，12MHz 晶振的最低频率为：

$$\text{频率}_{\min} = \frac{12000000}{4 \times (65536)} = 45.78\text{Hz}$$

类似的，最大频率为：

$$\text{频率}_{\max} = \frac{12000000}{4} = 3.0\text{MHz}$$

在时钟输出模式下，定时器 2 的回卷（即溢出）不会产生中断。

这个特性类似于定时器 2 被用作波特率发生器时的情况，所以可以同时使用定时器 2 作为波特率发生器和时钟发生器。然而需要注意，波特率和时钟输出的频率无法互相独立，因为两者都使用 RCAP2H 和 RCAP2L。

硬件资源

此模式需要利用定时器 2。在单处理器系统中，定时器 2 通常是最适合作调度器驱动的定时器，因而使用这种技术可能会影响系统的其他部分。

注意，如果可以采用双处理器设计（比如用基于 UART 或者基于中断的调度器），那么可以用主节点的定时器 2 驱动调度器，而把从节点的定时器 2 留着用于脉冲频率调制发生器，各种可能适用的多处理器设计详见第 6 篇。

可靠性和安全性

这种技术非常可靠。

可移植性

此技术只能用于基于 8052 的微控制器，也就是说，必须使用有定时器 2 的微控制器。虽然多数较新的 8051 系列微控制器都有定时器 2，但某些流行的 8051 系列微控制器（如 Atmel 的小 8051）没有定时器 2。

优缺点小结

- ◎ 输出宽频率范围脉冲频率调制信号的简单、有效的方法。
- ◎ 几乎没有软件或者 CPU 负荷。
- ◎ 需要占用定时器 2。

相关的模式和替代方案

最相近的模式是软件脉冲频率调制模式。另外，软件脉宽调制模式、硬件脉宽调制模式，特别是 3 级脉宽调制模式在某些情况下可作为替代方案的基础。

在硬件资源一节中也提到过，可以在多处理器设计的从节点上使用这一技术，各种多处理器模式详见第 6 篇。

例子：8052 的硬件脉冲频率调制

本节将给出基于硬件的脉冲频率调制设计的一个控制软件例子。

在源程序清单 31.1 和源程序清单 31.2 中，使用了一个 16 位（无符号）整型变量（PRM_reload_G）控制脉冲频率。如先前的讨论，频率为：

$$\text{频率}_{\text{脉冲}} = \frac{\text{频率}_{\text{振荡器}}}{4 \times (65536 - RCAP2H, RCAP2L)}$$

这样，对于 12MHz 振荡器而言，脉冲频率将反复地慢慢从 45Hz 增加到大约 3MHz。

```
/*
-----*
Main.c (v1.00)
-----
硬件脉冲频率调制的演示。
*-----*/
#include "Main.h"
#include "PRM_Hard.h"
// -----公有变量声明 -----
extern tWord PRM_reload_G;
/* ..... */
/* ..... */
void main(void)
{
    tLong Count = 0;
    PRM_Hardware_Init();
    while(1)
    {
        if (++Count > 10000UL)
        {
            // 缓慢地改变频率
            PRM_reload_G++;
            PRM_Hardware_Update();
            Count = 0;
        }
    }
}
*-----*
----文件结束
*-----*/
```

源程序清单 31.1 通用脉冲频率调制（基于硬件）例子的部分代码

```
/*
-----*
PRM_Hard.C (v1.00)
```

硬件脉冲频率调制（定时器2）的简单程序库例子
详见第31章。

```
/*
#include "Main.h"
-----公有变量定义
tWord PRM_reload_G = 0;
-----
PRM_Hardware_Init()
启动脉冲频率调制。
*/
void PRM_Hardware_Init(void)
{
    T2CON &= 0xFD; // 只清除 C/T2 位
    T2MOD |= 0x02; // 设置 T2OE 位（对基本 8052 仿制微控制器可省略）
    // 从最低频率开始（对 12MHz 晶振约为 45Hz）
    TL2      = 0x00; // 定时器 2 低字节
    TH2      = 0x00; // 定时器 2 高字节
    RCAP2L  = 0x00; // 定时器 2 重加载捕捉寄存器，低字节
    RCAP2H  = 0x00; // 定时器 2 重加载捕捉寄存器，高字节
    ET2      = 0;     // 没有中断
    TR2      = 1;     // 启动定时器 2
}
/*
PRM_Hardware_Update()
仅在需要改变脉冲频率时调用此函数。
所得的频率参见正文。
*/
void PRM_Hardware_Update(void)
{
    TR2      = 0;
    TL2      = PRM_reload_G % 256;
    RCAP2L  = TL2;
    TH2      = PRM_reload_G / 256;
    RCAP2H  = TH2;
    TR2      = 1;
}
-----文件结束 -----
*/

```

源程序清单 31.2 通用脉冲频率调制（基于硬件）例子的部分代码

进阶阅读

软件脉冲频率调制

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何不使用片上硬件而输出具有一定频率和脉冲宽度的方波脉冲？

背景知识

脉冲频率调制的基本背景资料参见硬件脉冲频率调制。

解决方案

用调度器建立脉冲频率调制输出很容易做到，实际上，介绍过的所有调度器的例子都可以使用。例如，看一下曾在第 14 章给出的源程序清单 31.3。

```
void main(void)
{
    //设置调度器
    SCH_Init_T2();
    //准备闪烁 LED 任务
    LED_Flash_T2();
    //加入 LED 闪烁任务（亮 1000ms，灭 1000ms）
    // -时间单位是时标间隔数（1ms 时标间隔）
    // （最大间隔/延迟是 65535 个时标间隔）
    SCH_Add_Task(LED_Flash_Update, 0, 1000);

    //启动调度器
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
```

源程序清单 31.3 基于软件的脉冲频率调制的一个简单形式

闪烁 LED 任务本身可以用源程序清单 31.4 中的程序实现。

```
void LED_Flash_Update(void)
{
    //将 LED 从关切换到开（反之亦然）
    if (LED_state_G == 1)
    {
```

```
    LED_state_G = 0;
    LED_pin = 0;
}
else
{
    LED_state_G = 1;
    LED_pin = 1;
}
```

源程序清单 31.4 闪光 LED 任务

这段程序代码以 0.5Hz 的频率闪烁 LED，如果使用合适的硬件接口也能驱动各种其他设备。同样的技术可以很容易地修改应用于可变频率的低频软件脉冲频率调制，源程序清单 31.5 说明了这一点。

```
void PRM_Soft_Update(void)
{
    // 位置变量加 1
    if (++PRM_position_G >= PRM_period_G)
    {
        PRM_position_G = 0;
        PRM_period_G = PRM_period_new_G;
        PRM_pin = 0;
        return;
    }
    // 生成脉冲频率调制输出
    if (PRM_position_G < (PRM_period_G / 2))
    {
        PRM_pin = 1;
    }
    else
    {
        PRM_pin = 0;
    }
}
```

源程序清单 31.5 可变频率软件脉冲频率调制的实现

源程序清单 31.5 的关键是变量 PRM position G, PRM period G 和 PRH period new G。

- PRM_period_G 是当前脉冲频率调制周期，用户不能改变。注意，如果更新函数每毫秒调度一次，则该周期是以毫秒计的。
 - PRM_period_new_G 是下一个脉冲频率调制周期—该周期可由用户按需要改变。注意，新的值只在脉冲频率调制周期末复制到 PRM_period_G，以避免噪音。如果更新函数每毫秒调度一次，此变量的单位也是毫秒。
 - PRM_position_G 是脉冲频率调制周期的当前位置，由更新函数增加。如果更新函数每毫秒调度一次，此变量的单位也是毫秒。

PRM_period_G、PRM_position_G 和脉冲频率调制输出的关系如图 31.3 所示。

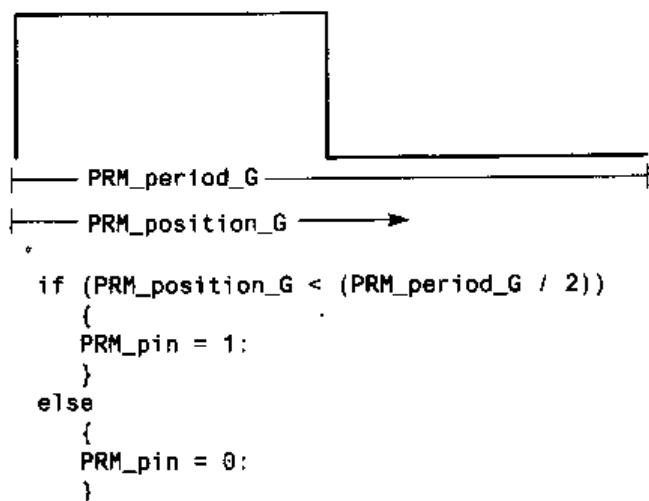


图 31.3 脉冲频率调制的软件实现

硬件资源

本模式只有很小的软件和硬件开销。

可靠性和安全性

这个方法很可靠。

可移植性

软件完全基于调度器的核心代码，因而和调度器一样，具有固有的可移植性。

优缺点小结

- ☺ 简单易用，CPU 和存储开销很小。
- ☺ 能够很容易地生成非常低频率的信号。
- ☹ 最大频率受限于调度器，如果调度器的定时周期是 T (秒)，则最大脉冲频率是 $1/2T$ (Hz)。

相关的模式和替代方案

替代方案参见硬件脉冲频率调制。

例子：8051 软件脉冲频率调制的通用代码

源程序清单 31.6 到源程序清单 31.8 中给出了一个完整的软件脉冲频率调制的简单例子。

```
/*
 *-----*
 * Port.H (v1.00)
 *
```

```
-----  
项目 PRM_Soft (参见第 31 章) 的端口头文件 (参见第 10 章)  
-----/*-----  
// ----- Sch51.C -----  
// 如果不需要报告错误, 注释掉此行。  
#define SCH_REPORT_ERRORS  
#ifdef SCH_REPORT_ERRORS  
// 用于显示错误码的端口  
// 只在报告错误时用到  
#define Error_port P2  
#end if  
// ----- PRM_Soft.c -----  
// 脉冲频率调制输出引脚  
sbit PRM_pin = P2^0;  
/*-----  
----文件结束-----  
-----*/
```

源程序清单 31.6 基于软件的脉冲频率调制的例子的部分代码

```
-----  
Main.c (v1.00)  
-----  
Demo of Software PRM.  
-----/*-----  
#include "Main.h"  
#include "2_01_12g.h"  
#include "PRM_Soft.h"  
/* ..... */  
/* ..... */  
void main()  
{  
    SCH_Init_T2();  
    PRM_Soft_Init();  
    // 每毫秒调用一次以更新脉冲频率调制输出  
    SCH_Add_Task(PRM_Soft_Update, 10, 1);  
    // 每分钟调用一次以改变脉冲频率调制控制值  
    SCH_Add_Task(PRM_Soft_Test, 0, 60000);  
    SCH_Start();  
    while(1)  
    {  
        SCH_Dispatch_Tasks();  
    }  
/*-----  
----文件结束-----  
-----*/
```

源程序清单 31.7 基于软件的脉冲频率调制的例子的部分代码

```

/*
PRM_Soft.c (v1.01)

软件脉冲频率调制的简单程序库。
*/
#include "Main.h"
#include "Port.h"
#include "2_01_12g.h"
#include "PRM_Soft.h"
//-----公有变量定义-----
// 将该变量的值设置为所需的脉冲频率
tWord PRM_period_new_G;
// -----私有变量定义 -----
// 脉冲频率调制计数器
static tWord PRM_position_G;
static tByte PRM_period_G;
/*
PRM_Soft_Init()
准备软件脉冲频率调制。
*/
void PRM_Soft_Init(void)
{
    //初始化主要变量
    PRM_period_G = 2;
    PRM_period_new_G = 2;
    PRM_position_G = 0;
}
/*
PRM_Soft_Update()
更新软件脉冲频率调制输出。
有三个主要变量（详见正文）：
1. PRM_period_G 是脉冲频率调制周期
   （如果每毫秒调度一次，单位是毫秒）
2. PRM_period_new_G 是新的脉冲频率调制周期，由用户设置
   （新的值只在脉冲频率调制周期末复制到 PRM_period_G，以避免噪音）
   （如果每毫秒调度一次，单位是毫秒）
3. PRM_position_G 是脉冲频率调制周期的当前位置
   （如果每毫秒调度一次，单位是毫秒）
*/
void PRM_soft_Update(void)
{
    // 位置变量加 1
    if (++PRM_position_G >= PRM_period_G)
    {
        PRM_position_G = 0;
        PRM_period_G = PRM_period_new_G;
        PRM_pin = 0;
        return;
    }
}

```

```
// 生成脉冲频率调制输出
if (PRM_position_G < (PRM_period_G / 2))
{
    PRM_pin = 1;
}
else
{
    PRM_pin = 0;
}
/*-----*
PRM_Soft_Test()
为了测试脉冲频率调制程序库，此函数每分钟调用一次，以改变脉冲频率调制输出设置。
*-----*/
void PRM_Soft_Test(void)
{
    PRM_period_new_G += 2;
    if (PRM_period_new_G >= 60000)
    {
        PRM_period_new_G = 2;
    }
}
/*-----*
----文件结束 -----
*-----*/

```

源程序清单 31.8 基于软件的脉冲频率调制的例子的部分代码

进阶阅读

Chapter 32

模拟-数字转换器（ADC）的应用

引言

模拟信号记录是许多状态监视、数据采集和控制系统的重要组成部分。

在本章中，将讲述如何用 8051 微控制器读取模拟量值。下面列出了相关的模式：

- 单次模数转换讲述了如何用微控制器不定期地测量模拟电压信号。
- 模数转换前置放大讲述了如何放大模拟信号，以使其电压幅值范围适合于随后的模数转换。
- 序列模数转换讲述了如何用微控制器记录一个序列的模拟采样。
- A-A 滤波器讲述了如何用滤波器滤除模拟信号的高频分量。
- 电流传感器讲述了如何监视流过直流负载的电流。

单次模数转换

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

怎样以不定的间隔或者很不频繁的间隔测量模拟电压或者电流信号？

背景知识

在本节中，将介绍一些基本背景材料。

测量电压

看一下图 32.1 中从电位器处引入的模拟输入。

图中电路所得的模拟电压范围为 0~5V，如果用片上或片外模拟-数字转换器（ADC）对其进行转换（这类 ADC 器件很常用），此信号就可以用于人机接口^①。

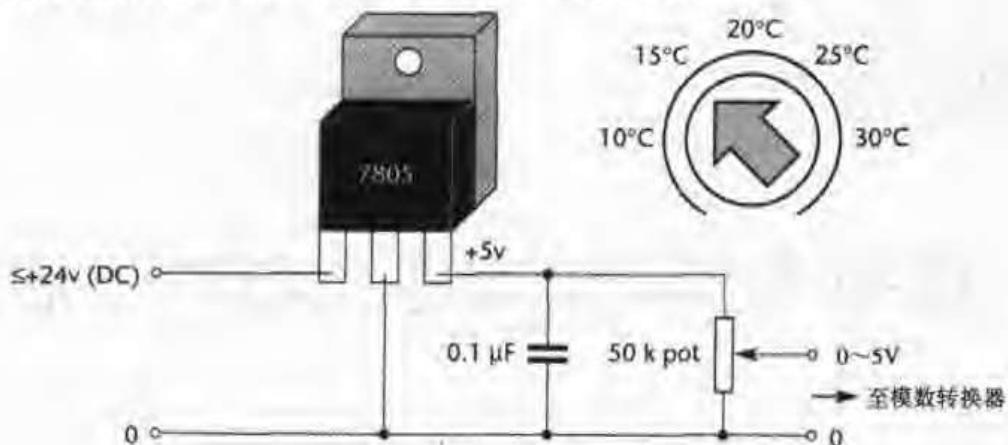


图 32.1 用人机接口设计的人机接口电路

这种通用的方法有很多的用途。例如，见图 32.2。这里使用了三个电位器（和合适的三通道模数转换器）测量工程挖掘机的角度。通过测量点 A、B 和 C 处的角度，就能够确定铲（X）的位置（即，铲的深度）。

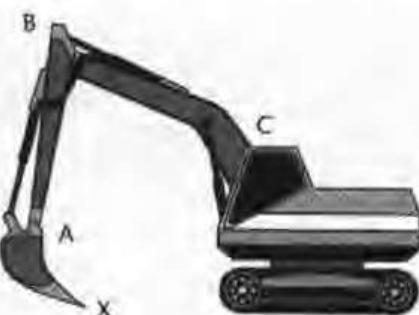


图 32.2 用电位器测量工程挖掘机的角度

注意，通过测量点 A、B 和 C 处的角度，就能够确定铲（X）的位置（即，铲的深度）。

电流的测量

在前面的简单例子中，举例说明了在嵌入式系统中模拟电压信号是获得信息的有效途径。然而，模拟电流信号的测量更有用，特别是在工业应用中。

电流信号更有用的原因参见图 32.3。

图 32.3 中展示了一个传感器，假定其通过很长的导线（电阻为 R_{wire} ）连接至微控制器的模拟电压输入端。假设传感器测量温度并产生一模拟输出，5V 表示 100°C，1V 表示 0°C。并且只可能是正的温度，如果传感器或者导线损坏，测量出的电压将为 0V，表示出错了。

^① 请注意，虽然这个电路被广泛地使用，但可能有更好的人机接口电路设计方法，参见相关模式和替代方案。

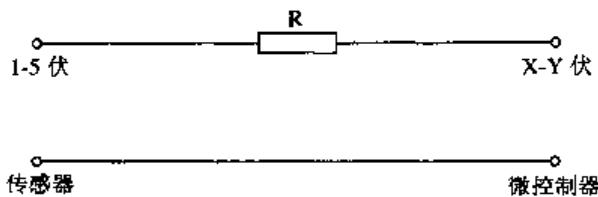


图 32.3 输出电压的传感器的问题

这个方案有两个问题：

1. 微控制器处的电压会低于或大大低于传感器处的电压。可能需要仔细调整传感器的程序代码，以抵消导线电压降的影响。
2. 导线电阻（从而其上的电压降）是随温度而变的，所以即使仔细调整了代码后也很难得到准确的读数。

这个问题的一个很好的解决方法是在传感器处将模拟读数数字化，再将电压信号的数字表示传达给主微控制器，第6篇中讨论的技术可用于这种解决方案的实现。

另一种在加工工业非常普遍的解决方法是使用变化的电流（例如 4~20mA）表示传感器信息（如图 32.4 所示）。

因为电流源传感器能够根据需要调整输出电压，使电流保持在指定水平，所以即使导线电阻随温度而变，也能在长距离上良好地工作。注意，在微控制器端，可以简单地用一个固定电阻器将电流信号转换为电压信号。

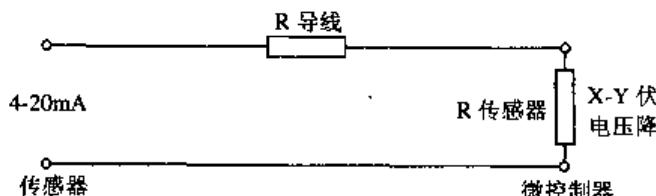


图 32.4 电流模式传感器的原理图

解决方案

本节先简单地考虑一下可用于测量模拟电压或者电流信号的一些硬件设计方法。具体地说，将考虑以下 4 种选择：

- 使用具有片上（电压模式）模数转换器的微控制器
- 使用外部串行总线（电压模式）模数转换器
- 使用外部并行总线（电压模式）模数转换器
- 使用电流模式模数转换器

后面的例子中将给出相应的软件程序库。

使用具有片上（电压模式）模数转换器的微控制器

许多 8051 系列的微控制器包含片上模数转换器。通常，使用片上硬件能增强可靠性，这

是因为硬件和软件的复杂性都降低了。另外，片上硬件的设计通常外形更小，系统成本更低。这里有两个 8051 微控制器的片上模数转换组件的例子。

Infineon C515C

摘自 Infineon^② C515C 数据手册：

C515C 内建了一个高性能/高速 6 通道 10 位模数转换器（ADC）。以逐次逼近法进行模数转换，并使用了自校准机制以减少和补偿偏移和线性度误差。C515C 的模数转换器提供了以下特性：

- 8 个多路切换输入通道（端口 6），也可以用作数字输入。
- 10 位分辨率。
- 单次或者连续转换方式。
- 可内部或者外部触发启动转换。
- 使用电容阵列的逐次逼近转换技术。
- 内建的偏移和线性度误差校准。

Analog Devices ADμC812

摘自 Analog Devices^③ ADμC812 数据手册：

ADμC812 的模数转换模块包含一个快速、多通道、12 位、单电源模数转换器。该模块提供了多路切换、采样/保持、片内参考源、校准特性和模数转换器。模块内的所有组件很容易由微控制器核心通过特殊功能寄存器配置。

模数转换器采用基于电容数模转换器的传统逐次逼近转换技术。可接受范围在 0 到+VREF 内的模拟输入。芯片内还提供了一个高精度、低漂移的 2.5V 参考源。也可通过外部 VREF 引脚引入外部参考源，代替内部参考源。外部参考源的电压范围为 2.3V~AVDD。

可由软件启动单次或者连续转换，或者由外部引脚上的转换信号启动，也可以配置定时器 2 反复产生模数转换触发信号。模数转换器也可配置为以 DMA（直接存储器存取）模式工作，此时，模数转换模块连续地进行转换和采样值读取，不需要微控制器核心的干预。

模数转换器核心包含自动校准和系统校准功能，能确保长期工作和不同温度下的转换准确性。片内的带隙参考和绝对温度成比例，接至模数转换器的前端多路切换器，就可以作为温度传感器。

使用外部并行总线（电压模式）模数转换器

片上模数转换器的传统替代方案是外部并行总线模数转换器。通常，并行总线模数转换器有以下的优点和缺点：

- ◎ 能够提供快速的数据传输。
- ◎ 通常比较便宜。
- ◎ 软件的框架结构非常简单。

② www.infineon.com

③ www.analog.com

- ② 需要很多的端口引线。就16位模数转换器来说，数据传输需要16根引脚，再加上一到三个引脚用于控制数据传输。
- ③ 在某些情况下，接线的复杂性可能导致可靠性问题。
后面将会给出并行总线模数转换器的应用例子。

使用外部串行总线(电压模式)模数转换器

许多更新的模数转换器都有串行总线接口。通常，串行总线模数转换器有以下优点和缺点：

不管模数转换器的分辨率是多少，都只需要很少的端口引线(两至四根)。

需要串行总线协议的片上支持，或者使用合适的软件库。

数据传输可能比并行总线的方案慢。

相对较昂贵。

后面将给出两个串行总线模数转换器应用的例子。

使用电流模式模数转换器

正如在背景知识中所讨论的，在一些情况下，基于电流的信号传输很有用。

现在已经有许多电流模式传感器组件(如，Burr-Brown[®] XTR105)和模数转换器(如，Burr-Brown RCV420)可用。

另外，电流传感器一节讨论了使用电压模式的模数转换器进行电流测量。

硬件资源

使用片上模数转换器通常至少占用一个输入引脚，使用外部模数转换器会占用大量的引脚。

使用片上模数转换器可能也会增加微控制器的功耗。

可靠性和安全性

使用模数转换器没有特别的安全性要求。如果其他条件等同，使用片上模数转换器很可能比使用外部模数转换器更可靠，这主要是因为片上解决方案的硬件复杂性比外部解决方案小得多。

可移植性

模数转换器各方面的特性各不相同。例如，为一个串行总线模数转换器编写的程序代码不经改写经常无法用在另一个串行总线模数转换器上。但是，所需的修改通常较小。

优缺点小结

- ① 对许多系统，模数转换输入是必需的。
- ② 带模数转换器的微控制器比不带模数转换器的微控制器昂贵。

相关的模式和替代方案

现在，许多微控制器都有多个模数转换通道，可用于实现廉价的用户输入接口，例如设定温度、工作点等等。

这类模拟输入的主要优点是能向用户直接提供反馈。例如，简单地在控制电位器的周围标上刻度，指示所需的温度。这种输入设计的成本常常比对应的数字式设计（两个开关加上显示）低得多（如图 32.5 所示）。

然而，几乎在所有的情况下，数字式设计都能提供更精确的控制，并且不会随时间而改变（而电位器的性能很可能随时间而改变）。此外，如果希望用软件改变温度设定，用数字式设计可以很容易地做到，更新温度显示即可。而电位器通常无法由软件控制旋转到新的温度设定刻度上。

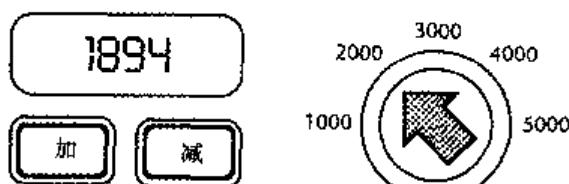


图 32.5 两种可能的用户界面设计，一个基于 LCD/LED 显示和开关，另一个基于模拟电位器

尽管如此，对不需要高精度的应用场合，模拟设计通常就够用了，例如，控制室内恒温器，电位器就很合适。

例子：外部 SPI 模数转换器的应用

这个例子说明了外部、串行总线（SPI）模数转换器的用法（源程序清单 32.1~源程序清单 32.3）：SPI 协议已经在第 24 章详细讲述过了。

硬件系统包括一个 Atmel AT89S53 微控制器和一个 Maxim[®] Max1110 模数转换器，芯片间的连接见图 32.6。

```
/*
Port.H (v1.00)

项目 SPI_ADC 的端口头文件 (参见第 10 章)
*/
// ----- SPI_Core.C -----
// 为所需的片选信号建立 sbits
sbit SPI_CS = P1^4;
// 注意：引脚 P1.4、P1.5、P1.6 和 P1.7 也使用了——参见正文
/*
---文件结束 ---
*/

```

源程序清单 32.1 串行总线（SPI）模数转换器应用例子的部分源程序

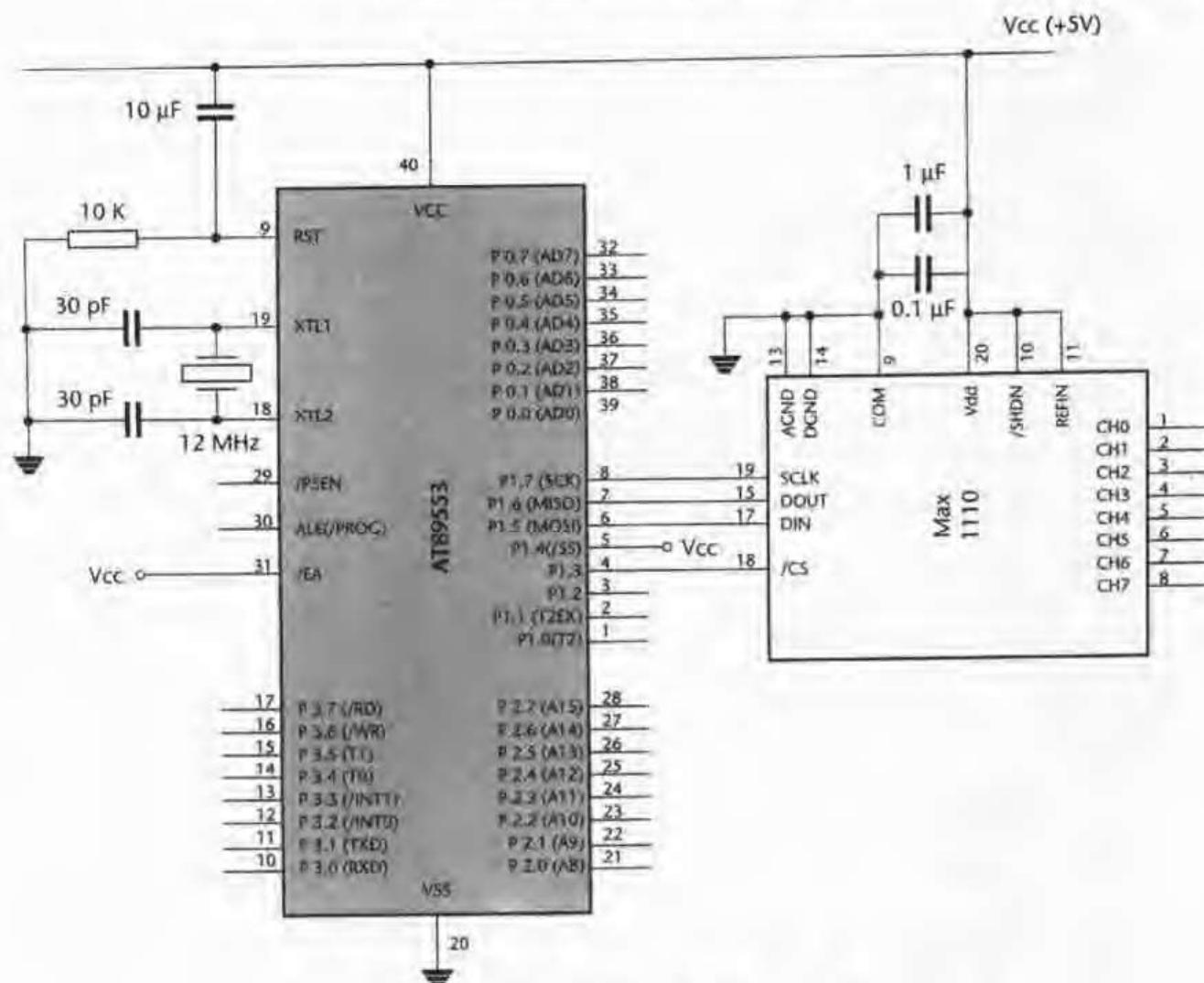


图 32.6 连接 SPI 模数转换器和 8051 微控制器

```
/*
 Main.C (v1.00)
 -----
 SPI 代码库的简单测试程序。
 读 Max1110/1111 SPI 模数转换器。
 */
#include "Main.h"
#include "SPI_Core.h"
#include "SPI_1110.h"
#include "Delay_T0.h"
// 此处定义了测试程序中的错误代码变量
tByte Error_code_G = 0;
void main(void)
{
```

```

tByte Data1 = 0;
tByte data2 = 0;
tWord Data_address = 0;
// 详见正文
// SPI 控制寄存器
// 位 0 = SPR0
// 位 1 = SPR1 (这两位控制了时钟的频率)
// 位 2 = CPHA (传输格式, 参见 AT89S53 文档, 第 15 页)
// 位 3 = CPOL (时钟极性, 1=空闲时高, 0=空闲时低)
// 位 4 = MSTR (主节点为 1, 从节点为 0)
// 位 5 = DORD (数据传输次序, 1 为最低位先传输, 0 为最高位先传输)
// 位 6 = SPE (使能 SPI)
// 位 7 = SPIE (如果 ES 也为 1 的话, 使能 SPI 中断)
// 为了和 MAX1110 模数转换器接上, 需要频率 50-500kHz 的时钟, 对于 12MHz 的振荡器
// 和 SPR0、SPR1 分别设为 1、0 时, SPI 的速度是 Fosc/64=187.5kHz
//
// CPHA 和 CPOL 应为零, 参见 MAX1110 的文档
// DORD 应为零 (高位优先)
// MSTR、SPE、SPIE 应为 1
// ->SPCR = 0x52;
SPI_Init_AT89S53(0x52);
while(1)
{
    // 读模数转换器字节
    Data2 = SPI_MAX1110_Read_Byte();
    // 显示数据
    P2 = 255-Data2;
    // 显示错误代码 (如果有的话)
    P3 = 255-Error-code-G;
    Hardware_Delay_T0(1000);
}
*/
-----文件结束-----
*/

```

源程序清单 32.2 串行总线 (SPI) 模数转换器应用例子的部分源程序

```

/*
-----SPI_1110.C (v1.00)
-----简单的 Atmel AT89S53 SPI 程序库
-----用于从 Max1110/1111 模数转换器读取数据
*/
#include "Main.H"
#include "Port.h"

```

```
#include "SPI_Core.h"
#include "SPI_1110.h"
#include "TimeoutH.h"
// -----公有变量声明-----
// 错误代码变量
//
// 错误代码的显示端口和详情参见 Port.H
extern tByte Error_code_G;
/*-----*/
SPI_MAX1110_Read_Byte()
从模数转换器读取一个字节的数据。
*/
tByte SPI_MAX1110_Read_Byte(void)
{
    tByte Data, Data0,Data1;
    // 0.拉低/CS引脚以选通器件
    SPI_CS = 0;
    // 1.向MAX1110发送一个控制字节
    // 位7=1(控制字节的开始)
    // 位6=SEL2(SEL2, SEL1, SEL0选择输入通道)
    // 位5=SEL1(参见Maxim文档)
    // 位4=SEL0
    // 位3=1单极型,0双极型
    // 位2=1单端,0差分驱动
    // 位1=1正常运行,0省电模式
    // 位0=1外部时钟,0内部时钟
    //
    // 控制字节0x8F设置为单端单极性模式,输入通道0(引脚1)
    // 0(pin 1)
    SPI_Exchange_Bytes(0x8F);
    // 2.请求的数据在SO上移出,通过发送两个空字节
    Data0 = SPI_Exchange_Bytes(0x00);
    Data1 = SPI_Exchange_Bytes(0x00);
    // 数据包含在Data0的0~5位引脚Data1的6~7位——将这些字节移位得到一个混合字节
    Data0 <<= 2;
    Data1 >>= 6;
    // 3.将/CS引脚拉高,操作结束
    SPI_CS = 1;
    // 4.返回所需的数据
    return Data; // 返回SPI数据字节
}
/*-----文件结束-----*/
/*-----*/
```

源程序清单32.3 串行总线(SPI)模数转换器应用例子的部分源程序

例子：外部 I²C 模数转换器的应用

这个例子说明了外部串行总线 (I²C) 模数转换器的用法（源程序清单 32.4~源程序清单 32.6），I²C 协议已经在第 23 章详细讲述过了。

模数转换硬件包括一个 Maxim[®] Max127 模数转换器，Max127 与微控制器的连接见图 32.7。

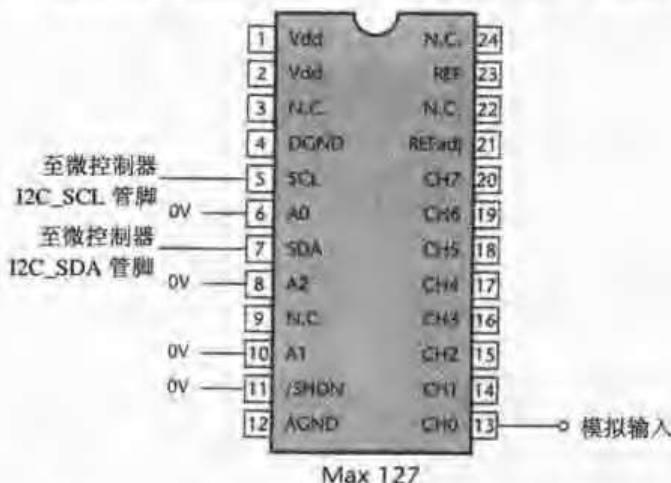


图 32.7 Maxim Max127 串行总线 (I²C) 模数转换器

```
/*
Port.H (v1.00)
端口头文件 (第 10 章) 见第 32 章的内容。ASC_M12 的项目
// ----- Sch51.C -----
// 如果不需要错误报告, 注释掉此行。
#define SCH_REPORT_ERRORS
#define SCH_REPORT_ERRORS
// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P2
#endif
// ----- I2C_Core.C -----
// 两线 I2C 总线
sbit I2C_SCL = P1^7;
sbit I2C_SCL = P1^6;
/*-----文件结束-----*/
*/
```

源程序清单 32.4 串行总线 (I²C) 模数转换器应用例子的部分源程序

```
/*
Main.C (v1.00)

-- I2C (Max127 模数转换器) 程序库的简单测试程序。
将 Max127 连接到库文件中所说的 SDA 和 SCL 引脚 (I2C_Core.C)。
通常总线上不需要终端电阻器
*/
#include "Main.h"
#include "I2C_m127.h"
#include "Delay_T0.h"
extern tByte ADC_G;
// 此处定义了测试程序中的错误代码变量
// (通常在调度器程序库中)
tByte Error_code_G = 0;
void main(void)
{
    while(1)
    {
        I2C_ADC_MAXq27_Read();
        P1 = ADC_G;
        P2 = Error_code_G;
        Hardware_Delay_T0(1000);
    }
}
/*-----文件结束-----*/

```

源程序清单 32.5 串行总线(I²C)模数转换器应用例子的部分源程序

```
/*
I2C_m127.C (v1.00)

-- Max127 (I2C) 模数转换器的程序库
*/
#include "Main.h"
#include "Port.h"
#include "I2C_Core.h"
#include "I2C_m127.h"
#include "Delay_t0.h"
//----公有变量定义-----
// 模数转换器的值
tByte ADC_G;
// ----公有变量声明-----
// 错误代码——参见调度器
extern tByte Error_code_G;
```

```
-----私有常数-----
// 芯片地址 = 0101xxxxW
#define I2C_MAX127_ADDRESS (80)
// 起始位设置
// 正常电源模式（非省电模式）
// 电压范围 0~5V
#define I2C_MAX127_MODE (0x80)
-----私有变量定义 -----
// 模数转换器通道 (0~7)
// ***所需的值为通道值左移 4 位***
// ***使用通道 2 ***
static tByte I2C_MAX127_Channel_G = 0x20;
/*-----*
I2C_ADC_MAX127_Read()
从 I2C12 位模数转换器读数据
ADC_Channel_G 提供所使用的信道
该版本只读取 8 位数据
*-----*/
void I2C_ADC_Max127_Read()
{
    I2C_Send_Start(); // 读 I2C 12 位模数转换器
    // 发送模数转换器地址 (带写请求)
    if (I2C_Write_Byte(I2C_MAX127_ADDRESS | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_ADC_MAX127;
        return;
    }
    // 设置模数转换器模式和通道——参见上面的程序
    if (I2C_Write_Byte(I2C_MAX127_MODE | I2C_MAX127_Channel_G))
    {
        Error_code_G = ERROR_I2C_ADC_MAX127;
        return;
    }
    I2C_Send_Stop(); // 生成停止条件
    I2C_Send_Start(); // 再次生成启动条件
    // 发送 Max127 地址 (带读访问请求)
    if (I2C_Write_Byte(I2C_MAX127_ADDRESS | I2C_READ))
    {
        Error_code_G = ERROR_I2C_ADC_MAX127;
        return;
    }
    // 从 I2C 总线接收第一个 (最高) 字节
    ADC_G = I2C_Read_Byte();
    I2C_Send_Master_Ack(); // 主节点确认
    // 这里温度只需要精确到 1°C
```

```

// -丢弃低字节(进行一次空读)
I2C_Read_Byte();
I2C_Send_Master_NAck(); // 主节点NACK
I2C_Send_Stop(); // 生成停止条件
}
/*-----*
----文件结束-----
*-----*/

```

程序清单 32.6 串行总线(I²C) 模数转换器应用例子的部分源程序

例子：外部并行总线模数转换器的应用

这个例子说明了 8 位并行总线模数转换器 Maxim[®] Max150 (图 32.8) 的应用 (源程序清单 32.7~源程序清单 32.9)。

```

/*-----*
Port.H (v1.00)

项目 ADC_m150 (参见第 32 章) 的端口头文件 (参见第 10 章)
*-----*/
// ----- ADC_M150.C -----
//与 Max150 并行总线模数转换器的接口
sbit ADC_MAX150_NOT_Read_pin = P1^0;
sbit ADC_MAX150_NOT_Write_pin = P1^1;
sbit ADC_MAX150_NOT_Int_pin = P1^2;
#define ADC_MAX150_port P2
// ----- Bargraph.C -----
//将 LED 经过合适的电阻从+5V 连至这些引脚
//[详见第 7 章]
//如果需要，8 根端口引线可以分布在几个端口上。
sbit Pin0 = P2^0;
sbit Pin1 = P2^1;
sbit Pin2 = P2^2;
sbit Pin3 = P2^3;
sbit Pin4 = P2^4;
sbit Pin5 = P2^5;
sbit Pin6 = P2^6;
sbit Pin7 = P2^7;
/* -- *
----文件结束 -----
*-----*/

```

源程序清单 32.7 并行总线(8 位) 模数转换器应用例子的部分源程序



图 32.8 并行总线(8位)模数转换器 Max150 的引脚连接

```

/*
Main.c (v1.00)

模数转换器->柱状图显示的演示程序
所需的链接程序选项（详情参见第 14 章）：
OVERLAY
(main ~ (ADC_Get_Sample,Bargraph_Update),
sch_dispatch_tasks ! (ADC_Get_Sample,Bargraph_Update))
*/
#include "Main.h"
#include "2_01_12g.h"
#include "ADC_m150.h"
#include "BarGraph.h"
/* ..... */
/* ..... */
void mai(void)
{
    SCH_Init_T2();           // 设置调度器
    ADC_MAX150_Init();       // 设置模数转换器
    BARGRAPH_Init();         // 设置柱状图显示(P4)
    // 规律地定期读取模数转换器
    SCH_Add_Task(ADC_MAX150_Get_Sample, 10,1000);
    // 这里简单地显示计数(柱状图显示)。
    SCH_Add_Task(BARGRAPH_Update, 12, 1000);
    // 所有的任务已加入：启动调度器
    SCH_Start();
    While(1)
    {
        SCH_Dispatch_Tasks();
    }
}
/*-----文件结束-----*/

```

源程序清单 32.8 并行总线(8位)模数转换器应用例子的部分源程序

```
/*
  ADC_m150.c (v1.00)

  简单的C515c单通道8位模数转换(输入)程序库
  - 使用Max150 8位并行总线模数转换器。
  详见第32章。
*/
#include "Main.H"
#include "Port.h"
#include "Bargraph.h"
//-----公有变量定义-----
//存储最新的模数转换读数
tByte Analog_G;
// -----公有变量声明 -----
extern tByte Error_code_G;
/*
  ADC_MAX150_Init()
  设置Max150模数转换器。使用WR-RD模式(参见数据手册)
*/
void ADC_MAX150_Init(void)
{
    // NOT read引脚设置为高
    ADC_MAX150_NOT_Read_pin = 1;
    // NOT write引脚设置为高
    ADC_MAX150_NOT_Write_pin = 1;
    //设置NOT Int引脚为读
    ADC_MAX150_NOT_Int_pin = 1;
}
/*
  ADC_MAX150_Get_Sample()
  从模数转换器读取单个采样数据(8位)。
*/
void ADC_MAX150_Get_Sample(void)
{
    tWord Time_out_loop = 1;
    // 拉低NOT Write引脚，启动转换
    ADC_MAX150_NOT_Write_pin = 0
    // 从模数转换器取采样(简单地用循环等待时间到)
    while ((ADC_MAX150_NOT_Int_pin == 1) && (Time_out_loop != 0));
    {
        Time_out_loop++; // 禁止，为了使用dScope...
    }
    if (!Time_out_loop)
    {
        // 时间到
        Error_code_G =
        Analog_G = 0;
    }
    else
```

```

{
// 端口设置为读模式
ADC_MAX150_port = 0xFF;
// NOT read 引脚设置为低
ADC_MAX150_NOT_Read_pin = 0;
// 模数转换器结果已得出
Analog_G = ADC_MAX150_port;
// NOT read 引脚设置为高
ADC_MAX150_NOT_Read_pin = 1;
}
// NOT write 引脚拉为高
ADC_MAX150_NOT_Write_pin = 1;
}

-----文件结束-----
*/

```

源程序清单 32.9 并行总线（8位）模数转换器应用例子的部分源程序

例子：C515C 内部模数转换器的应用

Infineon C515C 片上模数转换器应用的完整程序库请参见序列模数转换。

进阶阅读

Lynn, P. and Fuerst, W. (1998) *Introductory Digital Signal Processing with Computer Applications*, Wiley, Chichester.

模数转换前置放大器

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何将模拟电压信号转换到适合于随后的模拟数字转换的电压范围？

背景知识

在 5V 系统中，模数转换器一般对从 0V 到 5V 左右范围内的模拟信号进行转换。如果模拟信号的范围为 0~5mV，则需要在模数转换之前先放大这个信号，否则转换出的数字信号将不能很好地反映模拟信号。

本模式将讲述一些适用的电路。注意，本模式只用到硬件，不需要软件。

解决方案

在按比例缩放模拟信号的设计中，运算放大器是各种广泛使用的解决方案的基础，本模式中也将使用运算放大器。

可用的运算放大器的选择范围很宽，从经典的741或者411芯片到更新的诸如Microchip MCP601的芯片。^⑧

这里将讨论运放的两种基本应用：放大和电压平移。

电压放大

当用于实现简单的电压放大器时，所有的运算放大器的用法都是一样的。图32.9说明了具体的方法。这个电路的增益 G ，由下式给定：

$$G = \frac{V_{out}}{V_{in}} = \frac{R_1 + R_2}{R_1} = 1 + \frac{R_2}{R_1}$$

这个电路应用起来非常简单，这将在下面的例子中说明。注意，如果需要精确的增益，必须使用1%精度的高品质电阻，5%的精度通常不够用。

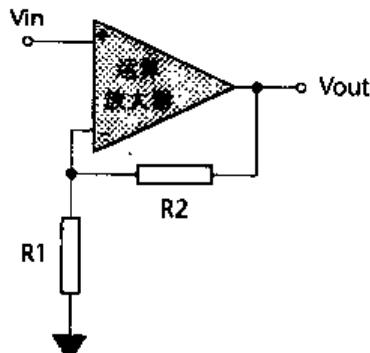


图32.9 用运算放大器放大电压信号

电压平移

假设需要采样如图32.10所示的语音信号。

信号的最大值大约为50毫伏，所以需要在采样以前先放大。但是，此信号的平均值约为0V；因此必须将电压平移到正电压范围，以便处理。

图32.11是一个简单的运放电压平移电路。

此电路的输出由下式给定：

$$V_{out} = V_{ref} - \frac{R_2}{R_1} V_{in}$$

另一个电压平移的替代方案见图32.12。此电路的输出由下式给定：

^⑧ www.microchip.com

$$V_{out} = \frac{R4}{R3} V1 - \frac{R2}{R1} V_{in}$$

原理图见图 32.12，此电路是一个非常灵活的电路。



图 32.10 平均值大约为 0V 的语音信号

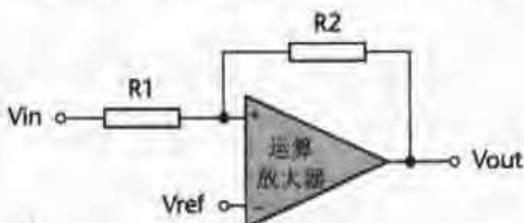


图 32.11 例示了一个可以用于电压平移的运放电路

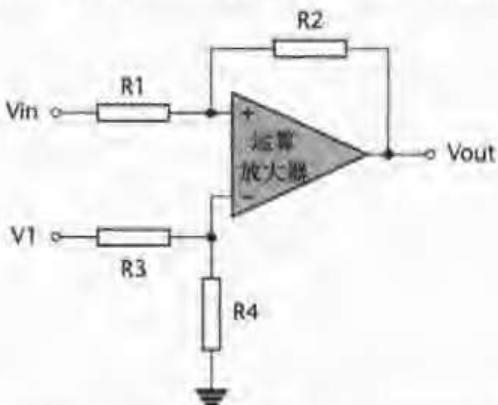


图 32.12 运放电压平移电路的一种替代方案

硬件资源

使用这个模式并不需要占用微控制器上的硬件资源（例如，CPU 的机时或者存储器）。但是显而易见，运算放大器和电阻将增加附加的成本。

此外，一些运算放大器电路可能需要正负电源供应（例如，+15V、-15V）。这会增加电源供应系统的设计复杂性，而且难以用电池供电（或增加电池供电的成本）。现在，单电源的运算放大器越来越多，经常能够用在放大器设计中。

可靠性和安全性

无特别的可靠性或者安全性要求。

可移植性

这个模式只需要硬件，能和任何微控制器配合使用。

优缺点小结

- ◎ 简单而且有效。
- ◎ 可能需要正负极性的电源。

相关的模式和替代方案

例子：一个增益为 1000 的放大器

假设，有一个最大输出为 5mV 的传感器，而模数转换器的输入为 5V，所以需要 1000 的增益。

根据第 607~608 页上的等式选择 R2 和 R1，使它们的阻值比为 1000:1；这里取 $R1=1k\Omega$, $R2=1m\Omega$ (1% 误差)。

例子：话筒前置放大器

图 32.13 所示的（由 Analog Devices 的数据手册中的电路更改而来）是电压放大器的一个应用：驻极体话筒的前置放大器，其放大级的增益大约为 10。

注意，AD8517 是单电源运算放大器。

$R1$ 用于驻极体话筒的偏置， $C1$ 用于隔离直流电压分量。当 $R2 = 10 \times R1$ 时，放大器增益的大小约为 $R3/R2$ 。 $VREF$ 应等于 $1/2 1.8V$ 以达到最大电压摆幅。

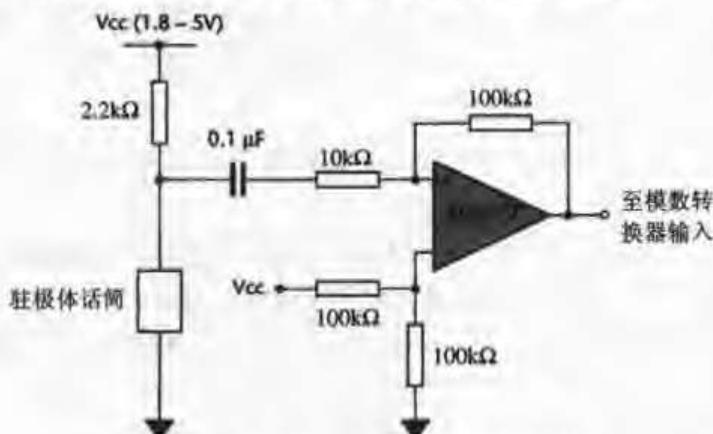


图 32.13 话筒的前置放大器

进阶阅读

Elgar, P. (1998) Sensors for Measurement and Control, Longman, London.

Franco, S. (1998) Design with Operational Amplifiers and Analog Integrated Circuits, 2nd edn, McGraw-Hill, Boston, MA.

序列模数转换

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

怎样记录一个序列的模拟采样？

背景知识

在单次模数转换模式中，主要关心用模拟信号解决诸如下面的一些问题：

- 用户需要多高的集中供暖温度？
- 起重机的当前角度是多少？
- 温室的湿度是什么水平？

在现在这个模式中，主要关心怎么记录模拟采样序列，以解决诸如下面的一些问题：

- 汽车的加速有多快？
- 飞机的转向速度有多快？
- 声音的频率是多少？

例如，假设需要为环境组织设计一个用于记录和自动分类鲸在南极水下的歌声的系统。此嵌入式系统将用塑料包裹，并附着在浮标上。这个装置要靠电池供电运行两年，当在附近探测到鲸的活动时需要向基地发送无线电广播。

在这种情况下，需要用水压传感器（水听器）将鲸歌引起的水压变动转换成模数转换器能够处理的电压信号。

模数转换器的输出是一个数列（如图 32.14 所示）。这些数字（例如 0.89）的平均值没有意义，而从它们组成的数列可以计算出组成鲸歌的各频率分量。

在本模式中，关心的是固定采样频率下的模拟信号序列。例如，对鲸歌以 50kHz 的频率进行采样。为了记录这样的序列，将以单次模数转中讨论的技术作为基础。然而，正如将在解决方案中讨论的，模拟信号序列的记录对开发者提出了一些新的挑战。

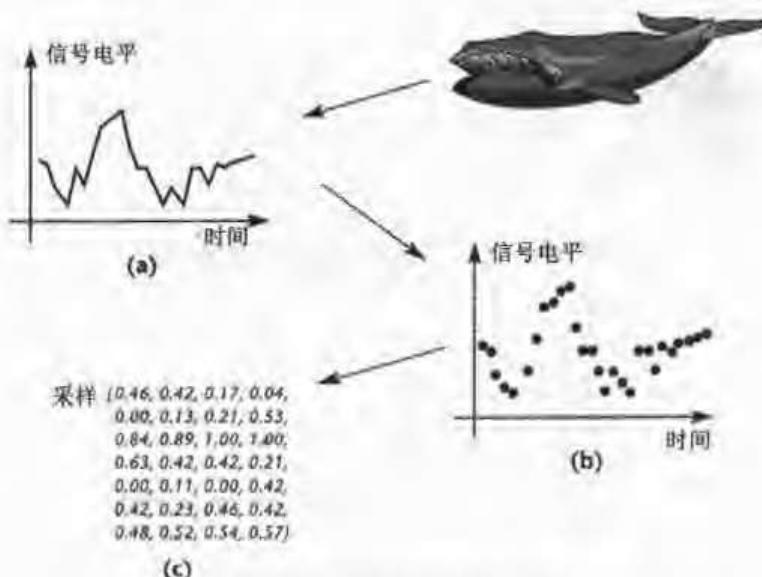


图 32.14 对鲸歌进行模拟数字转换采样

(a) 原始的模拟信号，从水下听音器得到的连续变化的电压信号。(b) 离散化了的信号。
(c) 存储在计算机上的鲸歌的数字表示

解决方案

序列模数转换的实现需要经过几个关键的设计阶段：

1. 需要确定所需的采样频率
2. 需要滤除输入信号中的高频分量
3. 需要决定所需的采样位数
4. 需要使用合适的软件结构
5. 需要选择合适的模数转换器

现在依次讨论以上各个设计阶段。

决定所需的采样频率

正如在背景知识一节中提到的，本模式关心的是以固定频率采样所得的模拟信号序列。第一个重要的设计决策就是决定所需的采样频率。

监视和信号处理系统

假如需要建立一个语音识别系统。目的是从语音波形（例如，话筒产生的随时间变化的电压波形）辨识出所说的单词（图 32.15）。

为了确定这类系统的采样频率，必须知道被采样信号的带宽。然后需要用至少两倍于此带宽的频率进行采样（这个频率被称作尼奎斯特频率）。

记住，带宽仅仅指信号中希望测量的最高频率分量。例如，图 32.16 中有一个理想化的例子：频率为 F_{sine} 的纯音（正弦波）。在这种情况下，信号带宽等于纯音的频率 (F_{sine})，正确

地表示此信号需要以 F_{sine} 的两倍频采样。

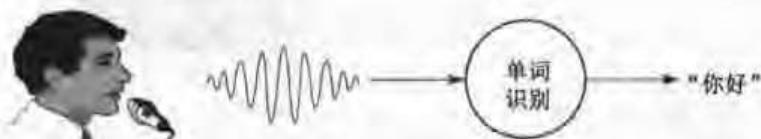


图 32.15 一个简单的语音识别(分类)系统

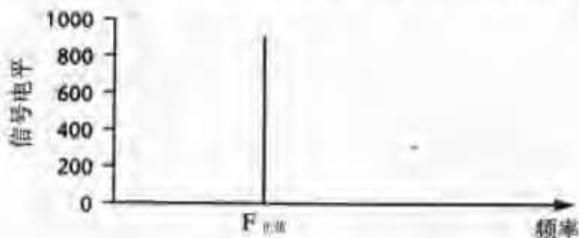


图 32.16 纯音的频域表示

更一般的情况是，信号具有宽带的特性（如图 32.17 所示）。

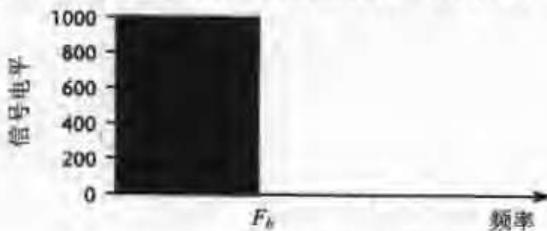


图 32.17 宽带信号的频域表示

在这种情况下，带宽指信号的最高频率， F_b 。通常，可能要用诸如频谱分析仪之类的专用硬件确定需要采样的信号的最高频率分量。就语音信号来说，如果使用这样的硬件进行测量，会发现最高频率分量大约为 20kHz，这意味着需要以大约 40kHz 的频率采样（如图 32.18 所示）。

注意，并不总是能够以最高信号频率为准进行采样，即使可以，也可能在经济上不合理；参见下面“滤除高频分量”，其中讨论了可用于处理这个问题的技术。



图 32.18 宽带信号(带宽 20kHz)的频域表示

控制系统

如果希望实现下列功能：

- 语音识别
- 心电图记录

- 听觉反应记录
- 振动监测

则需要用频谱分析仪分析信号的最高频率分量，并选择相应的采样频率。

可是，假设一下，研发一个如图 32.19 所示的数字控制系统。

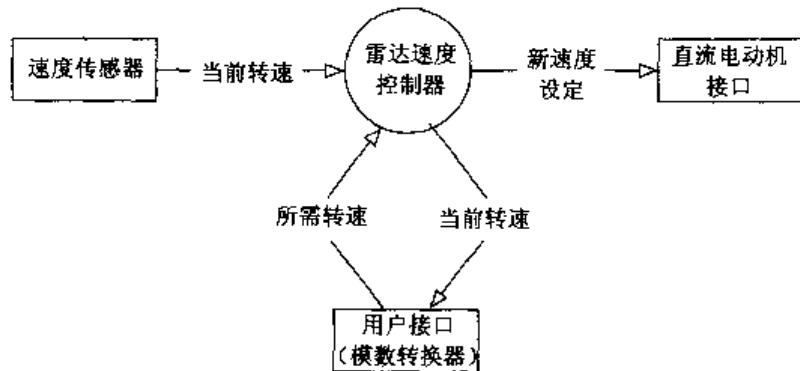


图 32.19 一个控制系统的初步设计

这类系统也用到规律的序列采样，如图 32.19 中所示的系统对电动机转速的采样。对这类控制系统，要用其他技术确定所需的采样频率，相关的技术将在第 35 章讨论。

滤除高频分量

如果以规则的频率 (F Hz) 对信号进行采样，通常需要在系统中用滤波器滤除所有高于 $F/2$ Hz 的频率，以避免混叠现象。

有关的详细情况参见 A-A 滤波器。

确定所需的采样频率

模拟数字转换的过程不可能是完美的，因为用来表示连续模拟信号是只能取有限个可取值的一系列数字（如图 32.20 所示）。

例如，如果用 3 位模数转换器，则只有 8 个可能的信号电平 (2^3) 表示模拟信号。数字化处理引入的误差为量化电平的一半。这样，对于 3 位模数转换器，误差将等于模拟电压信号范围的 $\pm 1/16$ 。对一个序列的采样而言，这些误差可以看成量化噪声的形式。

对于大多数的实际应用而言，12 位的模数转换器提供了足够的性能，即使是最复杂的语音处理系统也很少超过 16 位。

对于如何确定一般应用所需的采样频率已经超出了本书的范围：这个主题的介绍参见 Lynn 和 Fuerst (1998)，进一步的详细论述见 Oppenheim 等 (1999)。

软件体系结构

使用序列模数转换对软件体系结构的主要影响是必须规则和频繁地进行采样。

如果所需采样频率不超过 1kHz，一般没有什么问题。模数转换一般需要 100ns，本书介绍的调度系统能够支持这样的数据采集任务，同时系统也不会负载过重。

如果采样频率超过 1kHz，通常就需要使用快速的 8051 微控制器。例如，第 14 章中提到

的 Dallas 高速和超高速系列的微控制器，能够使用较短的时标间隔而不会使 CPU 过载。然而，即使系统能支持 10kHz 的采样频率，仍然有其他方面的重要设计约束——任务持续时间。在某些情况下，混合式调度器特别有用，因为数据采集可以配置为抢占式任务。

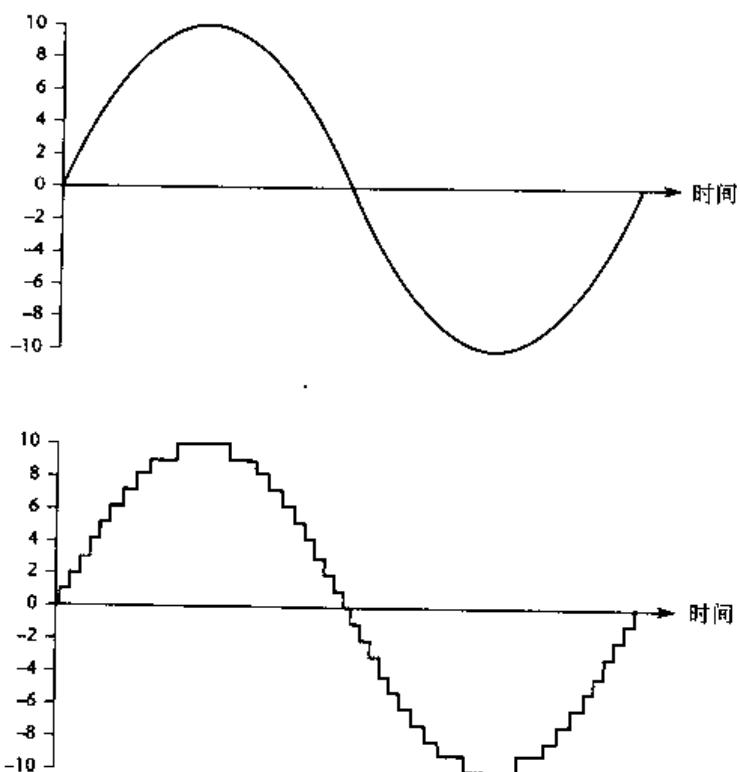


图 32.20 纯音的初始模拟形式（上图）和量化后的形式（下图）

选择合适的模数转换器

在单次模数转换模式中，讨论了一系列可用的模数转换器种类。更加详尽的资料请参看这个主题。

注意，和单次模数转换相比，序列模数转换对转换速度的要求高得多，确保所选择的模数转换器有高转换速度非常重要。这使得有些情况下，快闪模数转换器比逐次逼近模数转换器更适用。

硬件资源

使用片上模数转换器通常意味着至少要占用一个输入引脚。

使用片上模数转换器可能也会增加微控制器的功耗。

可靠性和安全性

如果其他条件等同，使用片上模数转换器很可能比使用外部模数转换器更可靠，这主要是因为片上解决方案的硬件复杂性比外部解决方案小得多。

另外，使用转换时间较长的模数转换器会增加延迟，这会降低信号处理系统的性能，可能还会影响控制系统的稳定性。要确保模数转换器的速度和系统相匹配。

可移植性

模数转换器各方面的特性各不相同。例如，为一个串行总线模数转换器编写的程序代码不经改写通常无法用在另一个串行总线模数转换器上。但是，其所需的修改一般较小。

优缺点小结

- ◎ 在许多系统中，序列模数转换都是必需的。
- ◎ 需要频繁进行采样对系统的体系结构有很大影响。

相关的模式和替代方案

相关的模式和替代方案参见本章的其余部分。

例子：C515C 内部模数转换器的应用

这个小型程序库说明了怎样利用 Infineon C515C 微控制器及其片上模数转换器显示模拟量的读数（源程序清单 32.10~源程序清单 32.12，并参见图 32.21）。



图 32.21 源程序清单 32.10~源程序清单 32.12 在 Keil 硬件模拟器上的输出

注意，这是 Keil C51 v5.5 硬件模拟器的输出。在本书写作的时候，Keil C51 v6.1（本书附带的 CD 上有）并不完全支持 Infineon 的微控制器。Keil 的网站^⑨上可能有产品更新。

先对模数转换器进行初始化。每当需要模拟读数时，启动模数转换并等待（方法是超时等待）转换完成。

每个模数转换任务所需的时间取决于片上模数转换器的转换速度。

```
/*
Port.H (v1.00)

项目 ADC_BAR (参见第 32 章) 的端口头文件 (参见第 10 章)
*/
// ----- ADC_515c.C -----
//从模数转换通道 0 (引脚 6.0) 读取
// ----- Bargraph.C -----
//将 LED 经过合适的电阻从 +5V 连至这些引脚
// [详见第 7 章]
//如果需要, 8 根端口引线可以分布在几个端口上
sbit Pin0 = P4^0;
sbit Pin1 = P4^1;
sbit Pin2 = P4^2;
sbit Pin3 = P4^3;
sbit Pin4 = P4^4;
sbit Pin5 = P4^5;
sbit Pin6 = P4^6;
sbit Pin7 = P4^7;
/*
----- 文件结束 -----
*/

```

源程序清单 32.10 Infineon C515C 微控制器片上模数转换器应用的例子的部分源程序

```
/*
Main.c (v1.00)

模数转换器 -> 柱状图显示的演示程序
所需的链接程序选项 (详情参见第 14 章):
OVERLAY
(main ~ (AD_Get_Sample,Bargraph_Update),
 sch_dispatch_tasks : (AD_Get_Sample,Bargraph_Update))
*/
#include "Main.h"
#include "Z_01_10i.h"
#include "ADC_515c.h"
#include "BarGraph.h"
/*
*/
void main(void)
{
    SCH_Init_T2(); // 设置调度器
    AD_Init(); // 设置模数转换器
    BARGRAPH_Init(); // 设置柱状图显示 (P4)
    // 规律地定期读取模数转换器
}
```

```

SCH_Add_Task(AD_Get_Sample, 10, 1000);
// 这里简单地显示计数 (柱状图显示)
SCH_Add_Task_(BARGRAPH_Update, 12, 1000);
// 所有的任务已加入; 启动调度器
SCH_Start();
While(1)
{
    SCH_Dispatch_Tasks();
}
/*
-----文件结束-----
*/

```

源程序清单 32.11 Infineon C515C 微控制器片上模数转换器应用的例子的部分源程序

```

/*
ADC_515c.c (v1.00)

简单的 C515c 单通道 8 位模数转换 (输入) 程序库
*/
#include "Main.h"
#include "Bargraph.h"
//-----公有变量定义-----
// 存储最新的模数转换读数
tByte Analog_G;
/*
AD_Init()
设置模拟-数字转换器
*/
void AD_Init(void)
{
    // 选择内部触发单次转换
    // 从 P6.0 (单通道) 读取
    ADEX = 0;      // 内部模/数转换触发
    ADM = 0;        // 单次转换
    MX2 = MX1 = MX0 = 0; // 从通道 0 (引脚 6.0) 读取
    // ADCON1 不变, 仍为复位值: 预分频为 /4
}

/*
AD_Get_Sample()
从 (10 位) 模数转换器得到单个采样数据 (8 位)。
*/
void AD_Get_Sample(void)
{
    tWord Time_out_loop = 1;
    // 从模数转换器取采样
    // 写 ADDATL (具体的数值无所谓) 以启动转换
    ADDATL = 0x01;
    // 从模数转换器取采样 (简单地用循环等待时间到)
}

```

```

while ((BSY == 1) && (Time_out_loop != 0));
{
    //      Time_out_loop++;    // 禁止, 为了使用 dScope...
}
if (!Time_out_loop)
{
    Analog_G = 0;
}
else
{
    // 已得到 10 位模数转换结果
    Analog_G = ADDATH;           // 只读出模数转换结果的高 8 位
}
/*
-----文件结束-----
*/

```

源程序清单 32.12 Infineon C51SC 微控制器片上模数转换器应用的例子的部分源程序

进阶阅读

Lynn, P. and Fuerst, W. (1998) *Introductory Digital Signal Processing with Computer Applications*, Wiley, Chichester.

Oppenheim, A.V., Schafer, R.W. and Buck, J.R. (1999) *Discrete-time Signal Processing*, Prentice Hall, New Jersey.

Smith, S.W. (1999) *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd edn, California Technical Publishing. [Available electronically at www.DSPguide.com]

A-A 滤波器

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何在信号数字化之前滤除高频分量？

背景知识

正如在序列模数转换一节所讨论的，数据采集系统的采样频率必须满足奈奎斯特准则；也就是说，采样频率 (Hz) 必须至少是想记录或者分析的最高频率 (Hz) 的两倍。例如语音记

录，对于用于工厂的语音识别系统，5kHz 的带宽就够了，因此采样频率定为 10 kHz（见图 32.22）。

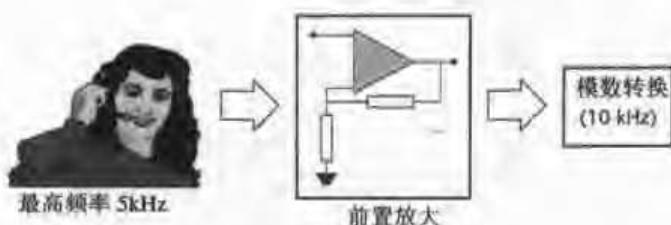


图 32.22 进行语音识别

然而，还有一个问题。在工厂的环境中，可能存在超过 5 kHz 限制的音频信号。语音信号本身最高会到 20kHz，其他环境中的音响甚至可能更高。所有话筒工作范围内的频率都在 5kHz 采样，将导致混叠现象。

这种情况下混叠会造成什么后果，先看一下图 32.23。图中所示的信号（纯音）的采样频率满足奈奎斯特准则。

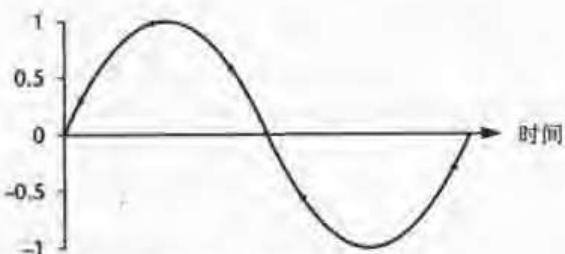


图 32.23 纯音信号的采样

图 32.24 则说明了混叠的影响。图中，实线是被采样的高频信号（频率大大高于奈奎斯特频率）。虚线是采样之后的信号表示。这张图说明，如果对频率高于 1/2 采样频率的高频信号进行采样，所得的原信号的数字表示将是错误的。

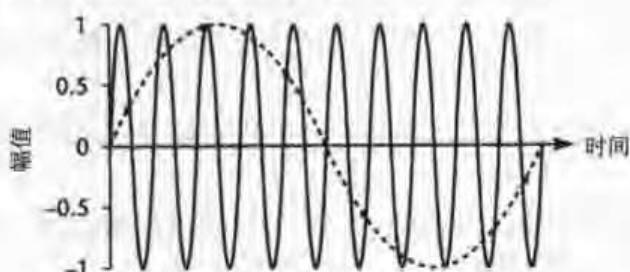


图 32.24 混叠的影响

重要的是要认识到，低频信号（图 32.23 中）和高频信号（图 32.24）采样后的数据表示完全相同；这两个信号在采样之后无法分辨，而且任何后续处理也无法逆转混叠了的数据。

解决信号混叠问题的惟一方法是在采样之前进行抗混叠滤波，抗混叠滤波器采用的是低通

滤波器的形式（如图 32.25 所示）。

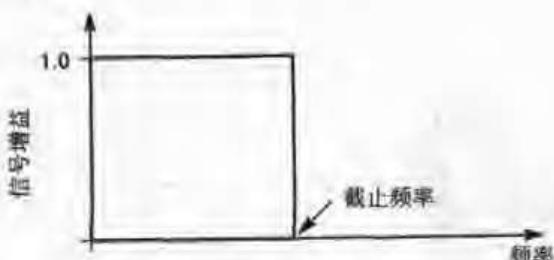


图 32.25 理想化的低通滤波器

广义上，可以认为滤波器的功能是：测量输入端上信号的各频率分量，改变输出端上各频率的幅值。被滤波器放大的输入信号频率范围（或者至少没有被衰减）被称为通带，而被滤波器抑制的输入信号频率范围被称为阻带——对低通滤波器（如图 32.25 所示），通带覆盖了较低的频率范围，而高频段被阻带覆盖。

如果在语音采集系统中加入抗混叠滤波器，那么其完整形式将如图 32.26 所示。

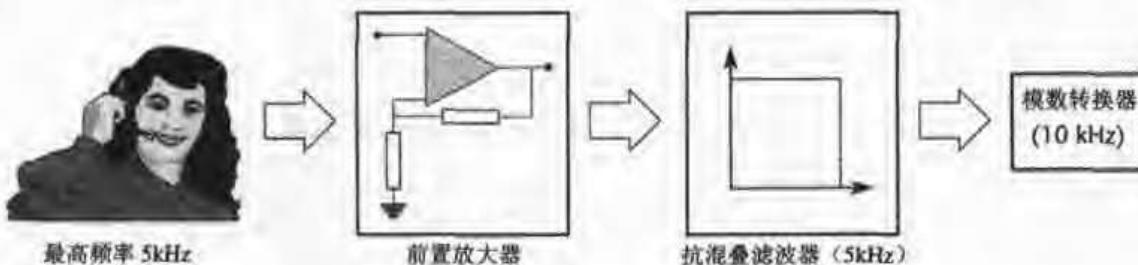


图 32.26 在语音识别系统中加入抗混叠滤波器

解决方案

在背景知识一节中讨论了抗混叠（A-A）滤波器的必要性。理论上，这种滤波器的设计和应用非常直截了当，只需要一个高品质的低通滤波器，其截止频率至多为希望采样的最高频率信号的频率的一半。

在实际系统中，图 32.25 中的理想化滤波器特性曲线是无法做到的，而图 32.27 的滤波器特性更能代表实际情况。

一般规律是，滤波器的特性越接近图 32.25 所示的理想滤波器，就越复杂、越昂贵。本节将考虑一些实际可行的 A-A 滤波器。

简单运放滤波器

模数转换前置放大器一节中考虑了模拟信号的预处理：信号放大和电压平移。滤波器特性曲线的设计将基于这个方法。图 32.28 说明了基本的方法，图 32.29 给出了大略的滤波器特性。根据图 32.29 所示， F_1 （赫兹）由 C （单位为法拉）和 R （单位为欧姆）按下式计算：

$$F_l = \frac{1}{2\pi C R_2}$$

低频段的通带增益为: $\frac{R_2}{R_1}$

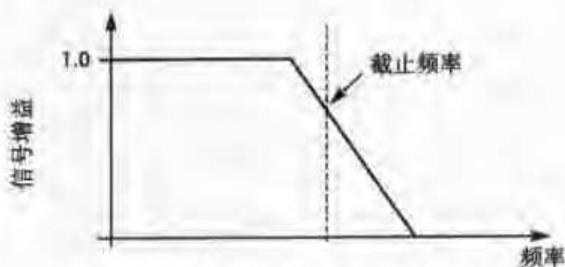


图 32.27 更实际的低通滤波器特性

注意，虽然此图提供了更实际的通带、阻带间的滤波器特性曲线，整个特性仍然是理想化的。

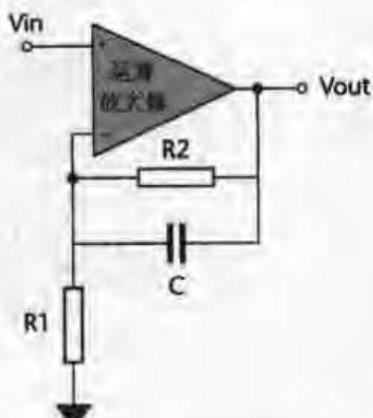


图 32.28 带增益的运放低通滤波器

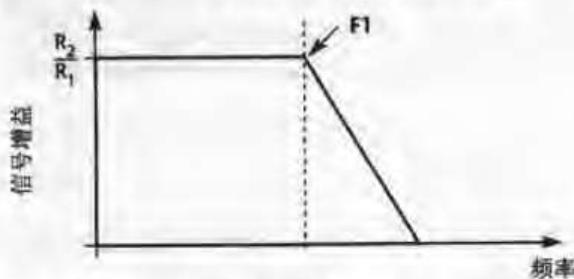


图 32.29 图 32.28 中滤波器的理想化特性

更高级的基于运放的设计

为了改善运放滤波器的性能，需要更复杂的多级式设计。但这种滤波器的设计已经超出了本书的范围。

然而注意，有多种滤波器设计软件能够帮助我们设计出合适的滤波器。例如，Microchip 的一个免费滤波器设计软件（FilterLab），可从 Microchip 的 WWW 网站^⑩下载，该软件可以用于设计合适的滤波器（图 32.30）。

开关电容滤波器 IC

在 A-A 滤波器设计中，一些开关电容滤波器得到了广泛地应用。这种滤波器的性能比上文提到的单运放滤波器高得多；虽然高级运放滤波器设计也能和这种滤波器相媲美（通常需要 10 个以上的运放），但是其体积庞大，而且需要小心地设计和制造。

开关电容滤波器由频率一般为截止频率 100 倍的外部时钟驱动。这样，对于 10kHz 的截止频率，所需的驱动时钟频率为 1MHz。这对于基于 8051 的设计很容易实现，关于生成此频率范围的信号而又几乎没有软件开销的技术，参见硬件脉冲频率调制。



图 32.30 用 Microchip 的软件 FilterLab 设计高品质的 A-A 滤波器
(承蒙 Microchip Technology Ltd 允许转载)

注意，用微控制器控制时钟频率意味着滤波器的截止频率能够通过改变时钟频率来控制。如果设计的应用需要改变采样频率，这个特性就非常有用了。

开关电容滤波器的主要缺点是会在经过滤波后的信号中引入时钟噪音。对于高要求的系统，开关电容滤波器一般和运放滤波器（就是在简单运放滤波器一节中讨论的运放滤波器）配合使用，以滤除开关噪声。

开关电容滤波器有各种各样的产品可用。例如，Linear Technology^①针对 A-A 应用设计的 1064 系列开关电容滤波器，使用广泛 Maxim^② Max7408（和类似的器件）也是一个可用的选择。注意，Maxim 的芯片一般为单电源供应，而 LT 的芯片通常需要正负电源供应。有关这些器件的详细情形请查阅制造商的数据手册。

连续时间滤波器 IC

也存在少数几种连续时间滤波器 IC。这类滤波器不用时钟驱动，因此不需要用于滤除时钟噪音的附加滤波器。Maxim 的 Max270 和 Max275 是其中的两个例子。

硬件资源

使用这个模式并不需要占用微控制器上的硬件资源（例如，CPU 的机时或者存储器）。但是显而易见，运算放大器和电阻将增加附加的成本。

此外，一些运算放大器电路可能需要正负电源供应（例如，+15V、-15V）。这会增加电源供应系统的设计复杂性，而且难以使用电池供电设计（或增加电池供电的成本）。现在，单电源的运算放大器越来越多，经常能够用在放大器设计中。

可靠性和安全性

无特别的可靠性或者安全性要求。

可移植性

这个模式只需要硬件，能和任何微控制器配合使用。

优缺点小结

- ⊕ 使用 A-A 滤波器能够大大地改善许多系统的性能。
- ⊗ 在设计中加入这种滤波器增加了产品的成本。

相关的模式和替代方案

高品质的 A-A 滤波器会显著地增加嵌入式系统的成本。有些情况下有可能降低使用 A-A 滤波器的必要性，或者只用简单的运放滤波器而不降低信号质量。一个可用的方法是对信号进行过采样。

再次考虑背景知识一节介绍的语音识别系统。为了分析最高可达 5kHz 的话音信号，该系统需要以 10kHz 的速度进行采样。然而，假定其他声音的频率会延伸到至少 20kHz，需要用高品质的 A-A 滤波器滤除超过 5kHz 的频率。

但是，如果采用以下的处理方式：

^① www.linear-tech.com

^② www.maxim-ic.com

- 用低品质的 5kHz 模拟 A-A 滤波器对信号进行滤波。
- 以 40kHz 的频率对信号进行采样（从而能够正确地对最高 20kHz 的信号进行采样）。
- 用软件对 40kHz 的信号进行数字低通滤波，滤除超过 5kHz 的频率。
- 丢弃每四个采样中的三个采样（四取一处理），得到所需要的 10kHz 数据。

这样处理能产生高品质的信号，无需在昂贵的模拟 A-A 滤波器上投资。这也是几乎所有的 CD 唱机厂商都使用过采样（一般 4 倍）技术的原因，这项技术能减少产品成本而又不牺牲品质。

数字滤波的操作简单明了（例如，参见 Lynn 和 Fuerst, 1998）。这种方法的主要缺点是需要高采样频率，这可能导致必须使用混合式调度器。

例子：在语音识别系统中应用 A-A 滤波器

考虑一下本模式前面讨论过的语音识别例子，图 32.31 为原理略图。

图 32.32 展示了该语音识别系统 A-A 滤波器的一个可能的设计，使用 Microchip FilterLab 软件设计。

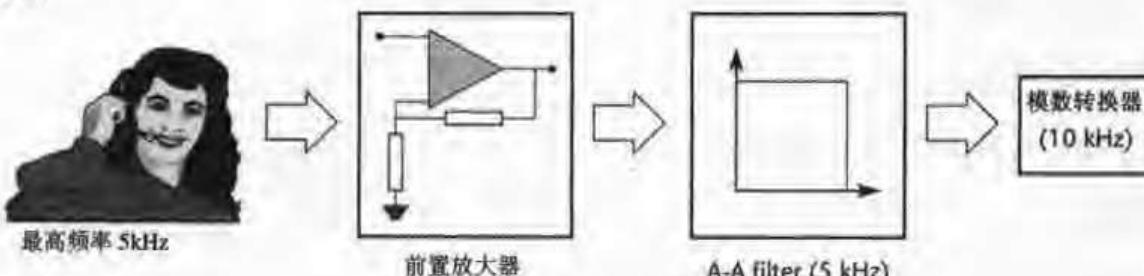


图 32.31 语音识别系统的回顾

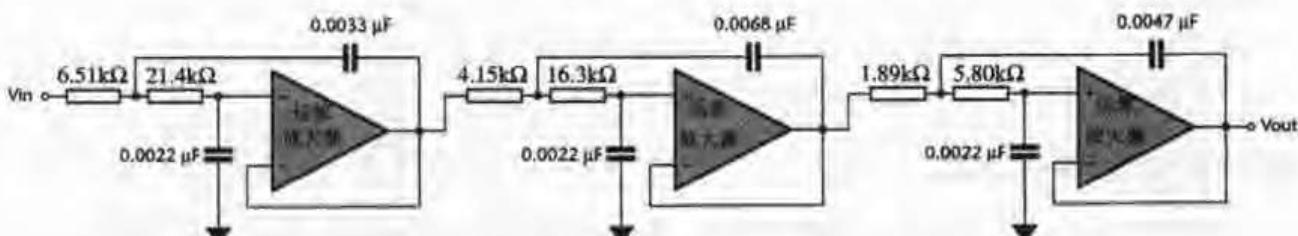


图 32.32 一个适用于语音识别系统例子的抗混叠滤波器

进阶阅读

Elgar, P. (1998) Sensors for Measurement and Control, Longman, London.

Franco, S. (1998) Design with Operational Amplifiers and Analog Integrated Circuits, 2nd edn, McGraw-Hill, Boston, MA.

Lynn, P. and Fuerst, W. (1998) Introductory Digital Signal Processing with Computer Applications, Wiley, Chichester.

电流传感器

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

怎样监测流过一个直流负载的电流？

背景知识

和表面的名称不同的是，电流传感器涉及模拟电压的测量技术，相关的背景资料参见单次模数转换和序列模数转换。

解决方案

本节将考虑电流传感问题的解决方案。

使用电流传感电阻器

传统电流传感技术的理论基础非常简单明了。例如，假设需要监测图 32.33 中流过负载的电流。

为了测量这个电流，可以用一个电阻与负载串联，并测量电阻上的电压降（如图 32.34 所示）。

流过负载的电流就能简单地由欧姆定律算出，如下式：

$$I_{\text{负载}} = \frac{V_{\text{负载}}}{R_{\text{负载}}}$$

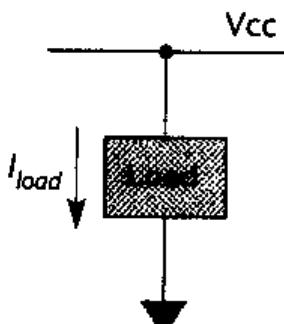


图 32.33 问题：需要测量流过负载的电流

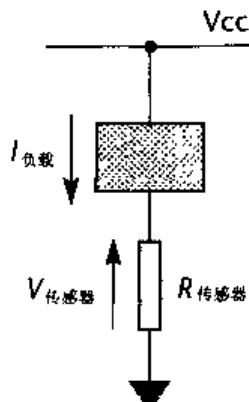


图 32.34 在电流传感应用中，用电阻作为电流-电压变换器

任何电阻都能用，但是在大多数情况下最好用阻值非常小的电阻。有这种专门为电流传感生产的电阻，一般使用的阻值小于 1Ω 。

为了说明为什么需要小电阻，假设有一个 $12V/1A$ 的负载（连接到 $12V$ 电源）；这是许多小灯泡或者直流电动机的典型负载。如果用一个 10Ω 的电阻与负载串联，电阻两端的电压降（由欧姆定律决定， $V = IR$ ）将高达 $10V$ 。而如果用一个 0.2Ω 电阻，电压降将降低为 $0.2V$ ：对大多数情况已经可以接受了。

注意，传感器电阻两端的电压必须足够大，以便用模数转换器测量该电压。为了监测的可靠性，通常需要大约 $0.1V$ 的电阻电压降。如果电压低于 $0.1V$ ，测量就可能受到电源波动或者电磁干扰影响。

注意，还必须确保传感器电阻有足够的额定功率。所需的电阻额定功率（以瓦特为单位），可用下式确定：

$$P_{\text{电阻}} = R_{\text{传感器}} (I_{\text{负载}})^2$$

因此，对于 0.5Ω 电阻和 $3A$ 负载，所需的额定功率为 $4.5W$ 。电流传感电阻有 $50W$ 和更高额定功率的。不过，电阻的额定功率通常是指电阻连在散热器上时的情况，如果不使用散热器，则选用的电阻额定功率至少要加倍。

不使用电阻的替代方案

在使用 MOSFET 或者 BJT 开关负载的微控制器系统中，经常能够不用单独的电流传感电阻就能测量电流。例如，见图 32.35。

BJT 饱和时发射极-集电极间的电压降约为 $1V$ 。因此通过测量发射极相对于地的电压，能够确定灯泡是否已经烧毁。如果电压为 $1V$ ，灯泡就是点亮的。注意，用这种方法并不能可靠地确定负载电流的大小，只能确定负载是否正被驱动。MOSFET 也有类似的解决方案，例如，见图 32.36。对图中的电路，如果 MOSFET 的导通电阻为 R_{on} ，则驱动负载时，MOSFET 漏极的电压为（根据欧姆定律）：

$$V_{\text{Drain}} = I_{\text{load}} R_{\text{on}}$$

例如,IRF540N的Ron是 0.055Ω ,所以对于2A左右的典型负载电流,漏极电压约为0.1V。这很容易测量,甚至用8位模数转换器就可以。

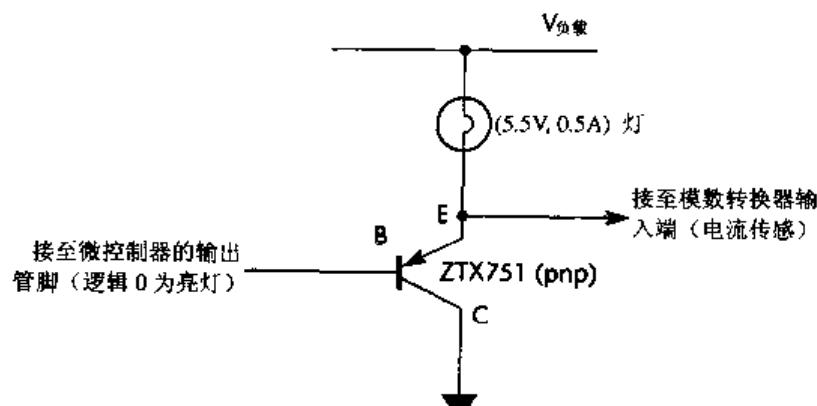


图32.35 不用电阻的电流传感1

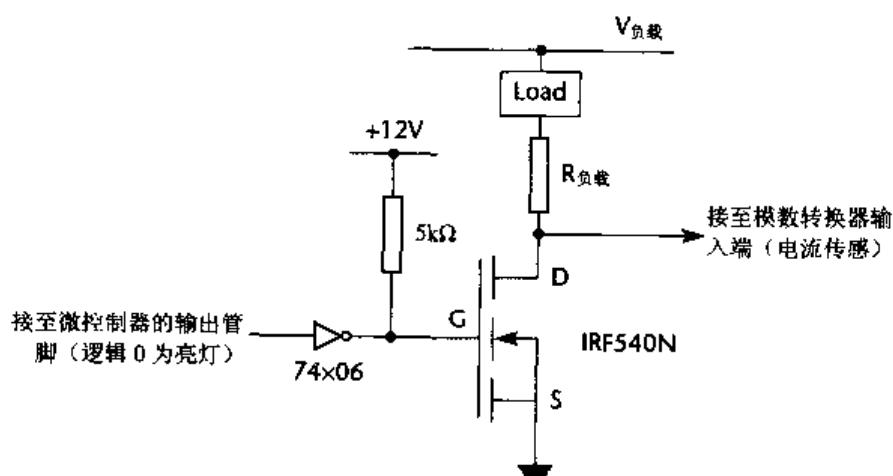


图32.36 不用电阻的电流传感2

硬件资源

应用此模式通常需要使用片上或者外部模数转换器或者比较电路。

可靠性和安全性

适当地应用这个技术能够监测负载的变化（例如，灯泡烧断，直流电动机堵转），从而改善系统的可靠性和安全性。

可移植性

这个主要依靠硬件的模式具有很高的可移植性。

优缺点小结

- ◎ 监测负载电流可以改善可靠性和安全性。
- ◎ 需要使用模数转换器或者类似的硬件。

相关的模式和替代方案

电流传感的替代方案参见单次模数转换。

例子：检测烧断的灯泡

作为安全防范系统的一部分，由一个微控制器系统控制一个 12V、20W 的灯泡（实际上是汽车的头灯灯泡），其基本方案如图 32.37 所示。

当灯泡烧断时，电压为 0；灯泡点亮时，能测量到 $\geq 0.1\text{V}$ 的电压。

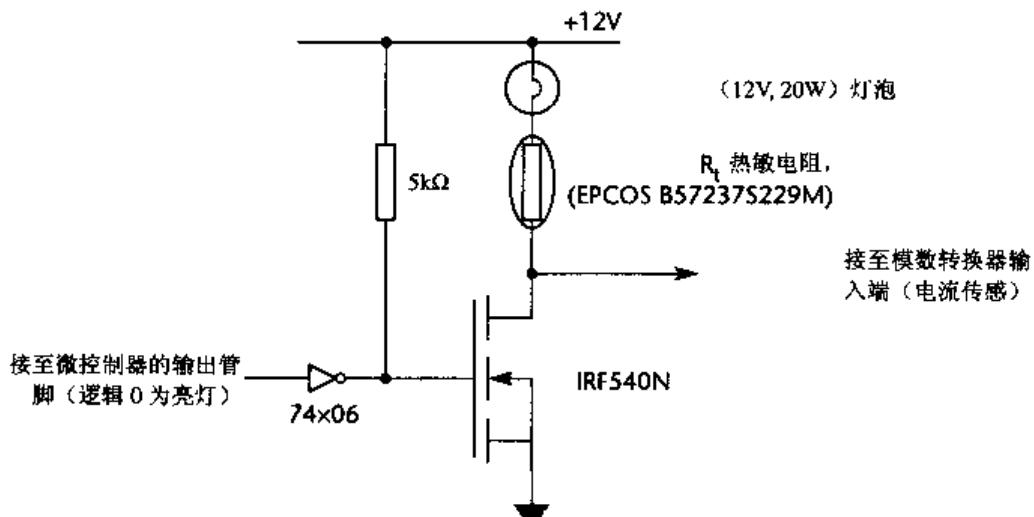


图 32.37 检测烧断的灯泡

进阶阅读

Chapter 33

脉冲宽度调制

引言

天黑后，在办公室或者家里，找一个有钨丝灯泡的房间。尽可能快地开关灯。灯光闪烁，房间也将变暗，这就是脉宽调制（PWM）的工作原理，不过其切换频率要高得多。更具体地说，脉宽调制能按一定的占空比开关灯，从而控制灯的亮度，同时也不会有可见的闪烁。

对于控制大功率负载，脉宽调制是一种有效的技术，特别是在直流（和交流）电动机转速控制等方面有广泛的应用。

对采用时间触发体系的系统，有两种实用的脉宽调制信号产生方法：

- 一些8051系列的微控制器提供了片上脉宽调制硬件支持，一般可以很容易地实现脉宽调制。如果没有片上脉宽调制硬件，则可以用专用的外部硬件产生高频率脉宽调制信号输出。
- 对于低频的脉宽调制信号，只用软件就可以生成。

这些方法将在硬件脉宽调制模式和软件脉宽调制模式中讨论。本章也将考虑脉宽调制信号的后续滤波处理，包括脉宽调制平滑滤波器模式和不用专门硬件的高频率三级脉宽调制模式。

硬件脉宽调制

适用场合

- 用8051系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何输出高频率的脉宽调制信号？

背景知识

脉冲宽度调制信号的许多特征和硬件脉冲频率调制输出中讨论过的脉冲频率调制信号相同。

类似脉冲频率调制，脉宽调制就是将一个引脚的电平在一个时段（X）设为逻辑 1，然后在另一个时段（Y）设为逻辑 0，并不断重复这一过程（如图 33.1 所示）。

端口引脚的平均电压取决于输出波形的占空比，占空比和脉宽调制的其他特征量定义如下：

$$\text{占空比 (\%)} = \frac{x}{x+y} \times 100$$

Period = $x + y$ ，此处 x 和 y 的单位是秒。

$$\text{Frequency (频率)} = \frac{1}{x+y} \text{，此处 x 和 y 的单位是秒。}$$

要注意的关键一点是，负荷的平均电压为占空比和工作电压的乘积。

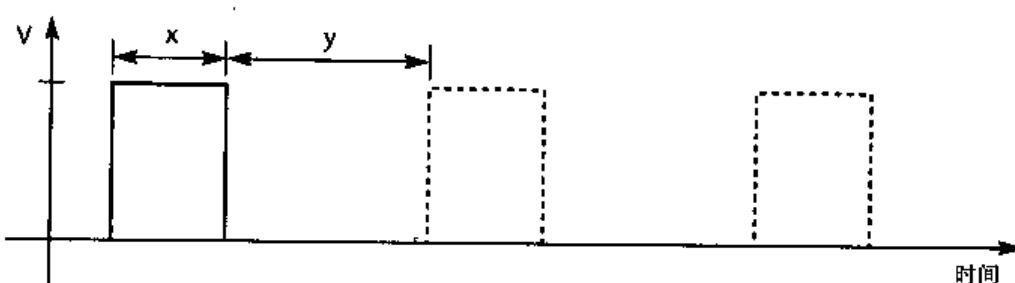


图 33.1 脉冲频率调制的基本定律

解决方案

本节将讨论以下几个方面的内容：

- 用片上硬件输出脉宽调制信号
- 用外部硬件输出脉宽调制信号
- 确定所需的切换频率
- 缓冲和驱动的限制
- 平滑脉宽调制输出信号

用片上硬件建立脉宽调制信号

本节将讨论用片上硬件产生脉宽调制信号的两个具体例子。

Infineon C515C

Infineon C515C 的脉宽调制定时器（基于定时器 2）每个机器时钟（振荡器频率的 1/6）加 1。对于最高工作频率下（10MHz）的 16 位脉宽调制信号，定时器每 300 纳秒（0.3 微秒）加

1。总的周期是 $2^{16} \times 0.3$ 皮秒，即 19 660.8 微秒或者 19.7ms，所以频率大约是 50Hz。

Dallas 87C550

Dallas 87C550 内建了一个高速 4 通道的脉宽调制硬件，最高脉宽调制频率（8 位脉宽调制、12MHz 振荡器）大约是 46kHz。

用外部硬件建立脉宽调制信号

如果微控制器没有硬件脉宽调制支持，可以使用外部脉宽调制芯片。例如，廉价的 Dallas DS1050 系列^①，可在 100kHz 频率下输出 5 位脉宽调制信号）。

确定所需的切换频率：一般的准则

理论上，确定系统所需的脉宽调制频率通常不容易：这取决于与负载和驱动有关的许多因素，通常需要进行一些实际测试，这些测试的具体情况将在后面讨论。

不管怎样，下面给出了一些基本准则：

- 人眼能够觉察最高 50Hz 的闪烁。如果脉宽调制硬件用于控制类似灯泡之类的电器设备，切换频率低于 50Hz 时，用户就有可能看到闪烁。
- 在出生时，人的听觉系统大约能听到 20Hz~20kHz 范围内的声音。如果在这个频率范围内切换大功率负载，很可能产生可闻的噪声，通常是一种非常讨厌的呜呜声。

确定所需的切换频率：应用时的考虑

对于一个实际系统，确定所需的切换频率的惟一办法是试验各种不同的频率。

有一种灵活的试验方法，用信号发生器驱动硬件并观察结果。

另一个经常能用于原型系统的有效方法是使用硬件脉冲频率调制输出模式。这种方法能够以固定占空比（50%）产生可变频率的方波，从不到 100Hz 直到 3MHz 以上。

缓冲和驱动的限制

脉宽调制硬件的速度当然不单和脉宽调制信号的生成硬件有关。例如，图 33.2 展示了一个脉宽调制交流电动机速度控制的尝试。

这个方法肯定会失败，因为电磁继电器的切换时间为 10 毫秒级，所以脉宽调制切换频率限制在大约最高 100Hz。对于很多系统，这个切换频率太低了。

所以在研发脉宽调制应用时，必须同时检查切换硬件和相关的缓冲电路的切换时间。

平滑脉宽调制输出信号

有时为了去掉高频谐波，需要对脉宽调制输出进行平滑滤波，详情参见脉宽调制平滑滤波器。

^① www.dalsemi.com

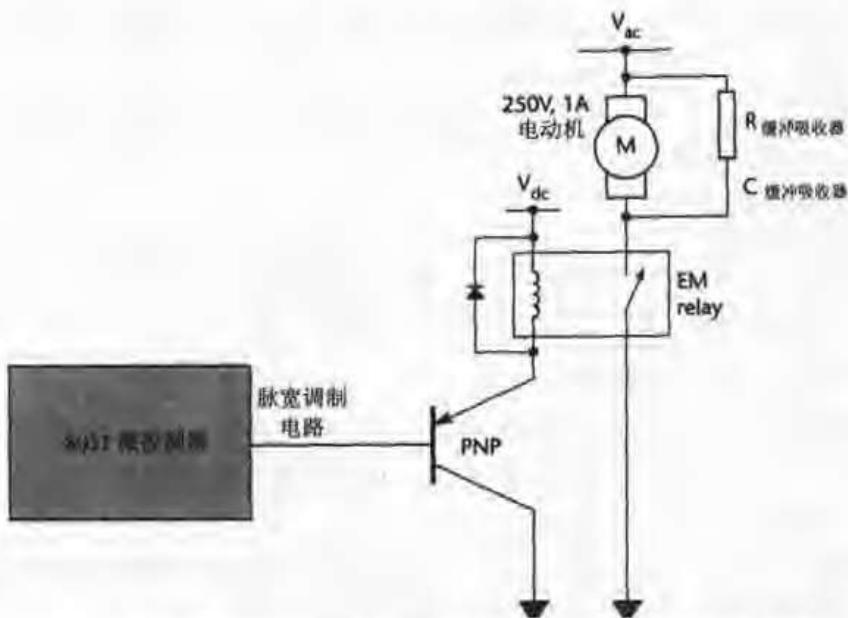


图 33.2 脉宽调制交流电动机速度控制的尝试，以电磁继电器驱动

注意，这个方法行不通，因为切换频率很低。

硬件资源

如果使用片上的脉宽调制硬件（通常基于定时器 2），主要的硬件资源是定时器 2，这个定时器也经常被用于驱动系统调度程序（在单处理器系统中）。

可靠性和安全性

脉宽调制常用于大功率负载，因此应照例需要确保系统处于安全的状态。例如，当微控制器复位时，相关的机器应低速启动，而非高速启动。这个问题进一步的讨论和可行的解决办法参见第 7 章和第 8 章的内容。

另一个问题是，高频率的脉宽调制信号是电磁干扰的一个来源，特别是容易干扰基于微控制器的嵌入式系统。对这个问题的更进一步的讨论请参见 Ong 等 (2001)。

可移植性

片上脉宽调制硬件的一个缺点是其具体实现在各种 8051 微控制器上各不相同。通常，外部脉宽调制硬件的程序代码更容易在 8051 微控制器间移植。

优缺点小结

- ◎ 多年来，脉宽调制一直被用作为慢速的外部组件（诸如交流和直流电动机、大容量的加热元件）提供连续变化的输出电压的有效方法。现在也仍然是这类系统的一个非常有效的控制方法，和数模转换器（参见数模转换器输出）不同，脉冲宽度调制

- 并不需要复杂的外部硬件。
- 脉宽调制可能形成电磁干扰。
- 并不是所有的 8051 微控制器都有脉宽调制的片上支持。
- 对带片上脉宽调制支持的微控制器，软件标准没有什么意义，因为专为某种 8051 微控制器写的脉宽调制程序代码不具有可移植性。

相关的模式和替代方案

参见软件脉宽调制。

参见脉宽调制平滑滤波器。

参见数模转换器输出。

例子：C515C 片上模数转换器和脉宽调制硬件的应用

在这个例子中，将用 C515C 的脉宽调制输出控制一个灯泡的亮度（源程序清单 33.1~源程序清单 33.4）。

对于交流或者直流灯泡而言，任何合适的硬件接口都可以使用，具体的方案参见第 7 章和第 8 章的内容。

注意，如果加电时慢慢地增加光强，灯丝承受的电流冲击将大大地减少，灯泡的寿命能够延长。脉宽调制电路能够很容易地做到这一点，这在更换损坏的灯泡很困难或者很昂贵时特别有用。

```
/*
-----*
Port.H (v1.00)
-----
项目 ADC_PWM (参见第 33 章) 的端口头文件 (参见第 10 章)
-----*/
// ----- ADC_515c.C -----
// 从引脚 6.0 读模数转换器
// ----- PWM_515c.C -----
// 引脚 1.1 上输出脉宽调制信号
----- 文件结束 -----
-----*/
```

源程序清单 33.1 说明 Infineon 515 系列微控制器片上脉宽调制硬件应用的例子的部分源码

```
/*
-----*
Main.c (v1.00)
-----
模数转换器->脉宽调制的简单程序例子 (c515c)
-----*/
#include "Main.h"
#include "ADC_515c.h"
#include "PWM_515c.h"
```

```

extern tByte Analog_G;
/*.....*/
/*.....*/
void main()
{
    AD_Init();
    PWM_Init_T2();
    while(1)
    {
        AD_Get_Sample();
        PWM_Update_T2(Analog_G);
    }
}
/*-----文件结束-----*/
/*-----*/

```

源程序清单 33.2 说明 Infineon 515 系列微控制晶片上脉宽调制硬件应用的例子的部分源码

```

/*-----*
 * PWM_515c.c (v1.00)
 *
 * 基本的脉宽调制的 80c515c 程序库。
 *-----*/
#include "Main.h"
#include "PWM_515c.h"
/*-----*
 * PWM_Init_T2()
 * 设置 c515c 的片内脉宽调制单元。
 *-----*/
void PWM_Init_T2(void)
{
    //---- T2 模式 -----
    // 方式 1 = 定时器功能
    // 预分频: Fcpu/6
    //---- T2 重新装载模式选择 -----
    // 方式 0 = 定时器溢出时自动重新加载
    // 将定时器寄存器预置为自动重装值: 0x0000:
    TL2 = 0x00;
    TH2 = 0xFF;
    //---- T2 比较模式 -----
    // 所有通道设为模式 0
    T2CON |= 0x11;
    // ----- T2 中断 -----
    // 禁止定时器 2 溢出中断
    ET2 = 0;
    // 禁止定时器 2 外部重新加载中断
    EXEN2 = 0;
    // ----- 比较/捕捉通道 0 -----
    // 禁止
}

```

```

// 比较寄存器 CRC 设置为: 0xFF00;
CRCL = 0x00;
CRCH = 0xFF;
// 禁止 CC0/ext3 中断
EX3 = 0;
// -----比较/捕捉通道 1 -----
// 使能比较
// 比较寄存器 CC1 设置为: 0xFF80;
CCL1 = 0x80;
CCH1 = 0xFF;
//禁止 CC1/ext4 中断
EX4=0;
// -----比较/捕捉通道 2 -----
// 禁止
// 比较寄存器 CC2 设置为: 0x0000;
CCL2 = 0x00;
CCH2 = 0x00;
// 禁止 CC2/ext5 中断
EX5 = 0;
// -----比较/捕捉通道 3 -----
// 禁止
// 比较寄存器 CC3 设置为: 0x0000;
CCL3 = 0x000;
CCH3 = 0x000;
// 禁止 CC3/ext6 中断
EX6=0;
// 设置上述所有模式, 通道 0~通道 3
CCEN = 0x08;
}

/*
-----*
PWM_Update_T2()
更新脉宽调制输出值 (捕捉/比较通道 1)
由引脚 1.1 输出。
注意: 如果没有软件介入, 硬件将一直生成这个值 (无穷地), 直到下次更新。
-----*/
void PWM_Update_T2(const_tByte New_PWM_value)
{
    CCL1 = New_PWM_value;
}
/*
-----文件结束 -----
-----*/

```

源程序清单 33.3 说明 Infineon 515 系列微控制芯片上脉宽调制硬件应用的例子的部分源码

```

/*
-----*
ADC_515c.c (v1.00)

简单的 C515c 单通道 8 位模数转换 (输入) 程序库
-----*/

```

```

#include "Main.H"
//#include "Bargraph.h"
-----公有变量定义-----
// 存储最新的模数转换读数
tByte Analog_G;
/*
AD_Init()
设置模拟-数字转换器。
*/
void AD_Init(void)
{
// 选择内部触发单次转换
// 从 P6.0 (单通道) 读取
ADEX = 0; // 内部模/数转换触发
ADM = 0; // 单次的转换
MX2 = MX1 = MX0 = 0; // 从通道 0 (引脚 6.0) 读取
// ADCON1 不变, 仍为复位值: 预分频为 /4
}
/*
AD_Get_Sample()
从 (10 位) 模数转换器得到单个采样数据 (8 位)。
*/
void AD_Get_Sample(void)
{
tWord Time_out_loop = 1;
// 从模数转换器取采样
// 写 ADDATL (具体的数值无所谓) 以启动转换
ADDAL = 0x01;
// 从模数转换器取采样 (简单地用循环等待时间到)
while ((BSY == 1) && (Time_out_loop != 0));
{
// Time_out_loop++; // 禁止, 为了使用 dScope...
}
if (!Time_out_loop)
{
Analog_G = 0;
}
else
{
// 已得到 10 位模数转换结果
Analog_G = ADDATH; // 只读出模数转换结果的高 8 位
}
}
/*
---文件结束---
*/

```

源程序清单 30.4 说明 Infineon 515 系列微控制器片上系统调试硬件应用的例子的部分源码

进阶阅读

Ong, H.L.R, Font, M.J. and Peasgood, W. (2001) 'Do software-based techniques increase the reliability of embedded applications in the presence of EMI?', *Microprocessors and Microsystems*, 24 (10): 481-91.

脉宽调制信号平滑滤波

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

怎样对脉宽调制信号发生器的输出进行滤波？

背景知识

相关的背景信息参见硬件脉宽调制和 A-A 滤波器。

解决方案

首先讨论一下，为什么需要对脉宽调制信号进行平滑滤波。然后再讨论怎样进行滤波。最后，考虑一些可能不需要脉宽调制滤波器的情况。

为什么需要对脉宽调制信号进行平滑滤波？

书中讨论的脉宽调制信号的频率都是保持不变的，而占空比按照原始信号的振幅在 0% 到 100% 间变化，这一信号的时域表示如图 33.3 所示。

如果考虑图 33.3 中信号的频域表示，很明显在 $1/T$ 频率上有一个很强的峰值，这是脉宽调制信号的基本部分。除此之外，在频率 N/T 处（ N 是整数），存在谐波峰值（如图 33.4 所示）。

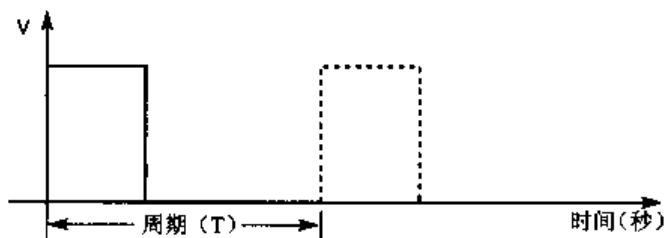


图 33.3 脉宽调制信号的时域表示

脉宽调制信号的基本部分不能滤除，所以必须保证选择合适的频率（例如，在听觉范围之

外)。然而, 谱波分量在我们所关心的应用范围内, 是无用的噪声来源, 通常必须用合适的滤波器滤除。

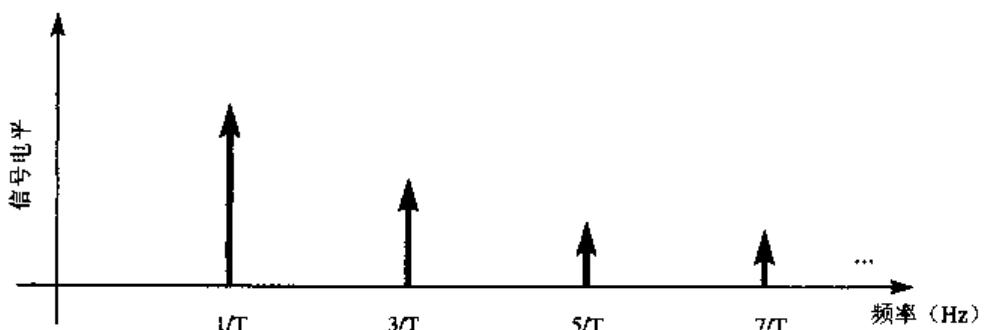


图 33.4 脉宽调制信号的频域表示

注意, 信号频率表示可以用频谱分析仪得到。

怎样滤除脉宽调制信号的噪音?

为了滤除脉宽调制信号的噪音, 滤波器需要具有类似 A-A 滤波器的特性: 也就是说, 理想的脉宽调制平滑滤波器应具有“砖墙”特性和 $1/T$ 的截止频率(如图 33.5 所示)。

正如在 A-A 滤波器的章节中讨论的, 这种特性的滤波器对大多数的系统在经济上是不可行的, 而图 33.6 中的设计在大多数的情况下是够用的。

实现细节参见 A-A 滤波器。

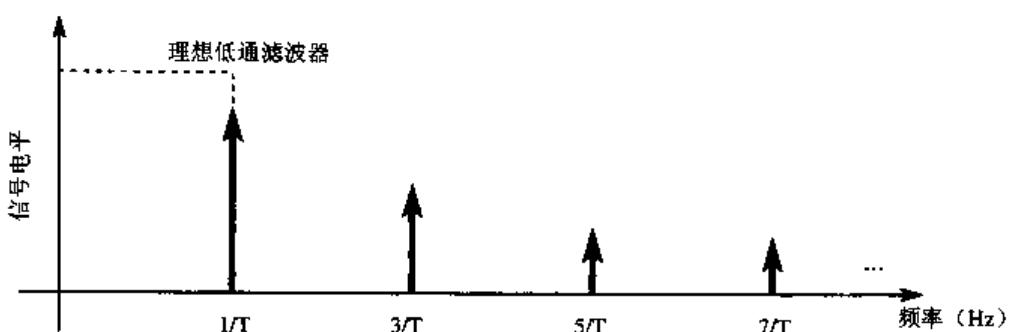


图 33.5 用理想低通滤波器滤除脉宽调制输出的噪音

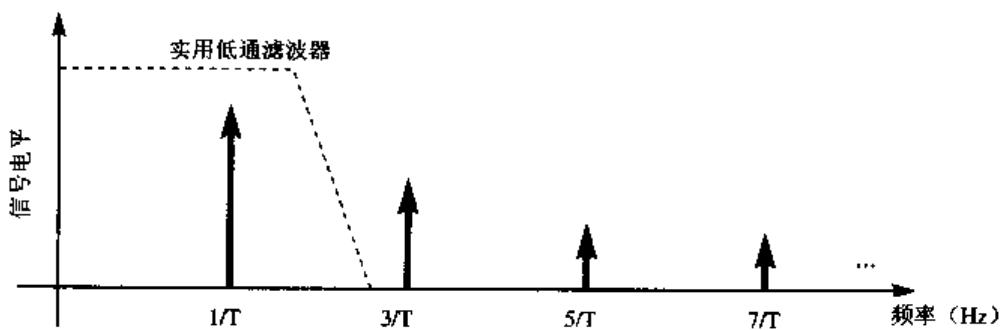


图 33.6 用实用低通滤波器滤除脉宽调制输出的噪音

脉宽调制滤波器是必需的吗？

并不是所有的系统都需要脉宽调制滤波器。例如，在电动机控制（一个很常见的应用领域）中，脉宽调制滤波器通常是不必要的。实际上，需要滤波器的主要领域是音频应用，例如语音生成。但是注意，即使在音频应用中，如果使用高脉宽调制频率，也能够降低使用滤波器的必要性，例如使用 100kHz 的基频，谐波分量将超过人的听觉范围（但对另外的一些物种并非如此）。

硬件资源

这个主要依靠硬件的模式没有占用特别的微控制器硬件资源。

可靠性和安全性

使用合适的平滑滤波能够显著地减少脉宽调制硬件产生的高频电磁干扰的水平，这一点对可靠性可能很重要。

可移植性

这个主要依靠硬件的模式具有很高的可移植性。

优缺点小结

- ☺ 减少脉宽调制输出的音频和电磁干扰。
- ⊗ 增加了系统的成本。

相关的模式和替代方案

参见 A-A 滤波器。

例子：建立一个 5kHz 的脉宽调制平滑滤波器

合适的滤波器的实现细节参见 A-A 滤波器中的“语音识别”例子。

进阶阅读

Lynn, P. and Fuerst, W. (1998) Introductory Digital Signal Processing with Computer Applications, Wiley, Chichester.

Oppenheim, A.V., Schafer, R.W. and Buck, J.R. (1999) Discrete-time Signal Processing, Prentice-Hall NJ.

Palachcra, A. (1997) 'Using PWM to generate analog output', Microchip Application Note AN538.

3 级脉宽调制

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何不用专用硬件输出高频率的脉宽调制信号？

背景知识

参见硬件脉冲频率调制输出中有关脉冲频率调制的讨论。

解决方案

假设需要控制一个直流电动机的转速。如果通过微控制器的输出引脚以较慢的速度开关电动机，可以视为 1 位脉宽调制或者 2 级脉宽调制。也就是说，用零脉冲宽度或者无穷大脉冲宽度控制电动机。

这个方法能进一步改进。正如在硬件脉冲频率调制输出看到的，大多数现代 8051 微控制器包含定时器 2，这个定时器能够编程产生一个频率非常高（最高 3MHz，即使是对 12MHz/12 振荡器的 8051 微控制器）的 50% 占空比的输出信号。

使用这个特性，再加上刚提到的开、关控制，能用最小的软件负载做到：

- 全速运转电动机。
- 用高频率的脉宽调制输出半速运转电动机。
- 停止电动机的运转。

这些对许多系统已经足够了，而且实现成本很低。当然，也可同样用于其他的外部设备，诸如交流或者直流照明。

硬件资源

此模式需要利用定时器 2。在单处理器系统中，定时器 2 通常是最适合作调度器驱动的定时器，因而使用这种技术可能会影响系统其他部分。

注意，如果可以采用双处理器设计（比如用基于 UART 或者基于中断的调度器），那么可以用主节点的定时器 2 驱动调度器，而把从节点的定时器留着用于脉宽调制信号发生器。可能适用的各种多处理器设计详见第 6 篇。

可靠性和安全性

这种技术非常可靠。

可移植性

这一技术只能用于基于 8052 的微控制器，也就是说，微控制器必须内建定时器 2。虽然多数较新的 8051 系列微控制器都有定时器 2，但某些流行的 8051 系列微控制器（如 Atmel 的小 8051）没有定时器 2。

优缺点小结

- ⊕ 一个输出宽频带脉冲频率调制信号的简单、有效的技术。
- ⊕ 几乎没有存储器或者 CPU 负载。
- ⊖ 需要占用定时器 2。

相关的模式和替代方案

比较接近的模式是硬件脉宽调制模式和软件脉宽调制模式。

在硬件资源一节中也提到过，可以在多处理器设计的从节点上使用这一技术。各种多处理器模式详见第 6 篇。

例子：菜单驱动 3 级脉宽调制的例子

用如图 33.7 所示的硬件控制小灯泡的亮度。

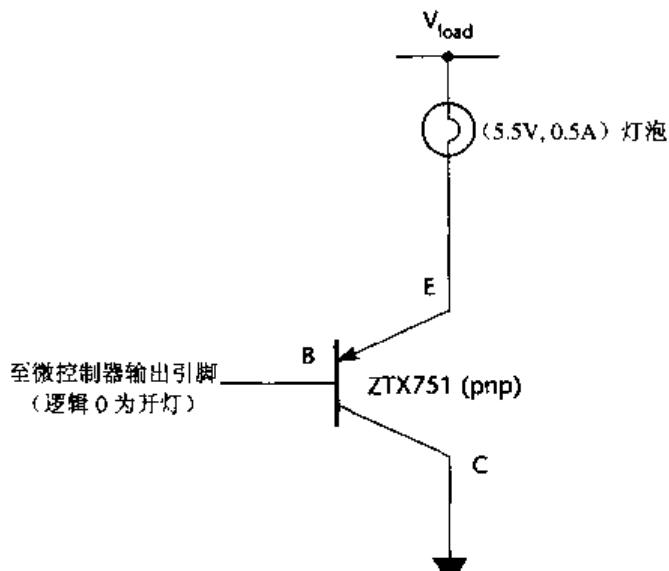


图 33.7 用 BJT (双结型晶体管) 驱动器控制一个小灯泡

源程序清单 33.5~源程序清单 33.7 给出了一个合适的 3 级软件脉宽调制的小型程序库。图 33.8 展示了例子程序在 Keil 硬件模拟器上运行的情况。



图 33.8 3 级脉宽调制程序库在 Keil 硬件模拟器上的运行演示

```
/*
  Port.H (v1.00)
  项目 PWM_3lev (参见第 33 章) 的端口头文件 (参见第 10 章)
*/
// ----- Sch51.C -----
// 如果不需要错误报告, 注释掉此行。
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P2
#endif
// ----- PC_IO.C -----
// 引脚 3.0 和引脚 3.1 用作 RS-232 接口
// ----- 3_PWM.C -----
sbit PWM_pin = P1^0;
/*-----文件结束-----*/
*/
```

源程序清单 33.5 不用专门硬件生成 3 级脉宽调制信号的例子的部分源代码

```
/*
  Main.c (v1.00)
  菜单驱动 3 级脉宽调制程序库的测试程序。
  链接选项:
  OVERLAY (main ~ (PWM_3_Command_Processor),
  SCH_Dispatch_Tasks ! (PWM_3_Command_Processor))
*/
#include "Main.h"
#include "0_05_llg.h"
#include "PC_IO_T1.h"
```

```
#include "PWM_3.h"
/*
 */
void main(void)
{
    // 设置调度器
    SCH_Init_T0();
    // 波特率设置为 9600: 对一般的 8051 微控制器
    PC_LINK_IO_Init_T1(9600);
    // 必须调度此任务 (每秒 10x - 100x)
    //
    // 时间单位是时标间隔数而非毫秒 (5ms 时标间隔)
    SCH_Add_Task(PWM_3_Command_Processor, 10, 2);
    SCH_Start();
    while(1)
    {
        // 在 P4 上显示错误代码 (参见 Sch51.C)
        SCH_Dispatch_Tasks();
    }
}
/*-----文件结束-----*/

```

源程序清单 33.6 不用专门硬件生成 3 级脉宽调制信号的例子的部分源代码

```
/*
 PWM_3.C (v1.00)

简单的 3 级脉宽调制例子 (参见第 33 章)
使用超级终端 (在 Windows 95、98、2000 等操作系统上) ,
或者其他操作系统上的类似终端仿真程序。
终端选项:
-数据位 = 8
-奇偶校验=无
-停止位 = 1
-流量控制 = Xon / Xoff
*/
#include "Main.h"
#include "Port.h"
#include "0_05_11g.h"
#include "PWM_3.h"
#include "PC_IO_T1.h"
//-----私有常数-----
#define PWM_OFF 1
#define PWM_ON 0
/*
PWM_3_Command_Processor()
此函数是主菜单的命令处理函数
大约每 10ms 调度一次

```

第7篇 监视与控制组件

```
-----*/
void PWM_3_Command_Processor(void)
{
    static bit First_time_only;
    char Ch;
    if (First_time_only == 0)
    {
        First_time_only = 1;
        PWM_3_Show_Menu();
    }
    // 检查用户输入
    PC_LINK_Update();
    Ch = PC_LINK_Get_Char_From_Buffer();
    if (Ch != PC_LINK_NO_CHAR)
    {
        PWM_3_Perform_Task(Ch);
        PWM_3_Show_Menu();
    }
}
/*-----*/
PWM_3_Show_Menu()
在PC屏幕上显示菜单选项（通过串行口）
-根据具体应用的需要编写
-----*/
void PWM_3_Show_Menu(void)
{
    PC_LINK_Write_String_To_Buffer("Menu:\n");
    PC_LINK_Write_String_To_Buffer("a - 0%\n");
    PC_LINK_Write_String_To_Buffer("b - 50%\n");
    PC_LINK_Write_String_To_Buffer("c - 100%\n\n");
    PC_LINK_Write_String_To_Buffer("? : ");
}
/*-----*/
PWM_3_Perform_Task()
执行所需的用户任务
-根据具体应用的需要编写
-----*/
void PWM_3_Perform_Task(char c)
{
    // 回显菜单选项
    PC_LINK_Write_Char_To_Buffer(c);
    PC_LINK_Write_Char_To_Buffer('\n');
    // 执行任务
    switch (c)
    {
        case 'a':
        case 'A':
            {
                PWM_3_Set_000();
            }
    }
}
```

```
        break;
    }
case 'b':
case 'B':
{
    PWM_3_Set_050();
    break;
}
case 'c':
case 'C':
{
    PWM_3_Set_100();
}
}

/*-----*
 * PWM_3_Set_000()
 * 将脉宽调制输出的占空比设置为 0%
 *-----*/
void PWM_3_Set_000(void)
{
    PC_LINK_Write_String_To_Buffer("\n*** 0% ***\n\n");
    TR2 = 0; // 停止定时器 2
    PWM_pin = PWM_OFF;
}

/*-----*
 * PWM_3_Set_050()
 * 用定时器 2 将脉宽调制输出的占空比设置到 50%
 *-----*/
void PWM_3_Set_050(void)
{
    PC_LINK_Write_String_To_Buffer("\n*** 50% ***\n\n");
    T2CON &= 0xFD; // 只清除 C/T2 位
    T2MOD |= 0x02; // 设置 T2OE 位 (对基本的 8052 仿制微控制器可省略)
    // 设定为最低频率 (对 12MHz 晶振约为 45Hz)
    // -根据需要调整 (参见硬件脉冲频率调制)
    TL2 = 0x00; // 定时器 2 低字节
    TH2 = 0x00; // 定时器 2 高字节
    RCAP2L = 0x00; // 定时器 2 重加载捕捉寄存器, 低字节
    RCAP2H = 0x00; // 定时器 2 重加载捕捉寄存器, 高字节
    ET2 = 0; // 没有中断
    TR2 = 1; // 启动定时器 2
}

/*-----*
 * PWM_3_Set_100()
 * 将脉宽调制输出的占空比设置为 100%
 *-----*/
void PWM_3_Set_100(void)
{
```

```

PC_LINK_Write_String_To_Buffer("\n*** Doing C ***\n\n");
TR2 = 0; // 停止定时器 2
PWM_pin = PWM_ON;
}
/*
-----文件结束-----
*/

```

源程序清单 33.7 不用专门硬件生成 3 级脉宽调制信号的例子的部分源代码

进阶阅读

Huang, H-W (2000) *Using the MCS-51 Microcontroller*, Oxford University Press, New York.

软件脉宽调制

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

如何不使用专门硬件便能输出低频的脉宽调制信号？

背景知识

脉宽调制的一般背景材料参见硬件脉宽调制。软件脉冲频率调制中给出了一个相似的解决方案。

解决方案

在软件脉冲频率调制中讨论的技术可以很容易地更改用于产生低频的脉宽调制信号，源程序清单 33.8 示范了如何更改。

```

void PWM_Soft_Update(void)
{
    // 到了当前脉宽调制周期末了吗？
    if (++PWM_positon_G >= PWM_PERIOD)
    {
        // 复位脉宽调制位置计数器
        PWM_positon_G = 0;
        // 更新脉宽调制控制值
        PWM_G = PWM_new_G;
        // 设置脉宽调制输出为关闭
        PWM_pin = PWM_OFF;
    }
}

```

```

    return;
}
// 在脉宽调制周期中
if (PWM_position_G < PWM_G)
{
    PWM_pin = PWM;
}
else
{
    PWM_pin = PWM_OFF;
}
}

```

源程序清单 33.8 实现软件脉宽调制

源程序清单 33.8 的关键是变量 PWM_position_G、PWM_period_G 和 PWM_period_new_G：

- PWM_period_G 是当前脉宽调制周期。注意，如果更新函数每毫秒调度一次，该周期就是以毫秒计的。在程序执行期间 PWM_period_G 是固定不变的。
- PWM_G 表示当前脉宽调制信号的占空比（参见图 33.9）。
- PWM_new_G 是下一个周期脉宽调制信号的占空比。这个周期可由用户根据需要改变。注意，新的值只在脉宽调制周期的末端复制到 PWM_G，以避免噪音。
- PWM_position_G 是脉宽调制周期的当前位置，由更新函数增加。同样，如果更新函数每毫秒调度一次，变量的单位则为毫秒。

各变量和常数间的关系见图 33.9。

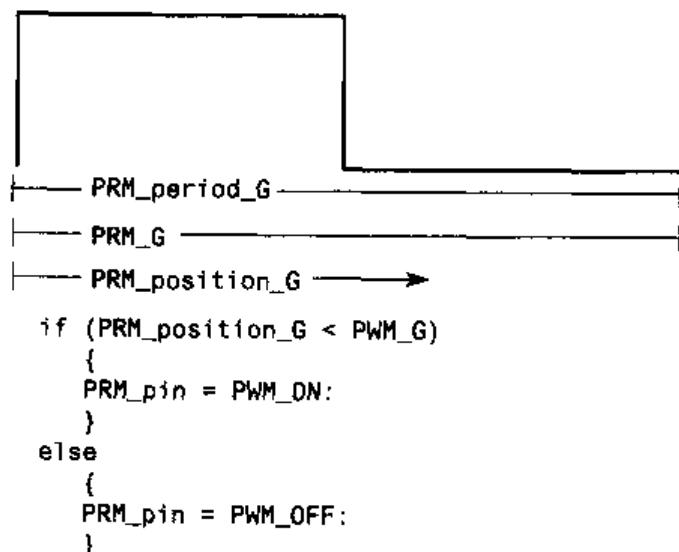


图 33.9 脉宽调制的软件实现

使用 1ms 的调度器，所得的脉宽调制频率 (Hz) 和脉宽调制分辨率 (%) 如下式：

$$\text{频率}_{\text{PWM}} = \frac{1000}{2^N}$$

和

$$\text{分辨率}_{\text{PWM}} = \frac{1}{2^N} \times 100$$

此处， N 是所用的脉宽调制位数。

例如，在 31Hz 的频率下，5 位脉宽调制可以达到大约 3% 的输出分辨率。

硬件资源

软件脉宽调制没有使用重要的硬件资源。

可靠性和安全性

当设计脉宽调制电路时，可能需要切换大功率负载或强电电源。所以需要注意启动时的状态，确保系统在启动时，所控制的任何设备都不会造成设备损坏或人员受伤。

可移植性

因为软件脉宽调制只用到了 8051 的核心特性，所以很容易应用于 8051 系列的各种微控制器。

优缺点小结

- ① 多年来，脉宽调制一直被用作为慢速的外部组件（诸如交流和直流电动机、大容量的加热元件）提供连续变化的输出电压的有效方法。现在也仍然是这类系统的一个非常有效的控制方法，而且无需数模转换器。
- ② 软件脉宽调制对 CPU 并不是很重的负载。
- ③ 软件脉宽调制只能在相对较低的频率下运作。例如，对于 1ms 的时标间隔和 8 位脉宽调制，脉宽调制的频率是 1000/256，即 4 Hz。典型的硬件脉宽调制单元可在 100kHz 下工作。对于许多系统，高速的脉宽调制是一个重要的因素，关于这个问题的更详尽资料请参见硬件脉宽调制。

相关的模式和替代方案

硬件脉宽调制可作为一个替代方案。

本模式的另一个替代实现方案也值得介绍一下，看一下源程序清单 33.9 中的程序。

```
void Fast_Software_PWM_Update(void)
{
    tWord PWM_position;
    for (PWM_position = 0; PWM_position < PWM_INCREMENTS; PWM_position++)
    {
```

```

if (PWM_position < PWM_G)
{
    PWM_pin = PWM_ON;
}
else
{
    PWM_pin = PWM_OFF;
}

// 在这里适当地延迟
PWM_Delay();
}
}

```

源程序清单 33.9 增加软件脉宽调制信号的频率

在这种解决方案中，每次调用任务时，生成整个脉宽调制周期，这样看起来可以产生更高频率的脉宽调制信号。例如，在 0.25ms 的时标间隔下，能够产生 4kHz 的脉宽调制频率，这对许多系统已经足够高了。此外，还能够生成任意数值的 PWM_INCREMENTS，也就是说，任意需要的位数。

然而，这种解决方案有两个问题。第一个问题是 CPU 忙于脉宽调制信号的生成。这并非不可克服，但可能意味着，这种方案只能在多处理器系统的从节点上实现。

第二个问题更加严重，涉及到循环内部所需的延迟。例如，在1ms时标间隔下，5位脉宽调制输出的产生需要精确的 $3.125\mu s$ 延迟。正如在第11章讨论过的，这很难达到。对于更高的位数和定时频率，所需的延迟更短，在大多数的实现中都无法做到。

例子：控制灯泡的亮度

用如图 33.10 所示的硬件控制小灯泡的亮度。

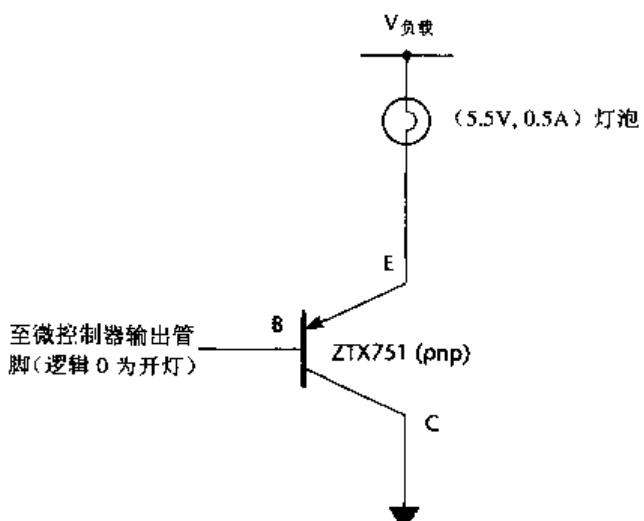


图 33.10 控制灯泡的亮度

源程序清单 33.10~源程序清单 33.12 给出了一个软件脉宽调制的小型程序库。

```
/*
  Port.h (v1.00)
  -----
  项目 PWM_Soft (参见第 33 章) 的端口头文件 (参见第 10 章)
  -----
// ----- Sch51.C -----
// 如果不需要错误报告, 注释掉此行
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P1
#endif
// ----- PWM_Soft.C -----
sbit PWM_pin = P1^0;
/*
  ----- 文件结束 -----
*/
```

源程序清单 33.10 不用专门硬件生成脉宽调制信号的例子的部分源代码

```
/*
  Main.C (v1.00)
  -----
  软件脉宽调制程序库的测试程序。
  详见第 33 章。
  -----
#include "Main.h"
#include "2_01_12g.h"
#include "PWM_Soft.h"
/*
  -----
/*
void main()
{
    SCH_Init_T2();
    PWM_Soft_Init();
    // 每毫秒调用一次以更新脉宽调制输出
    SCH_Add_Task(PWM_Soft_Update, 0, 1);
    // 每 5s 调用一次, 改变脉宽调制控制值
    SCH_Add_Task(PWM_Soft_Test, 10, 3000);
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
```

```

    }
}

/*-----*
---文件结束 -----
*-----*/

```

源程序清单 33.11 不用专门硬件生成脉宽调制信号的例子的部分源代码

```

/*-----*
  PWM_SOFT.C (v1.00)
-----*
  简单的软件脉宽调制程序库。
  详见第 32 章。
*-----*/
#include "Main.h"
#include "Port.h"
#include "2_01_12g.h"
#include "PWM_Soft.h"
//-----公有变量定义-----
// 将该变量设置为所需的脉宽调制值
tByte PWM_new_G;
//-----私有常数-----
#define PWM_ON 0
#define PWM_OFF 1
// -----私有变量 -----
// 脉宽调制计数器
static tWord PWM_position_G;
static tByte PWM_G;
/*-----*
  PWM_Soft_Test()
  为了测试脉宽调制程序库，每五秒调用此函数一次，以更新脉宽调制设置
  注意：
  在实际系统中，此函数由一个用户定义的函数代替，比如设置亮度、速度等等
*-----*/
void PWM_Soft_Test(void)
{
    if (++PWM_new_G >= PWM_PERIOD)
    {
        PWM_new_G = 0;
    }
}
/*-----*
  PWM_Soft_Init()
  准备一些关键的脉宽调制变量。
*-----*/
void PWM_Soft_Init(void)
{
    // 初始化主变量
    PWM_new_G = 0;
}

```

```
PWM_position_G = 0;
PWM_pin = PWM_OFF;
}
/*
  PWM_Soft_Update()
  脉宽调制的关键函数，尽可能快地调度。
*/
void PWM_Soft_Update(void)
{
    // 到了当前脉宽调制周期末了吗？
    if (++PWM_position_G >= PWM_PERIOD)
    {
        // 复位脉宽调制位置计数器
        PWM_position_G = 0;
        // 更新脉宽调制控制值
        PWM_G = PWM_new_G;
        // 设置脉宽调制输出为关闭
        PWM_pin = PWM_OFF;
        return;
    }
    // 在脉宽调制周期中
    if (PWM_position_G < PWM_G)
    {
        PWM_pin = PWM_ON;
    }
    else
    {
        PWM_pin = PWM_OFF;
    }
}
/*
  ---文件结束---
*/

```

源程序清单 33.12 不用专门硬件生成脉宽调制信号的例子的部分源代码

进阶阅读

Chapter 34

数模转换器的应用 (DAC)

引言

在 20 世纪 80 年代, 如果想用微控制器输出模拟信号, 通常都要使用数模转换器。近年来, 因为数字硬件的操作频率不断增加, 在很多情况下, 脉宽调制已经成为更经济的建立模拟信号的方法。因而, 数模转换器的应用领域诸如电动机控制和机器人逐渐缩减。

然而, 在两种应用中, 数模转换器仍然是经济的: 高频率/高位数应用 (特别是音频应用) 和过程控制。

本章将考虑如何使用数模转换器生成模拟输出, 以下模式将在本章中介绍:

- 数模转换输出
- 数模转换平滑滤波器
- 数模转换驱动器

数模转换输出

适用场合

为桌面或者嵌入式环境研发一个硬实时或者软实时系统 (各术语的定义参见第 1 章)。

问题

怎样用一个数模转换器 (DAC) 产生模拟信号?

背景知识

假设需要为家庭住宅研发一个入侵警报系统。当有人闯入时, 通常只是简单地响起警报(响铃) 告知周围的邻居。即, 入侵警报仅仅用于检测一类问题。

然而, 假设需要为工厂开发安全监测系统, 则各种各样的问题就都有可能出现了: 气体泄漏、核原料泄漏、火灾、闯入等等。在某些情况下, 工厂员工需要根据面临的威胁采取不同的

行动（例如，穿上各种不同的防护服装）。

怎样才能告诉工厂员工发生了哪种问题？

一个合适的解决方案是用数字化消息描述问题并告诉用户该如何处理。例如：“警告，检测到百万分之 345 单位浓度的 356 号污染物泄漏。请立刻使用呼吸器并按指定的疏散路线离开这幢建筑物。”

播放这类消息，通常需要使用数字-模拟转换器（DAC）。

数模转换器不仅用于数字化警告播放，在其他控制系统中也扮演了重要的角色。在本模式中将讨论如何使用数模转换器。

解决方案

设计数字模拟转换输出电路，需要做出几个关键的设计决策：

1. 确定所需的采样频率。
2. 确定所需的位数（数模转换器分辨率）。
3. 选择带有合适的片上数模转换器的 8051 系列微控制器，或者如果有必要的话，增加一个外部数模转换器。
4. 可能需要确定输出信号的频率响应特性曲线。
5. 选择合适的软件体系结构。

下面依次讨论以上各个设计阶段。

确定所需的采样频率

采样频率的讨论请参见序列模数转换。

确定所需的采样位数

采样位数的讨论请参见单次模数转换。

带片上数模转换器的 8051 微控制器

从单次模数转换可以看出，许多 8051 微控制器都包含片上模数转换器，但是这样的 8051 微控制器非常少。

以下是较新的两个提供了片上数模转换器组件的 8051 微控制器的例子。

Analog Devices AD μ C812

根据 Analog Devices^① 数据手册编写：

AD μ C812 内建了两个 12 位电压模式数模转换器。

数模转换器的操作通过三个特殊功能寄存器控制。在正常操作模式下，每个数模转换器在写入 SFR 的低字节（DACxL）时更新。通过设置 DACCON SFR 的同步位，两个数模转换器能够被同时更新。数模转换器能以 12 位或者 8 位模式工作，输出范围可编程为 0~2.5V 或者 0~VDD。

① www.analog.com

Cygnal 8051F000

根据 Cygnal^② C8051F000 数据手册编写：

C8051F000 系列微控制器有两个 12 位电压模式数模转换器 (DAC0、DAC1)。

每个数模转换器的输出摆幅为 0V~VREF-1 个最低位，对应的数字输入范围为 0x000~0xFFFF。以 DAC0 举例来说，12 位数据写入数据寄存器的低字节 (DAC0L) 和高字节 (DAC0H)。数据写入 DAC0H 寄存器后锁存在 DAC0 中，所以如果需要完整的 12 位分辨率，写入顺序应该是先 DAC0L 后 DAC0H。当使用数模转换模块的 8 位模式时，可以将 DAC0L 初始化为要求的值（一般为 0x00），仅向 DAC0H 写入数据。DAC0 控制寄存器 (DAC0CN) 可以使能/禁止 DAC0，以及更改输入数据的格式。

DAC0 的使能/禁止功能由 DAC0EN 位控制 (DAC0CN.7)。DAC0EN 置 1 时 DAC0 使能，DAC0EN 清 0 时 DAC0 停用。当禁止时，DAC0 的输出保持高阻状态，DAC0 馈电电流下降至 1 微安或者更小。

在有些情况下，输入数据在写入 DAC0 前需要先移位，以正确地与数模转换器输入寄存器对齐。这个操作一般需要用一个或多个读和移位指令完成，因而增加了软件开销，降低了数模转换器吞吐能力。为了减轻这个问题，数据格式化功能提供了允许用户编程控制 DAC0H 和 DAC0L 数据寄存器内 DAC0 数据格式的功能。三个 DAC0DF 位 (DAC0CN.[2:0]) 允许用户在五种数据格式中选择。

DAC1 的功能和上面所述的 DAC0 完全一样。

使用外部电压模式数模转换器

通常，使用片上数模转换器可以增加可靠性，减小系统体积，降低系统成本，但有时还是需要使用外部数模转换器。

就像模数转换器，(电压模式) 数模转换器也有并行总线连接和串行总线连接的。例如，对于并行总线的数模转换器，Analog Devices^③的产品有 AD7245a (12 位)、AD7248a (12 位) 和 AD7801 (8 位)，Maxim^④的产品有 Max5480 (8 位)。而对于串行总线数模转换器，Maxim 提供了 I²C 接口的 Max517 和 SPI 接口的 Max541。关于在时间触发环境中使用这些接口的讨论参见第 5 篇。

因为通常在需要高位数和高采样频率的环境中使用数模转换器，所以并行总线设计经常是最适用的解决方案。

使用外部电流模式数模转换器或者跨导放大器

正如在单次模数转换器中讨论的，在有些情况下，特别是监测或者过程控制中，使用电流模式器件（传感器或者传动器）可能比电压模式器件更合适。

用微控制器产生模拟电流信号主要有两种选择：

② www.cygnal.com

③ www.analog.com

④ www.maxim-ic.com

- 使用电流模式数模转换器
- 使用电压模式数模转换器和电压-电流转换器（通常被称为跨导放大器）

第二种选择可能对带片上数模转换器的微控制器特别有吸引力。

一个合适的数模转换器例子是 Analog Devices^⑤ AD421，特别针对 4~20mA 电流环设计，通过电流环供电。

跨导放大器的一个例子是 Burr Brown^⑥ 的 XTR 110。能被用于把 1~5V 的模拟电压（比如说微控制器电压模式数模转换器的输出）转换成过程控制系统需要的 4~20mA 电流。因而，对于一些智能过程传感器，跨导放大器是一种有用的器件。

频率响应特性曲线

使用数模转换器会引入噪音（通过混叠效应）和频率畸变。在大多数情况下，至少需要消除混叠效应。有关详细情形参见数模转换平滑滤波器。

一般的软件体系结构要求

在高频率下使用数模转换器（10kHz 或者 16kHz）对系统的总体结构有重大的影响。例如，即使只有 10kHz，也需要 0.1ms 的时标间隔，这对基本型 8051 微控制器是很大的负荷。

通常，只有每个指令的执行时钟周期数少于 12 的 8051 微控制器才能够提供这样的性能水平。使用较新的 8051 微控制器，例如 Dallas 89C420（一个每条指令只需要 1 个时钟周期的标准 8051，最高运行频率为 50MHz，参见第 3 章），以 16kHz（0.0625ms 时标间隔）的频率进行数模转换是能满足实用要求的。

硬件资源

使用片上模数转换器通常至少占用一个输入引脚，使用外部模数转换器会占用大量的引脚。

使用片上模数转换器可能也会增加微控制器的功耗。

可靠性和安全性

通常，使用片上数模转换器很可能能够改善系统的可靠性（和片外解决方案相比），因为硬件（或者和软件）的复杂性，焊点的数量等等都大大减少了。

另外，使用转换时间较长的模数转换器会增加延迟，这会降低信号处理系统的性能，可能会影响控制系统的稳定性。所以要确保数模转换器的速度和系统相匹配。

可移植性

各种数模转换器各有不同。基本原则和技术是可移植的，但细节依赖于具体的硬件。

⑤ www.analog.com

⑥ www.burr-brown.com

优缺点小结

- ◎ 数模转换输出在许多系统中都是必需的，并且通常易于使用。
- ◎ 带片上数模转换器的微控制器相对较少，而且比不带模数转换器的微控制器昂贵。

相关的模式和替代方案

大多数情况下，脉宽调制输出（参见第 33 章）是数模转换器的廉价而有效的替代方案，参见硬件脉宽调制。

适合于滤除使用数模转换器引入的噪音（由混叠效应引入）和频率畸变的技术参见数模转换平滑滤波器。

有关驱动诸如扩音器或者直流电动机之类的大功率负载所需硬件的介绍参见数模转换器驱动器。

例子：使用 12 位并行总线数模转换器的语音播放

这里考虑怎样使用一个 12 位并行总线数模转换器以 10kHz 的采样速度播放语音采样。

图 34.1 是语音片断的例子。

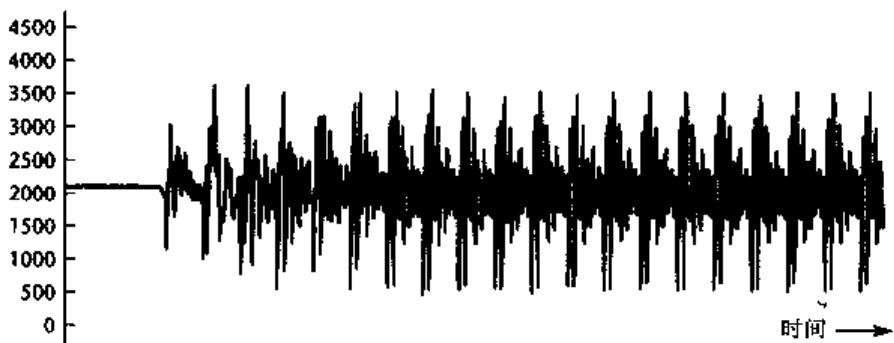


图 34.1 展示了语音波形

图 34.2 中为所使用的基本硬件。注意，平滑滤波和放大组件将在数模转换平滑滤波器和数模转换器驱动器中讨论。

关键代码在源程序清单 34.1~源程序清单 34.4 中给出。

```
/*
----- Port.H (v1.00) -----
项目 Play_DAC (参见第 34 章) 的端口头文件 (参见第 10 章)
----- Sch51.C -----
// 如果不需要错误报告, 注释掉此行
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
```

```

// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P1
#endif
// ----- Playback.c -----
#define SPEECH_Port P2
sbit SPEECH_CSLSB_pin = P0^0;
sbit SPEECH_CSMSB_pin = P0^1;
// ----- Swit_Ply.C -----
// 将一个按钮开关连接在这个引脚和地之间
// -由软件去抖动
sbit Sw_pin = P3^3; // 按钮开关引脚
/*
-----文件结束-----
*/

```

源程序清单 34.1 控制外部 12 位(并行总线)数模转换器例子的部分源程序

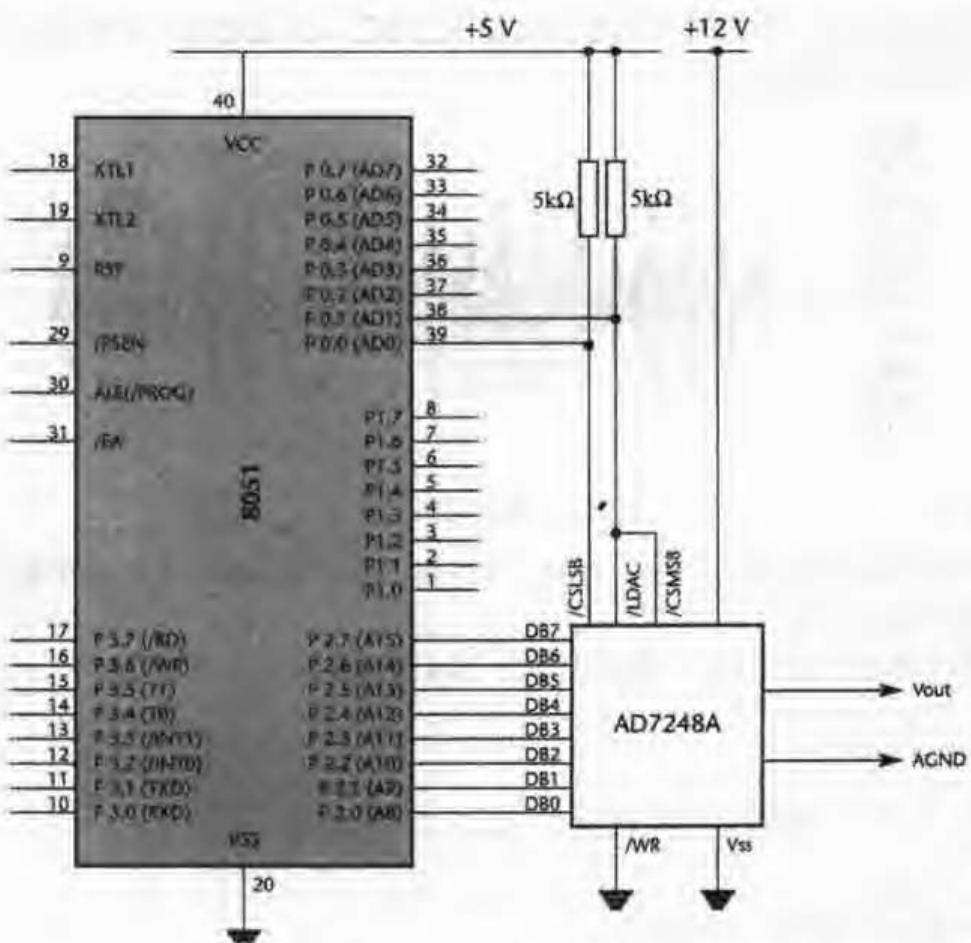


图 34.2 AD7148A 用于语音重放

第 34 章 数模转换器的应用 (DAC)

```
/*-----*
Main.c (v1.00)

语音重放的例子，使用混合式调度器。
所需的链接程序选项（详情参见第 14 章）：
OVERLAY
(main ~ (SWITCH_Update),
 SWITCH_Update ~ (SPEECH_PLAYBACK_Update),
 hSCH_dispatch_tasks ! (SWITCH_Update, SPEECH_PLAYBACK_Update))
*-----*/
```

```
#include "Main.h"
#include "2_01_12h.h"
#include "Swit_Ply.h"
#include "Playback.h"
/* ..... */
/* ..... */
void main(void)
{
    // 设置调度器
    hSCH_Init_T2();
    // 设置按钮开关引脚
    SWITCH_Init();
    // 加入开关任务（每 200ms 检查一次）
    // 这是个抢占式任务
    hSCH_Add_Task(SWITCH_Update, 0, 200, 0);
    // 注意：
    // 播放任务由 SWITCH_Update 任务加入
    // （按照用户的要求）
    // 播放任务是合作式任务
    // ***注意所需的链接器选项（参见上面的程序）***
    // 启动调度器
    hSCH_Start();
    while(1)
    {
        hSCH_Dispatch_Tasks();
    }
}
/*-----*
---文件结束 ---
*-----*/
```

源程序清单 34.2 控制外部 12 位（并行总线）数模转换器例子的部分源程序

```
/*-----*
Playback.C (v1.00)

播放存储在片上 ROM 中的语音采样的库函数
假设数模转换器为 AD7248A：硬件详见正文
当开关按下时连续不断地进行播放
采样数据以 10 kHz、12 位分辨率重放
*-----*/
```

```

-----*/
#include "Main.h"
#include "Port.h"
#include "Playback.h"
// -----公有常量-----
// 将要播放的语音数据
extern const tWord code BA_12_BIT_10KHZ_G[3500];
// -----公有变量声明-----
extern bit Sw_pressed_G;
// -----公有变量定义-----
bit SPEECH_PLAYBACK_Playing_G = 0;
// -----私有变量-----
static bit LED_state_G;
// -----私有常数-----
#define T_100micros (65536 - (tWord)((OSC_FREQ / 13000)/(OSC_PER_INST)))
#define T_100micros_H (T_100micros / 256)
#define T_100micros_L (T_100micros % 256)
/*-----*
SPEECH_PLAYBACK_Update()
播放软件的主要更新函数
根据需要，通常作为单次（合作式）任务调度
任务持续时间大约是 350ms
任何时候，用户都能松开开关，结束播放
*-----*/
void SPEECH_PLAYBACK_Update(void)
{
    int Sample;
    SPEECH_PLAYBACK_Playing_G = 0;
    // 定时器 0 配置为 16 位定时器
    TMOD &= 0xF0; // 清所有的 T0 位 (T1 不动)
    TMOD |= 0x01; // 设置所需要的 T0 位 (T1 不动)
    ET0 = 0; // 没有中断
    // 以~10 kHz 播放
    for (Sample = 0; Sample < 3500; Sample++)
    {
        // 避免此函数的重复调用
        SPEECH_PLAYBACK_Playing_G = 1;
        // 播放一个采样
        SPEECH_PLAYBACK_Play_Sample(BA_12_BIT_10KHZ_G[Sample]);
        // 延迟~0.1ms (为了以 10 kHz 采样速度播放)
        TR0 = 0;
        TH0 = T_100micros_H;
        TL0 = T_100micros_L;
        TF0 = 0; // 清除标志
        TR0 = 1; // 启动定时器
        while (!TF0);
        TR0 = 0;
        if (!Sw_pressed_G)
        {
}
}

```

```

        break; // 如果用户松开开关，退出循环
    }
}

// 退出此函数前，将标志设置为 0
SPEECH_PLAYBACK_Playing_G = 0;
}

/*-----*
 * SPEECH_PLAYBACK_Play_Sample()
 * 将 12 位采样数据送到 AD7248A 数模转换器
 *-----*/
void SPEECH_PLAYBACK_Play_Sample(const tWord SAMPLE)
{
    // 采样是 12 位，存储在 16 位 tWord 型变量中
    // 先送出低 8 位
    tByte Data_8bit = (tByte) (SAMPLE & 0x00FF);
    SPEECH_Port = Data_8bit;
    SPEECH_CSLSB_pin = 0;
    SPEECH_CSMSB_pin = 1;
    // 现在送出高 4 位
    Data_8bit = (tByte) ((SAMPLE >> 8) & (0x0F));
    SPEECH_Port = Data_8bit;
    SPEECH_CSLSB_pin = 1;
    SPEECH_CSMSB_pin = 0;
}
/*-----*
 *-----文件结束-----*
 *-----*/

```

源程序清单 34.3 控制外部 12 位（并行总线）数模转换器例子的部分源程序

```

/*-----*
 * SWIT_PLY.C (v1.00)
 *-----*
 * 简单的开关程序代码，带软件去抖动
 * 控制数模转换器语音播放
 *-----*/
#include "Main.h"
#include "Port.h"
#include "Swit_Ply.h"
#include "Playback.h"
#include "2_01_12h.h"
// -----公有变量定义-----
bit Sw_pressed_G = 0; // 当前开关状态
// -----公有变量声明-----
extern bit SPEECH_PLAYBACK_Playing_G; // 当前播放状态
// -----私有常量-----
// 允许使用常开或者常闭开关（或者其他接线方式）
#define SW_PRESSED (0)
// SW_THRES 必须>1，以确保正确地去抖动
#define SW_THRES (3)

```

```

/*
 *-----*
 * SWITCH_Init()
 * 开关程序库的初始化函数
 *-----*/
void SWITCH_Init(void)
{
    {
        Sw_pin = 1; // 用这个引脚作为输入
    }
/*-----*
 * SWITCH_Update()
 * 这是主开关函数。
 * 应该每 50~500ms 调度一次
 *-----*/
void SWITCH_Update(void)
{
    {
        static tByte Duration;
        if (Sw_pin == SW_PRESSED)
        {
            Duration += 1;
            if (Duration > SW_THRES)
            {
                Duration = SW_THRES;
                Sw_pressed_G = 1; // 开关被按下了...
                // 将播放任务加入到调度器
                // (在检查了播放任务是否已经在运行了以后)
                // 加入播放任务(持续时间 10 秒)
                // 这是一个合作式(单次)任务
                if (SPEECH_PLAYBACK_Playing_G == 0)
                {
                    hSCH_Add_Task(SPEECH_PLAYBACK_Update, 0, 0, 1);
                }
                return;
            }
            // 开关已经按下, 但按下时间还不够长
            Sw_pressed_G = 0;
            return;
        }
        // 开关没有被按下 - 复位计数
        Duration = 0;
        Sw_pressed_G = 0; // Switch not pressed... //开关没有被按下...
    }
/*-----*
 *-----文件结束 -----
 *-----*/

```

源程序清单 34.4 控制外部 12 位(并行总线)数模转换器例子的部分源程序

进阶阅读

数模转换平滑滤波

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

怎样减少数模转换输出中的转换噪音？

背景知识

背景资料参见数模转换器输出。

解决方案

假设需要建立一个用于水翼船的高品质数字通信系统。具体地说，假定水翼船包含一个电脑网络，用于非关键任务，例如监测客舱温度。这个网络有空闲的带宽，可以用于船员向旅客传达消息（如图 34.3 所示）。



图 34.3 设计供水翼船使用的数字通信系统

语音信号的播放（通过网络数字化传输）将使用图 34.4 所示的硬件。假定每个从节点都使用这样的硬件。

不幸的是，因为两个原因，这个系统产生的语音品质非常差。

播放品质差的第一个和最主要的原因是混叠效应：这个问题的讨论参见单次模数转换。混叠效应造成的影响可以用截止频率为采样频率一半的低通滤波器滤除。A-A 滤波器详细地讨

论了合适的滤波器实现。

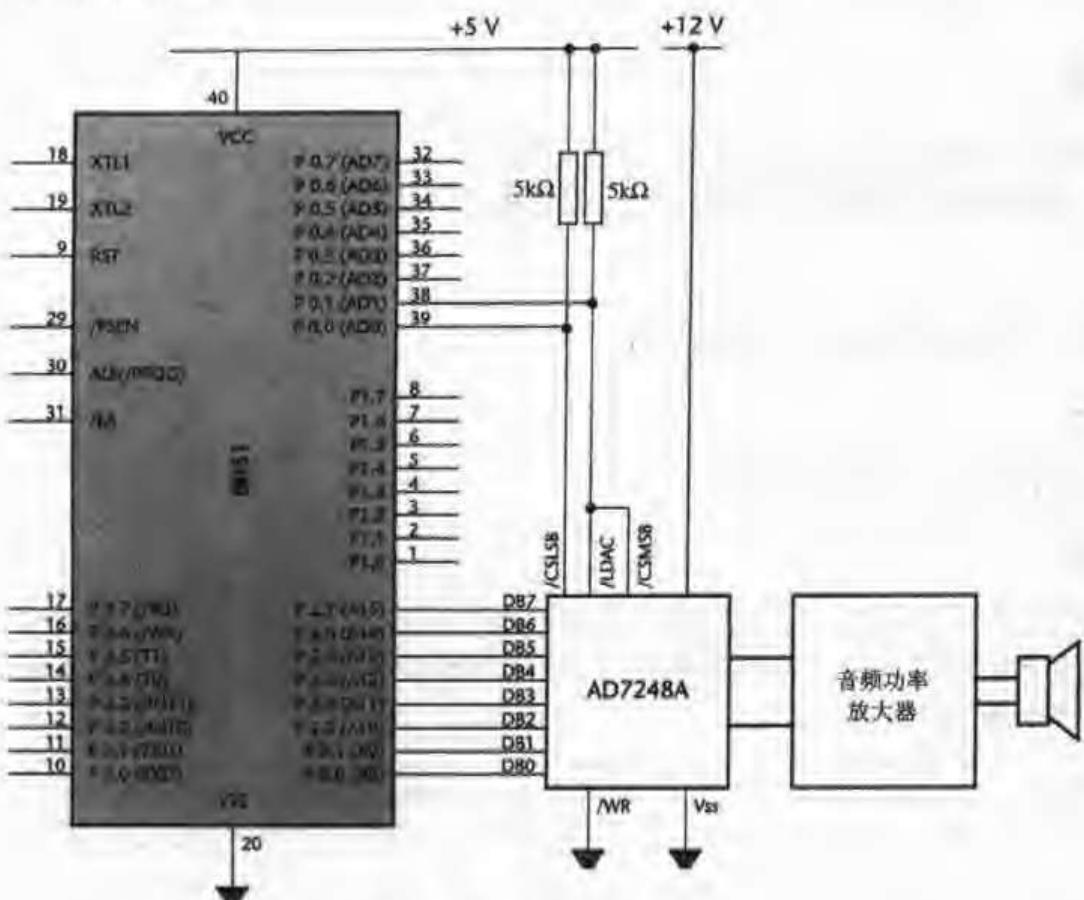


图 34.4 用于语音播放的硬件

播放品质不佳的第二个原因是数模转换输出是量化的，和图 34.5 上部所示的波形不同，数模转换产生的波形类似于图 34.5 下部所示的波形。

数模转换输出的阶梯特性引入了随频率而变的畸变。为了滤除畸变，需要使用称为正弦补偿的技术。这涉及到用一个相对较复杂的滤波器（正弦滤波器）对数模转换信号进行滤波。这种补偿的技术细节超出了本书的范围（详见 Smith, 1999）。

总的说来，为了达到最高品质的信号再现，需要图 34.6 所示的方案。然而，对于大多数的系统，用低通滤波器（截止频率为采样频率的一半）就能提供足够的性能。

硬件资源

这个模式只使用硬件，没有特别的（微控制器）硬件资源要求。

可靠性和安全性

使用合适的平滑滤波硬件有助于降低数据转换器硬件产生的高频率电磁干扰的水平。这一点

可能有助于改善系统的可靠性。

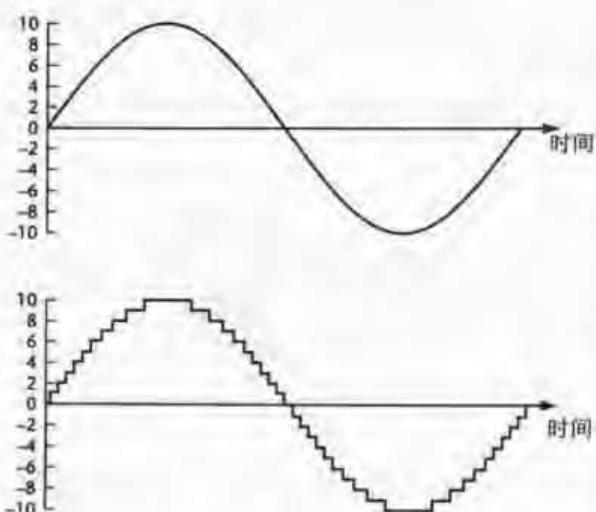


图 34.5 量化的正弦波形

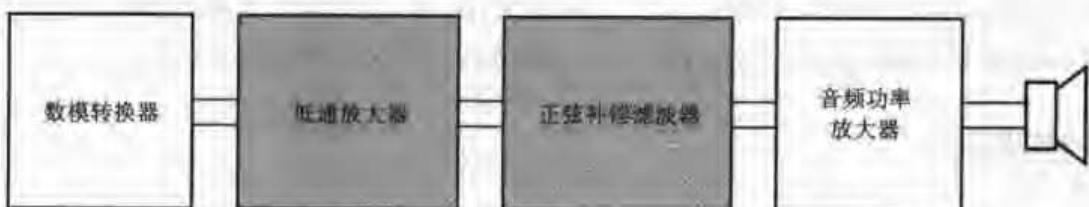


图 34.6 用数模转换器生成高品质的音频信号

可移植性

这个只使用硬件的模式具有很高的可移植性。

优缺点小结

- ⊕ 减少了数模转换输出的转换噪音。
- ⊗ 增加了系统的成本。

相关的模式和替代方案

参见 A-A 滤波器。

例子：使用 12 位并行总线数模转换器的语音播放

考虑一下第 657 页介绍的语音播放例子的数模转换输出。

为了例子中的硬件能构成一个完整的系统，至少还需要两个组件：合适的滤波器和某种形式的放大器。

这里考虑所需的滤波器。

正如模式中提到的，滤除高于 $F_s/2$ 的频率分量 (F_s 是采样频率) 对避免混叠效应是必不可少的，这里将不使用正弦补偿。

图 34.7 给出了一个合适的 5kHz 滤波器。设计这个运放滤波器所需的设计过程的详细情形请参阅 A-A 滤波器。

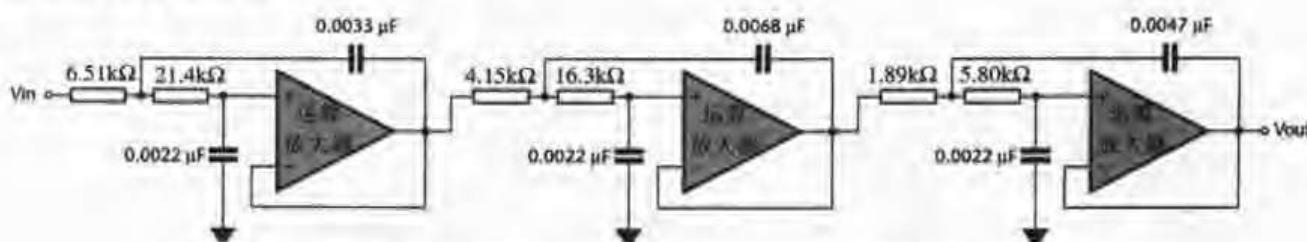


图 34.7 适用于语音播放例子的一个 5kHz 低通滤波器

进阶阅读

Smith, S.W. (1999) *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd edn, California Technical Publishing. [Available electronically at www.DSPguide.com]

数模转换驱动

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

怎样将电压模式模数转换器的输出转换为适合驱动大功率负载的形式？

背景知识

数模转换器的背景资料参见数模转换器输出。

解决方案

电压模式数模转换器是低功率设备。例如，16 位数模转换器 Maxim Max541 最大能驱动 50mA 电流^⑦。显而易见，如果想要驱动，比如说直流电动机这样的负载，需要某种形式的缓冲电路。

正如在第 7 章中看到的，当考虑开关直流负载时，主要有两个选择：使用基于分立元件的

^⑦ 这比许多普通数模转换器芯片要高。

设计（通常指 BJT）或者基于 IC 的设计（通常指功率运放）。

在随后的例子中，将说明这两种解决方案。

硬件资源

使用这个模式并不需要微控制器上的硬件资源（例如，CPU 的机时或者存储器）。

可靠性和安全性

无特别的可靠性或者安全性要求。

可移植性

这个模式只需要硬件，能和任何微控制器配合使用。

优缺点小结

- ⊕ 简单而且有效。
- ⊖ 如果使用运放，可能需要正负电源供应。

相关的模式和替代方案

参见数模转换器输出，以及数模转换平滑滤波器。

例子：用分立元件驱动扬声器

图 34.8 例示了一个基于 BJT 的放大器电路。具体地说，两个晶体三极管以达林顿管的形式连接，以增加电路的增益。一般这是必需的，因为功率晶体管的增益（例如 2N3055）不能太高。

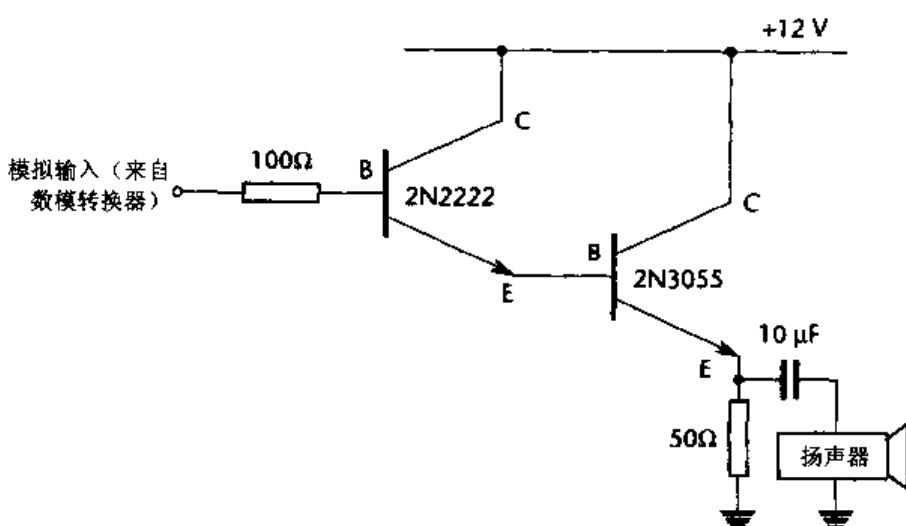


图 34.8 放大数模转换输出，以驱动小型扬声器

例子：用功率运放驱动扬声器

图 34.9 中的 National Semiconductor[®] 的 LM12CL 是一个适合于高品质音频设备的 IC 功率放大器，谐波失真号称只有约 0.01%。

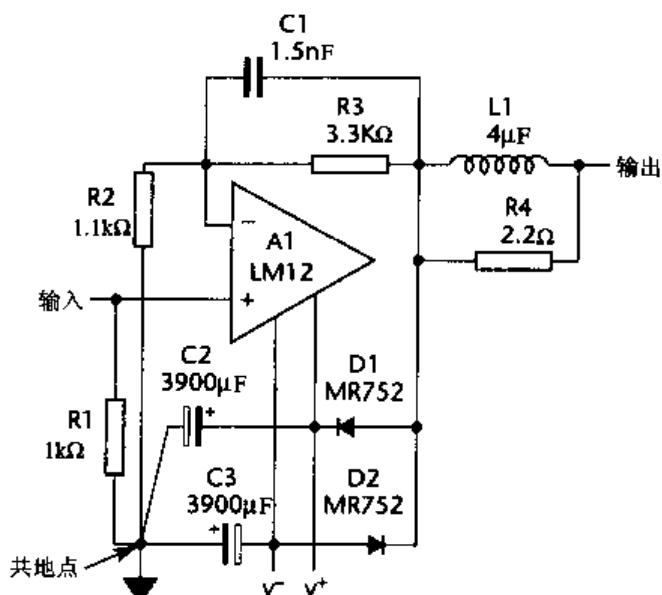


图 34.9 适用于高品质音频设备的功率放大器（承蒙 National Semiconductor 允许转载）

进阶阅读

Chapter 35

进行控制

引言

本章的重点是比例-积分-微分（PID）控制。PID 控制简单而且有效，是应用最广泛的控制算法。本章将着重讲述嵌入式系统中 PID 控制器的设计和实现技术。

PID 控制器

适用场合

- 用 8051 系列的微控制器开发一种嵌入式系统。
- 该系统为基于调度器的时间触发体系结构。

问题

怎样设计和实现一个 PID 控制算法？

背景知识

在本节中，将讨论为什么需要闭环的控制算法。

开环控制

假设需要控制一个直流电动机的速度，用作空中交通管制系统的一部分（如图 35.1 所示）。为了控制速度，假定使用数模转换器改变加在电动机上的电压。^①

正如前面所说的，这是普通开环控制方法的一个实例（如图 35.2 所示）。

使用开环控制时，需要了解系统参数，以产生所需的系统输出。因此，就空中交通管制系统来说，掌握了电动机、雷达硬件和电动机驱动电路的情况，就能够设置所需的转速。

^① 也可以使用脉宽调制控制。不过这会使这个例子复杂化，而最后得出的通用结论不会有什么不同。



图 35.1 作为空中交通管制系统的一部分的一台雷达

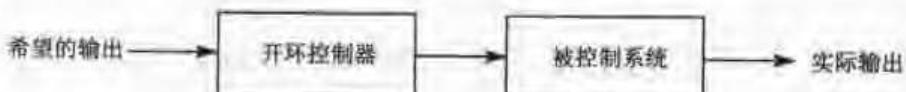


图 35.2 开环控制系统的简略原理图

在理想情况下，这类开环控制系统的设计很容易：只需要建立一个期望的电动机转速和所需功率参数的对应表格。例如，考虑一下用于导航系统核心部分的直流电动机，假定电动机的转速和外加电压成正比（如图 35.3 所示）。

所以，通过查找表，可以查出需要的转速（见表 35.1）。

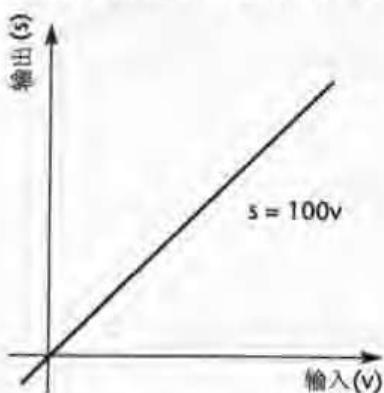


图 35.3 推算电压变化时直流电动机大概转速的简单模型

这是一个一般的线性单输入单输出（SISO）系统的例子：

这类系统可以用图 35.4 中的图表方式表示。

不幸的是，这种线性系统在实际系统中非常罕见。例如，实际的电动机有最高输入电压和对应的最高转速（如图 35.6 所示）。而且，实际的电动机在所加电压达到某个最小电压前不会开始转动，转速曲线也不会突然在最高转速处停止上升，而会表现为图 35.6 中所示的 I-O 曲

线。

表 35.1 将图 35.3 中的简单模型转换成查找表

雷达转速 (每分钟转数)	数模转换器设置 (8 位)
0	0
2	51
4	102
6	153
8	204
10	255

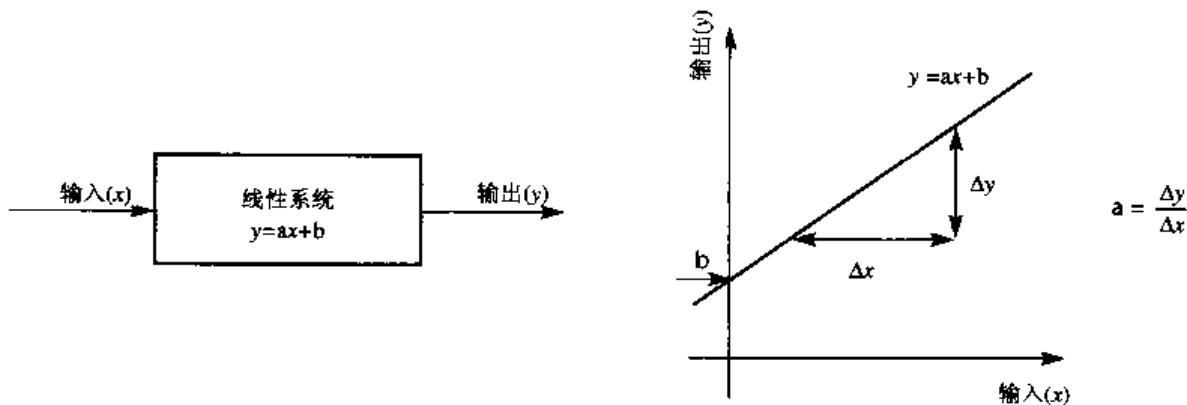


图 35.4 线性系统建模

总的说来，真正的电动机是一个非线性系统，不能用一个简单的线性（直线）模型精确表示。看一下图 35.6，描述这条曲线（也就是说，建立电动机的模型）比描述图 35.3 中的简单线性模型复杂得多。尽管如此，开环查找表方法能够处理非线性（见表 35.2）。

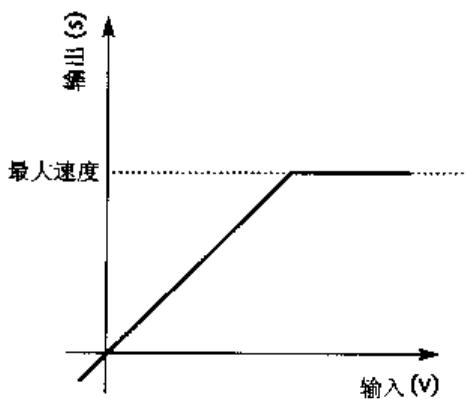


图 35.5 推算电压变化时直流电动机大概转速的更实际一点的模型

然而，需要处理的不只是这些问题。假设表 35.2 是根据一系列实际系统测试的结果编制的，因此考虑了电动机和系统其他组成部分的非线性特性。然而，除了非线性之外，大多数实际系统也表现出时变的特性。就雷达系统来说，表 35.2 没有考虑风速或者风向的影响。因而，表中的数值只在静止条件（无风）下是正确的。

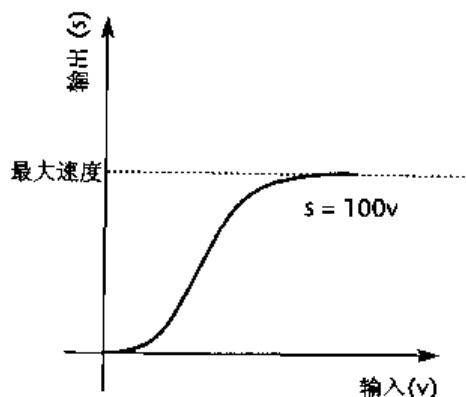


图 35.6 直流电动机模型的更进一步改进

表 35.2 将图 35.6 中的模型转换成查找表

雷达转速（每分钟转数）	数模转换器设置（8 位）
0	0
2	61
4	102
6	150
8	215
10	255

如果决定使用开环方法，则必须测量风速和风向（如图 35.7 所示）。然后还要编制一张有 5 英里/小时东南风时的表和另一张有 10 英里/小时西北风时的表，等等。

这样做下去，这种控制系统的设计方法很快就变得不切实际了。

闭环控制

开环控制系统的基本问题是雷达控制系统的控制器是盲目的：收不到所控制的系统输出（这里是转速）的反馈。

再看看闭环的雷达控制系统（如图 35.8 所示）。这种形式的控制器的关键特点是反馈回路（参见图 35.9）。能够让系统有效地对诸如风速或者风向变化这类的外界干扰做出反应。

应该使用哪种闭环控制算法？

有大量的控制算法可用于图 35.9 中的闭环控制器模块，新算法的开发和评价也是许多大



学的热门研究领域。Nise (1995)、Duttonetal. (1997)、Dorf 和 Bishop (1998) 讨论了一些可用的算法。

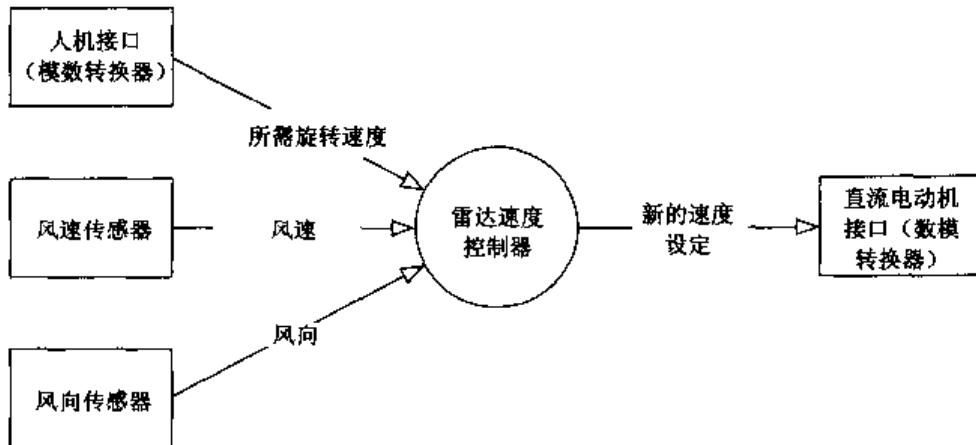


图 35.7 开环雷达控制系统设计方法的结果

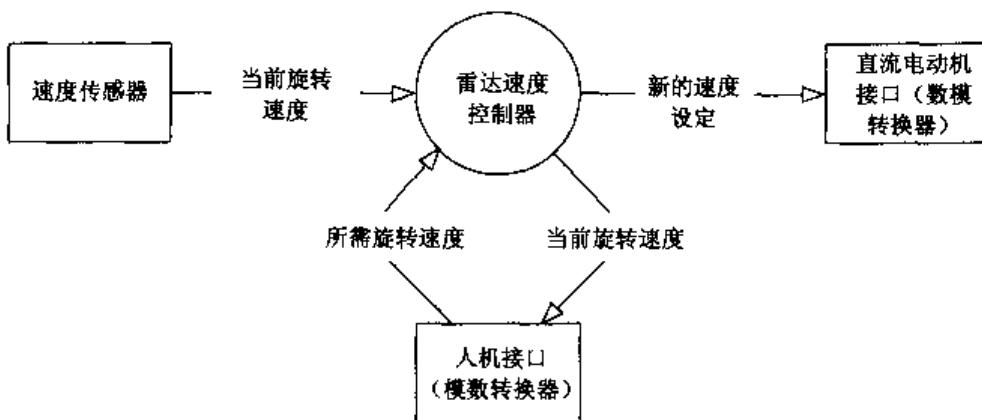


图 35.8 空中交通管制系统中的雷达转速控制（闭环版本）

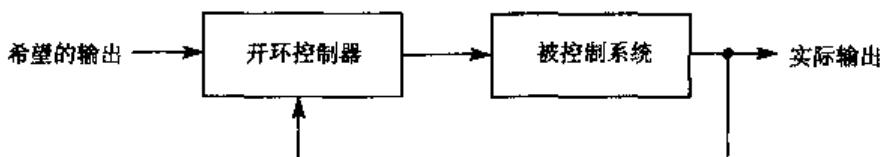


图 35.9 闭环控制系统的简略原理图

尽管可用的算法种类繁多，但比例-积分-微分 (PID) 控制在多数情况下非常有效，因而通常被认为是评判其他算法的标准。毫无疑问，PID 目前仍然是世界上使用最广泛的控制算法。

解决方案

在本节中，将考虑如何用微控制器实现基于比例积分微分的控制算法。

什么是比例-积分-微分控制?

翻开任何一本控制理论的教科书，都能看到包含类似图 35.10 中等式的比例-积分-微分控制的讲述。

这看上去似乎很复杂，但是如果了解算法要做的是什么，实际上能够很简单地实现。

例如，这里有一个完整的 PID 控制算法：

```
// 比例项
Change_in_controller_output = PID_KP * Error;
// 积分项
Sum += Error;
Change_in_controller_output += PID_KI * Sum;
// 微分项
Change_in_controller_output += (PID_KD * SAMPLE_RATE * (Error - Old_error));
```

$$u(k) = u(k-1) + k \left[\left(1 + \frac{T}{T_1} + \frac{T_D}{T} \right) e(k) - \left(1 + 2 \frac{T_D}{T} \right) e(k-1) + \frac{T_D}{T} e(k-2) \right]$$

其中：

$u(k)$ 是输出信号， $e(k)$ 是误差信号，两者都为第 k 个采样点的值

T 是采样周期（单位为秒）， $1/T$ 是采样频率（单位为 Hz）

K 是比例增益

$1/T_1$ 是积分增益

T_D 是微分增益

图 35.10 PID 控制算法的一种表示

算法有三个组成部分：比例部分、积分部分和微分部分。

开始将只考虑比例项。在后文会看到，比例项是控制算法的主要部分。实际上，仅用这一项就能构成许多有效的闭环控制系统，有时这被称为比例控制器。后面还将看到，如果需要的话，积分和微分项可用于调整基本的比例响应。

为了理解算法的运作，假设在公路上开车并且只用加速踏板（油门）控制速度。当前的速度是 30 英里/小时，如果希望加速到 35 英里/小时，只要轻轻地踩下踏板；如果希望加速到 70 英里/小时，则需要更有力地踩下踏板。

这就是比例控制的基础。具体地说，测量希望的系统输出（例子中希望的车辆速度）和当前系统输出（车辆的当前速度）之间的误差。差值（希望的速度-当前的速度）就是误差，比例算法设法将这个误差降低到 0。算法通过与误差项成比例地改变控制器的输出（例子中为踏板设置）做到这一点。

因此，PID 控制器的比例项可以实现如下：

```
Change_in_controller_output = PID_KP * Error;
```

其中， PID_KP 是比例增益，由用户校准，以适应特定系统的需要。

源程序清单 35.1 中给出了一个完整的 PID 控制算法实现，积分和微分项的影响将在后面讨论。

```
/*
  PID_f1.C (v1.00)
  简单的比例-积分-微分控制实现。
-----*/
#include "PID_f1.h"
// -----私有常量-----
#define PID_KP (0.7f)          // 比例增益
#define PID_KI (0.03f)         // 积分增益
#define PID_KD (0.04f)         // 微分增益
#define PID_MAX (0.5f)         // 最大 PID 控制器输出
#define PID_MIN (0.0f)          // 最小 PID 控制器输出
// -----私有变量定义-----
/*
static float Sum_G;           // 积分器部分
static float Old_error_G;    // 前一次的误差值
*/
PID_Control()
简单的浮点版本。
/*
float PID_Control(float Error, float Control_old)
{
  // 比例项
  float Control_new = Control_old + (PID_KP * Error);
  // 积分项
  Sum_G += Error;
  Control_new += PID_KI * Sum_G;
  // 微分项
  Control_new += (PID_KD * SAMPLE_RATE * (Error - Old_error_G));
  // Control_new 不能超过 PID_MAX 或者低于 PID_MIN
  if (Control_new > PID_MAX)
  {
    Control_new = PID_MAX;
  }
  else
  {
    if (Control_new < PID_MIN)
    {
      Control_new = PID_MIN;
    }
  }
  // 存储误差值
  Old_error_G = Error;
  return Control_new;
}
*/
```

```

    }
/*
-----文件结束-----
*/

```

源程序清单 35.1 PID 控制算法的一个完整实现

处理“顶死”（积分器饱和）

虽然源程序清单 35.1 中的算法能够正常工作，但是对积分环节做一个很小的改变，就能够进一步地改善性能，注意这个环节是如何实现的：

```

Sum_G += Error;
Control_new += PID_KI * Sum_G;

```

真实系统经常发生的一个问题是，有时传动装置已经达到最大（或者最小）极限；例如脉宽调制单元输出占空比已经达到 100%。在这种情况下，传动装置的输出已经不能再增加（或者减少），改变 Sum_G 的值已经不再起作用。相反，系统会在误差来源消除后反应缓慢。

所以，当传动装置到达极限时，停止更新 Sum_G 能够改善系统的性能。这个办法被称作“反顶死”。

源程序清单 35.2 中的比例积分微分实现包含了这种形式的顶死保护。

```

/*
-----*
PID_wf1.C (v1.00)
-----*
简单的比例-积分-微分 (PID) 控制实现，带顶死保护。
*/
#include "PID_wf1.h"
// -----私有常量-----
#define PID_KP (0.2f)           // 比例增益
#define PID_KI (0.01f)          // 积分增益
#define PID_KD (0.03f)          // 微分增益
#define PID_WINDUP_PROTECTION (1) // 设置为真 (1) 或者假 (0)

#define PID_MAX (1.0f)           // 最大 PID 控制器输出
#define PID_MIN (0.0f)           // 最小 PID 控制器输出
// -----私有变量定义-----
static float Sum_G;           // 积分器部分
static float Old_error_G;     // 前一次的误差值
/*
PID_Control()
简单的浮点版本。
详见正文。
*/
float PID_Control(float Error, float Control_old)
{
    // 比例项
    float Control_new = Control_old + (PID_KP * Error);
}
```

```

// 积分项
Sum_G += Error;
Control_new += PID_KI * Sum_G;
// 微分项
Control_new += (PID_KD * SAMPLE_RATE * (Error - Old_error_G));
// 可选的顶死保护——参见正文
if (PID_WINDUP_PROTECTION)
{
    if ((Control_new > PID_MAX) || (Control_new < PID_MIN))
    {
        Sum_G -= Error; // 不增加 Sum...
    }
}
// Control_new 不能超过 PID_MAX 或者低于 PID_MIN
if (Control_new > PID_MAX)
{
    Control_new = PID_MAX;
}
else
{
    if (Control_new < PID_MIN)
    {
        Control_new = PID_MIN;
    }
}
// 存储误差值
Old_error_G = Error;
return Control_new;
}

/*
-----文件结束 -----
*/

```

源程序清单 35.2 一个 PID 控制算法的完整实现，带顶死保护

选择控制器参数

PID 控制算法的两个方面会对新用户造成困扰。第一个是算法似乎很复杂：其实如前文举例说明过的，事实并非如此，PID 控制器能够非常简单地实现。

第二个是控制器参数的调整。幸而，这个问题通常也被夸大了。

建议采用以下方法调整比例积分微分参数：

1. 将积分 (KI) 和微分 (KD) 项设置为 0。
2. 慢慢地增加比例项 (KP)，直到系统输出发生连续的振荡。
3. 把 KP 的值降低一半。
4. 必要时，用较小的 KD 值进行试验，以阻尼响应中的振荡成分。
5. 必要时，用较小的 KI 值进行试验，以减少系统的稳态误差。

6. 如果 K_I 的值非零，一定要使用顶死保护。

注意，第 1~3 步中的技术是 Ziegler - Nichols 比例积分微分调整规则的简化版本，这种方法可以追溯到 20 世纪 40 年代（参见 Ziegler 和 Nichols, 1942; Ziegler 和 Nichols, 1943）。

下面的例子具体说明了这个方法。

应该采用多高的采样频率？

第 32 章讨论了如何选择合适的信号处理系统采样频率，并举例说明了过低的采样频率造成的混叠效应。具体而言，采样频率至少应该是系统带宽的两倍，而且此带宽由被分析系统的最高频率分量决定。这样，处理包含最高 4kHz 频率分量的语音信号，至少需要以 8kHz 的频率采样，并用抗混叠滤波器滤除所有更高的频率。这类系统所需带宽的确定一般使用频谱分析仪进行。

确定控制系统所需的采样频率要用一个稍有不同的方法进行。其中的一种有效技术是通过测量系统的上升时间进行的（如图 35.11 所示）。下面举例说明上升时间的开环测试方法，让一台发动机以最小速度运转，然后完全打开节气门，测量系统用了多长时间达到最高转速。

确定了上升时间后（以秒为单位），在一些简化假设的前提下，可以按如下公式计算所需的采样频率：

$$\text{采样频率} = \frac{6}{\text{上升时间}}$$

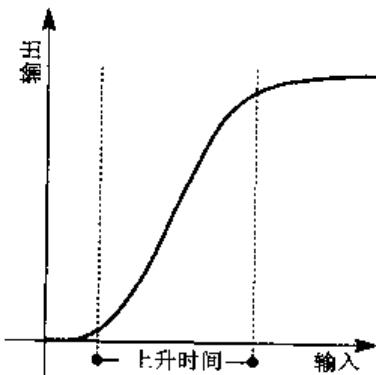


图 35.11 测量系统的上升时间

在这里，采样频率的单位是赫兹，上升时间的单位是秒。所以，如果图 35.11 中的上升时间是 0.10 秒，则所需的采样频率应为 6Hz。

这个方法假定带宽大约是系统固有频率的 20 倍，这是个合理的假定（有关的详细介绍请参见 Franklin 等, 1994）。

硬件资源

PID 控制算法的实现需要进行一些浮点或者整数的数学运算。准确的运算负荷随着具体的实现而变化，对一个典型的实现而言，需要 4 次乘法，3 次加法和 2 次减法。对于浮点运算，

这总共需要 2000 条左右的指令（使用 Keil 编译器、无硬件浮点支持的 8051）。如果没有其他消耗 CPU 计算能力的处理，这些运算能够在一个标准的 24MHz（12 个振荡周期/指令）8051 上每毫秒进行一次。

对大多数的控制系统每毫秒一次的循环时间已经大大短于要求的循环时间，典型的采样间隔一般是几百毫秒或者更长。当然，如果需要高性能的微控制器，有很多较现代的 8051 微控制器可用。例如，Dallas 520 微控制器（运行于 32MHz）能在 0.25ms 左右执行完一次 PID 运算。类似的微控制器，例如，带有硬件数学运算支持的 Infineon 517 和 509，也能够飞快地执行这些程序代码。

可靠性和安全性

任何控制危险传动装置的设备（例如，大功率电动机或者机器人）都有安全性要求，必须小心处理应用于这类负载的算法。

可移植性

这里讨论的 PID 算法能被用于任何微控制器或者微处理器。

优缺点小结

- ◎ 适用于许多单输入单输出（SISO）系统。
- ◎ 通常很有效。
- ◎ 实现简单。
- ◎ 不适用于多输入或者多输出的系统。
- ◎ 参数调整可能很费时。

相关的模式和替代方案

在本节中将考虑一些可能的 PID 控制替代方案。

为什么开环控制器有时仍然有用

背景知识一节中提出了闭环控制始终是比开环控制更好的方案，这是个过分简单化了的结论。

开环控制仍然扮演着一定的角色。例如，在汽车空调系统中控制电风扇的速度，当不需要精确的转速控制时，开环控制一般更为合适。

此外，被控制的量并不总是能够直接测量，在这种情况下，闭环控制是不切实际的。例如，用于治疗糖尿病人的胰岛素注射系统，需要设法控制血液中的葡萄糖水平。然而，现在的技术还不能制造葡萄糖传感器，所以必须使用开环控制器[有关详细情形参见 Dorf 和 Bishop(1998, p.22)]。

因为缺乏能测量产品质量的传感器，类似的问题在加工工业普遍存在。

PID 控制的限制

PID 控制只适用于单输入单输出 (SISO) 系统或者能分解为 SISO 组件的系统。PID 控制不适用于多输入或者多输出系统。此外，即使是 SISO 系统，比例积分微分也只能控制单个系统参数，所以不适合于多参数（有时被称为多变量）系统。

有关多输入、多输出和多参数控制算法的更进一步的讨论请参阅 Franklin 等 (1994)、Nise (1995)、Dutton 等 (1997)、Dorf 和 Bishop (1998)、Franklin 等 (1998)。

模糊控制

前面主要讲述的都是传统的（基于数学方法）控制系统设计方法。最近出现了一种不很正式的控制系统的设计方法，这种方法被称为模糊控制，适合于单个或者多个参数的 SISO、MISO 和 MIMO 系统（有关模糊控制的更加详尽资料参见 Passino 和 Yurkovich, 1998）。

例子：调整巡航控制系统的参数

在这个例子中，对汽车进行简单的计算机模拟，并开发一个巡航控制系统。

模型在源程序清单 35.3 中给出。

```
/*-----*
Cruise.CPP (v1.00)
-----*
举例说明 PID 控制的台式机 C++ 程序。
使用简单车辆模型的巡航控制系统。
-----*/
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "PID_f1.h"
// -----私有常量-----
#define MS_to MPH (2.2369) // 将米/秒转换为英里/小时
#define FRIC (50) // 摩擦系数——牛秒/米
#define MASS (1000) // 车辆的质量（公斤）
#define N_SAMPLES (1000) // 采样数
#define ENGINE_POWER (5000) // 牛顿
#define DESIRED_SPEED (31.3f) // 米/秒 [* 2.2369 转换为英里/小时]
/* ..... */ * /
/* ..... */ * /
int main()
{
    float Throttle = 0.313f; // 油门位置（分数）
    float Old_speed = DESIRED_SPEED, Old_throttle = 0.313f;
    float Speed, Accel;
    float Dist;
    float Sum = 0.0f;
    float Error;
    // 打开文件存入结果
```

```

fstream out_FP;
out_FP.open("pid.txt", ios::out);
if (!out_FP)
{
    cerr << "ERROR: Cannot open an essential file.";
    return 1;
}
for (int t = 0; t < N_SAMPLES; t++)
{
    // 误差驱动控制器
    Error = (DESIRED_SPEED - Old_speed);
    // 计算油门位置
    Throttle = PID_Control(Error, Throttle);
    // Throttle = 0.313f;
    // 用作开环控制演示
    // 简单汽车模型
    Accel = (float)(Throttle * ENGINE_POWER - (FRIC * Old_speed)) / MASS;
    // 扰动
    Dist = Old_speed + Accel * (1.0f / SAMPLE_RATE);
    Speed = (float) sqrt((Old_speed * Old_speed) + (2 * Accel * Dist));
    // Disturbances
    if (t == 50)
    {
        Speed = 35.8f; // 突然吹向汽车后部的阵风
    }
    if (t == 550)
    {
        Speed = 26.8f; // 突然吹向汽车前部的阵风
    }
    // 以英里/小时显示速度
    cout << Speed * MS_to MPH << endl;
    out_FP << Speed * MS_to MPH << endl;
    // 准备下一次循环
    Old_speed = Speed;
    Old_throttle = Throttle;
}
return 0;
}
-----*
---文件结束 ---
*/

```

源程序清单 35.3 用于说明巡航控制系统设计的一个简单汽车模型

通过模型在开环模式下的运行情况（如图 35.12 所示），能够说明模型的基本运转方式。

在图 35.12 中，汽车一直保持固定的油门位置。因为假设汽车在一条笔直、平坦、无风的道路上行驶，1000 秒过程中的大部分速度都是常数（70 英里/小时）。然而，有两处例外。在 $t = 50$ 秒处，模拟了汽车后部突然出现阵风，这使得汽车的速度上升，然后再慢慢地回到设定

的速度。类似地，在 $t = 550$ 秒处，模拟了汽车前部突然出现阵风，这使得汽车的速度下降。

下面将通过一系列的过程探讨 PID 控制算法的调整。调整的目的是让汽车遇到扰动时更迅速地恢复到指定的巡航速度。所用的比例积分微分算法已经在源程序清单 35.2 中给出了。

调整参数所用的方法就是在解决方案一节中提到过的方法：

1. 将积分 (KI) 和微分 (KD) 项设置为 0。
2. 慢慢地增加比例项 (KP)，直到系统输出发生连续的振荡。
3. 把 KP 的值降低一半。
4. 必要时，用较小的 KD 值进行试验，以阻尼响应中的振荡成分。
5. 必要时，用较小的 KI 值进行试验，以减少系统的稳态误差。
6. 如果 KI 的值非零，一定要使用顶死保护。

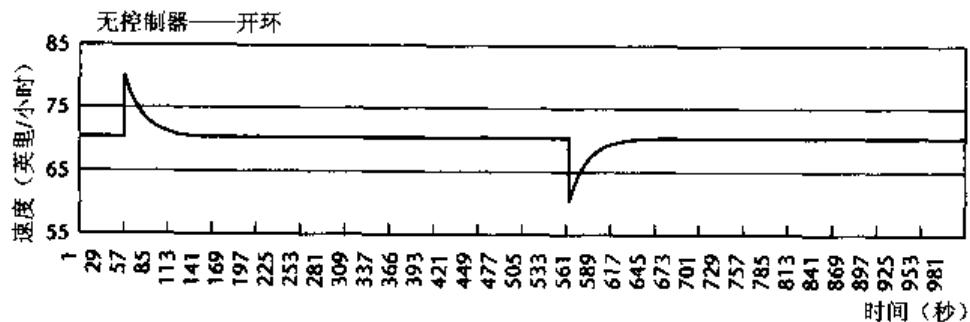


图 35.12 以开环控制方式运行巡航控制系统

图 35.13 中是第一步调整的结果。

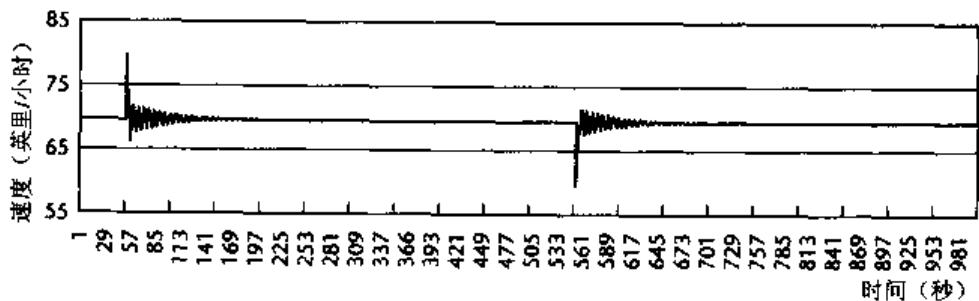


图 35.13 调整控制器

图 35.14 是第二步调整——稳定地增加 KP 的值，直到响应为较小的持续振荡的结果。

实验的结果说明 $KP = 0.5$ 是合适的取值（即，产生连续振荡时 KP 值的一半）。取这个值时，系统在扰动下的响应如图 35.15 所示。

这个响应结果可能对基本的巡航控制系统已经够用了，然而，调整准则中建议利用适当的微分项减小暂态振铃效应（每次扰动后都会出现）。经过简单的实验，确定图 35.16 中的参数是合适的。

注意，使用了这些参数，可以使系统在扰动发生后几秒内就恢复到制定的巡航速度。可以和图 35.12 中最初的开环控制版本比较一下。

同时也应注意到，由于省略了积分项而只使用比例微分控制，系统复杂性得到了降低。

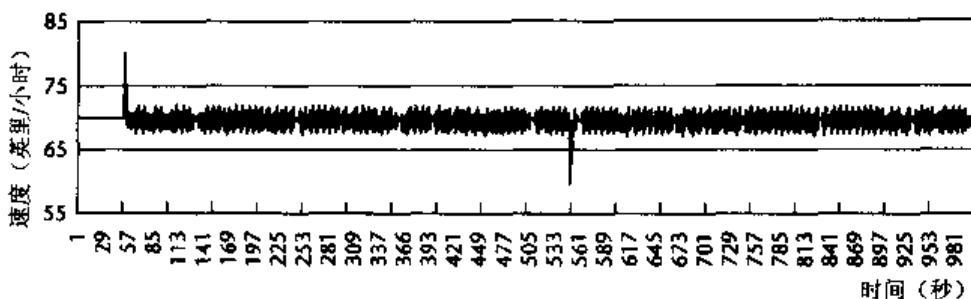


图 35.14 调整控制器

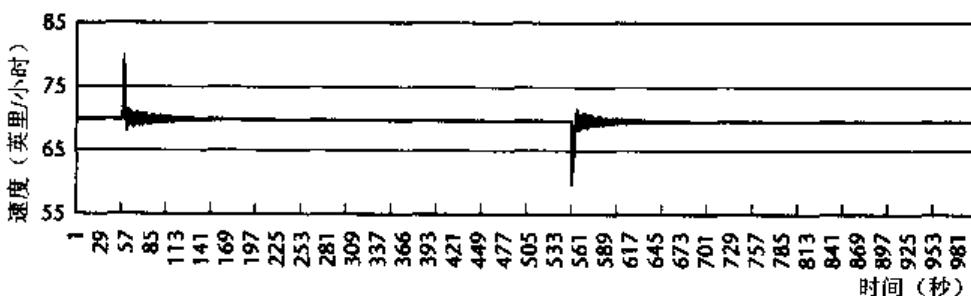


图 35.15 调整控制器

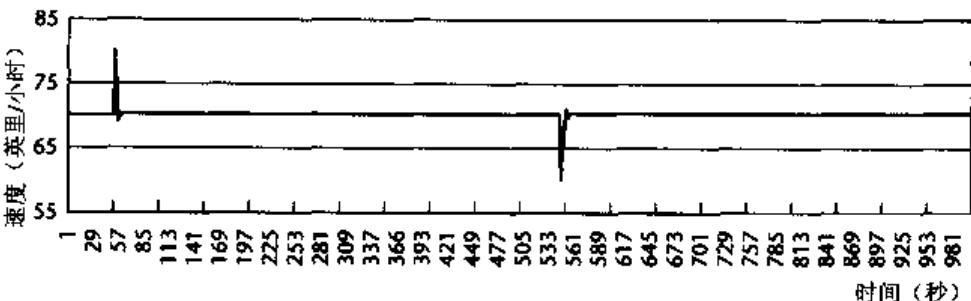


图 35.16 调整控制器

例子：直流电动机转速控制

上一个例子基于的是计算机模拟。在这个例子中，将举例说明一个实际的直流电动机的闭环控制。

电动机通过脉宽调制接口控制（如图 35.17 所示）。为了实现闭环控制，在电动机的轴上安装了一个编码器：轴每旋转一周，便生成一个脉冲。

这个系统所需的关键源程序文件在源程序清单 35.3~源程序清单 35.6 中给出，本书附带的 CD 上有这个项目所需的全部文件。

注意，这个例子使用了一个不同的、基于整数运算的 PID 实现。在硬件资源一节中已经讨论过，基于整数运算的解决方案的 CPU 负载比基于浮点运算的解决方案低得多。

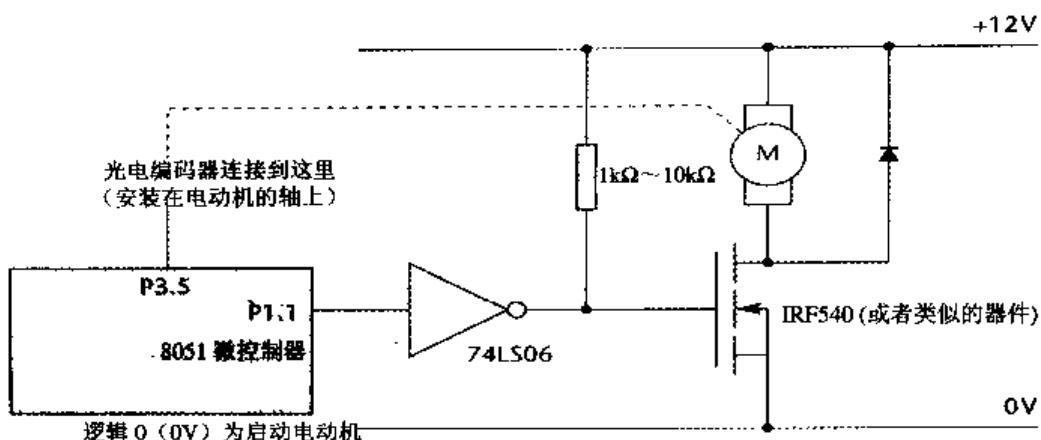


图 35.17 直流电动机转速控制系统的硬件

```
/*
Port.H (v1.01)

项目 PIDmotor 的端口头文件 (参见第 10 章)
*/
// ----- Sch51.C -----
// 如果不需要错误报告, 注释掉此行
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 用于显示错误码的端口
// 只在报告错误时用到
#define Error_port P5
#endif
// ----- PIDmotor.C -----
// 从引脚 6.0 读模数转换器
// 引脚 1.1 上输出脉宽调制信号
// 光电编码器 (或者类似的传感器) 连接到这个引脚
sbit Pulse_count_pin = P3^5;
// ----- Lnk_O.C -----
// 引脚 3.1 用作 RS-232 接口
/*
----- 文件结束 -----
*/

```

源程序清单 35.4 直流电动机 PID 控制例子的部分源程序

```
/*
Main.c (v1.01)

电动机 (转速) 控制示范
比例积分算法
*/
#include "Main.h"
#include "l_01_12i.h"
```

```

#include "PIDMotor.h"
#include "Lnk_O_B.h"
/* ..... */
/* ..... */
void main(void)
{
    SCH_Init_T2(); // 设置调度器
    PID_MOTOR_Init();
    // 波特率设置为 9600, 使用内部波特率发生器
    // 普通版本 8051
    PC_LINK_Init_Internal(9600);
    // 增加一个“脉冲计数轮询”任务
    // 时间单位是定时间隔数 (1ms 定时间隔)
    // 每 5ms 一次 (调度器 200 次)
    SCH_Add_Task(PID_MOTOR_Poll_Speed_Pulse, 1, 1);
    SCH_Add_Task(PID_MOTOR_Control_Motor, 300, 1000);
    // 数据送到串行端口
    SCH_Add_Task(PC_LINK_Update, 3, 1);
    // 所有的任务已加入: 启动调度器
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
/*-----文件结束-----*/

```

源程序清单 35.5 直流电动机 PID 控制例子的部分源程序

```

/*
  PID_Motor.c (v1.00)
  -----
  直流电动机 PID 控制的小程序库。
  为 C515c 微控制器编写。
  设定点 (需要的速度) 通过一个电位器和片上模数转换器读取
  当前速度通过光学编码器读取。用 T0 对编码器的脉冲进行计数
  速度通过脉宽调制设定, 使用片上捕捉一比较单元 (定时器 2)
*/
#include "Main.h"
#include "Port.h"
#include "PIDMotor.h"
#include "Lnk_O_B.h"
// -----公有常量-----
extern const char code CHAR_MAP_G[10];
// -----私有函数原型-----
static tByte PID_MOTOR_Get_Required_Speed(void);
static tByte PID_MOTOR_Read_Current_Speed(void);

```

```
static void PID_MOTOR_Set_New_PWM_Output(const tByte);
// -----私有常量-----
#define PULSE_HIGH (0)
#define PULSE_LOW (1)
#define PID_PROPORTIONAL (5)
#define PID_INTEGRAL (50)
#define PID_DIFFERENTIAL (50)
// -----私有变量-----
// 仅用于演示目的
tWord Ticks = 0;
// 存储最新的计数值
static tByte Pulse_count_G;
// 数据复制到串口
static char PID_MOTOR_data_G[50] = {" "};
// 实测转速，需要的转速和控制器输出变量
static tByte Speed_measured_G = 45;
static tByte Speed_required_G = 50;
static tByte Controller_output_G = 128;
static int Old_error_G = 0;
static int Sum_G = 0;
/*-----*
 * PID_MOTOR_Init()
 * 设置 UD 电动机控制
 *-----*/
void PID_MOTOR_Init(void)
{
    /*
    // 设置准备通过 RS-232 发给 PC 的起始数据
    //

    char* pScreen_Data = "Cur      Des      PWM      \n";
    tByte c;
    for (c = 0; c < 30; c++)
    {
        PID_MOTOR_data_G[c] = pScreen_Data[c];
    }

    // 设置模拟-数字转换器
    // 用于测量设定点（想要的电动机转速）
    //

    // 选择内部触发单次转换
    // 从 P6.0 (单通道) 读取
    ADCON0 = 0xC0; // 掩码位 0~5 到 0
    // 选择合适的预分频比，详见手册
    ADCON1 = 0x80; // 置位 7 = 1; 预分频比 = 8
    //

    // 设置脉宽调制输出 (捕捉比较) 单元——T2
    // (用于设定所需的电动机转速)
    //
*/}
```

```
----- T2 模式 -----
// 方式 1 = 定时器功能
// 预分频: Fcpu/6
----- T2 重新装载模式选择 -----
// 方式 0 = 定时器溢出时自动重新加载
// 将定时器寄存器预置为自动重装值: 0xFF00
TL2 = 0x00;
TH2 = 0xFF;
----- T2 比较模式-----
// 所有通道设为模式 0
T2CON |= 0x11;
----- T2 中断 -----
// 禁止定时器 2 溢出中断
ET2=0;
// 禁止定时器 2 外部重新加载中断
EXEN2=0;
----- 比较/捕捉通道 0 -----
// 禁止??
// 比较寄存器 CRC 设置为: 0xFF00
CRCL = 0x00;
CRCH = 0xFF;
// 禁止 CC0/ext3 中断
EX3=0;
----- 比较/捕捉通道 1 -----
// 使能比较
// 比较寄存器 CC1 设置为: 0xFF80
CCL1 = 0x80;
CCH1 = 0xFF;
// 禁止 CC1/ext4 中断
EX4=0;
----- 比较/捕捉通道 2 -----
// 禁止
// 比较寄存器 CC2 设置为: 0x0000
CCL2 = 0x00;
CCH2 = 0x00;
// 禁止 CC2/ext5 中断
EX5=0;
----- 比较/捕捉通道 3 -----
// 禁止
// 比较寄存器 CC3 设置为: 0x0000
CCL3 = 0x00;
CCH3 = 0x00;
// 禁止 CC3/ext6 中断
EX6=0;
// 设置上述所有模式, 通道 0~通道 3
CCEN = 0x08;
----- 
// 计数引脚 3.5 上的脉冲 [仅用软件]
```

```
// (用于测量当前的电动机转速)
// -----
Pulse_count_pin = 1;
Pulse_count_G = 0;
}

/*
PID_MOTOR_Control_Motor()
电动机主控制函数
*/
void PID_MOTOR_Control_Motor(void)
{
    int Error;
    int Control_i;
    // 取当前转速值 ( 0~255 )
    Speed_measured_G = PID_MOTOR_Read_Current_Speed();
    // 取所需转速值 ( 0~255 )
    Speed_required_G =
        PID_MOTOR_Get_Required_Speed();
    if (++Ticks == 100)
    {
        Speed_required_G = 200;
    }
    // 期望转速和实际转速之间的差 ( 0~255 )
    Error = Speed_required_G - Speed_measured_G;
    // 比例项
    Control_new = Controller_output_G + (Error / PID_PROPORIONAL);
    // 模数转换 (如果不需, 设置为 0)
    if (PID_INTEGRAL)
    {
        Sum_G += Error;
        Control_new += (Sum_G / (1 + PID_INTEGRAL));
    }
    // 微分项 (如果不需, 设置为 0)
    if (PID_DIFFERENTIAL)
    {
        Control_new += (Error - Old_error_G) / (1 + PID_DIFFERENTIAL);
        // 存储误差值
        Old_error_G = Error;
    }
    // 调整至 8 位范围
    if (Control_new > 255)
    {
        Control_new = 255;
        Sum_G -= Error; // 顶死保护
    }
    if (Control_new < 0)
    {
        Control_new = 0;
```



```

    Sum_G -= Error; // 顶死保护
}
// 转换为所需的 8 位格式
Controller_output_G = (tByte) Control_new;
// 更新脉宽调制设置
PID_MOTOR_Set_New_PWM_Output(Controller_output_G);
// 更新显示
PID_MOTOR_data_G[4] = CHAR_MAP_G[Speed_measured_G / 100];
PID_MOTOR_data_G[5] = CHAR_MAP_G[(Speed_measured_G % 100) / 10];
PID_MOTOR_data_G[6] = CHAR_MAP_G[Speed_measured_G % 10];
PID_MOTOR_data_G[12] = CHAR_MAP_G[Speed_required_G / 100];
PID_MOTOR_data_G[13] = CHAR_MAP_G[(Speed_required_G % 100) / 10];
PID_MOTOR_data_G[14] = CHAR_MAP_G[Speed_required_G % 10];
PID_MOTOR_data_G[20] = CHAR_MAP_G[Controller_output_G / 100];
PID_MOTOR_data_G[21] = CHAR_MAP_G[(Controller_output_G % 100) / 10];
PID_MOTOR_data_G[22] = CHAR_MAP_G[Controller_output_G % 10];
PC_LINK_O_Write_String_To_Buffer(PID_MOTOR_data_G);
}

/*-----*
 * PID_MOTOR_Get_Required_Speed()
 * 从端口和模数转换器获取速度设定
 *-----*/
tByte PID_MOTOR_Get_Required_Speed(void)
{
    // 从模数转换器取采样
    // 写 ADDATL (具体的数值无所谓) 以启动转换
    ADDATL = 0x01;
    // 等待转换完成
    // 注意: 这个演示将没有超时...
    while (BSY == 1);
    // 已得到 10 位模数转换结果
    // 返回 8 位结果
    return ADDATH;
}

/*-----*
 * PID_MOTOR_Set_New_PWM_Output()
 * 调整脉宽调制输出值
 *-----*/
void PID_MOTOR_Set_New_PWM_Output(const tByte Controller_output_G)
{
    // 改变 CCL1 的值以产生适当的脉宽调制占空比
    CCL1 = Controller_output_G;
}

/*-----*
 * PID_MOTOR_Read_Current_Speed()
 */

```

有规律地按一定的时间间隔调度此函数。

记住：最大计数值是 65536（16 位计数器）——必须确保计数不会溢出。应选择合适的调度间隔并留有一定裕度。

对高频率脉冲，需要考虑到在读计数器时计数会停止非常短的时间。

注意：第一个计数前的延迟通常应该和计数的间隔时间相同，以保证第一个计数尽可能的精确。

例如，这是正确的：

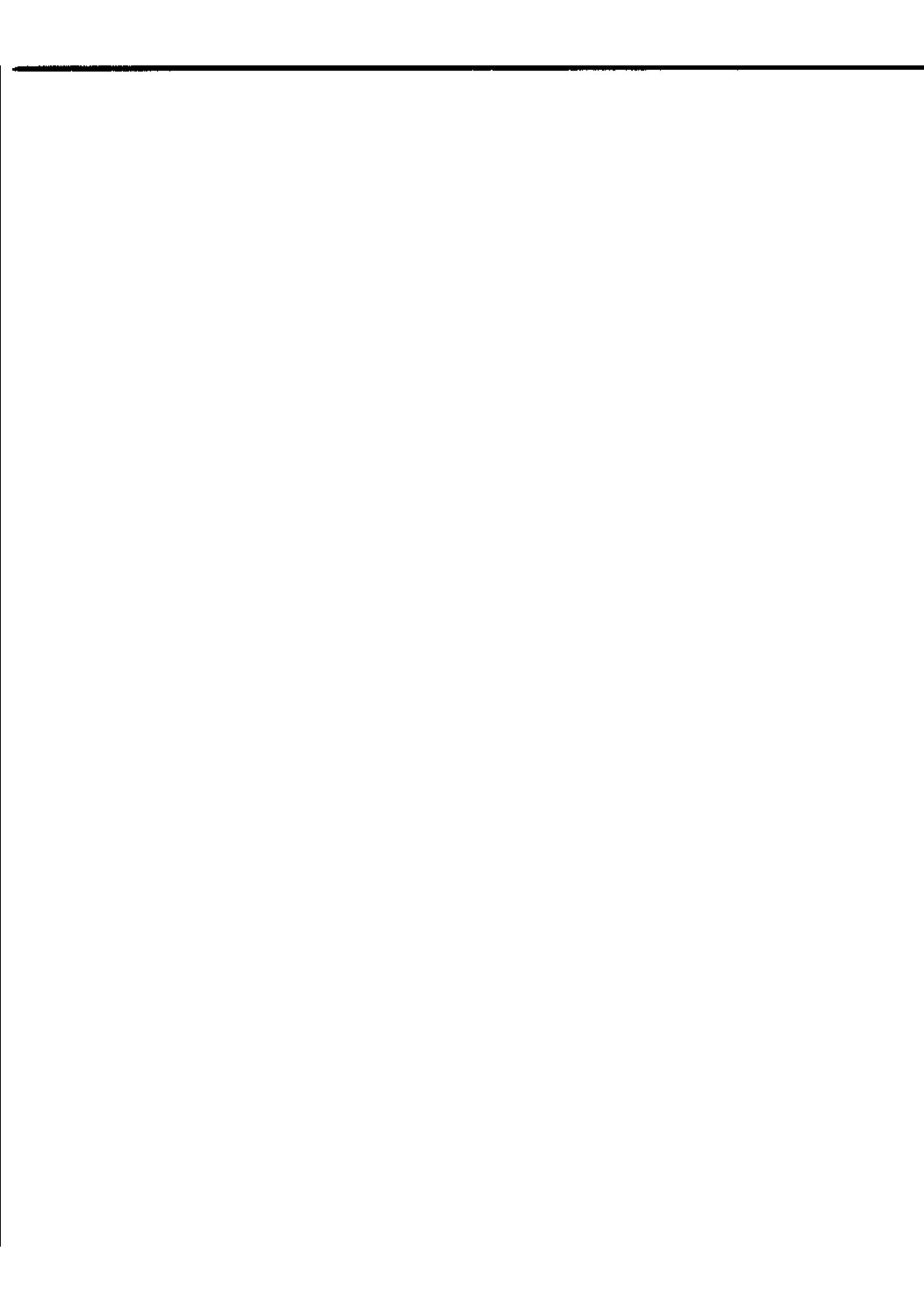
而这样会得到一个非常小的首次计数：

```
Sch_Add_Task(PID_MOTOR_Read_Current_Speed, 0, 1000);
/*-----*/
tByte PID_MOTOR_Read_Current_Speed(void)
{
    int C;
    tByte Count = Pulse_count_G;
    Pulse_count_G = 0;
    // 规格化: 0 -> 255
    C = 9 * ((int) Count - 28);
    if (C < 0)
    {
        C = 0;
    }
    if (C > 255)
    {
        C = 255;
    }
    return (tByte) C;
}
/*-----*/
PID_MOTOR_Poll_Speed_Pulse()
使用软件对指定引脚的下降沿进行计数
-此处没有使用 T0
/*-----*/
void PID_MOTOR_Poll_Speed_Pulse(void)
{
    static bit Previous_state;
    bit Current_state = Pulse_count_pin;
    if ((Previous_state == PULSE_HIGH) && (Current_state == PULSE_LOW))
    {
        Pulse_count_G++;
    }
    Previous_state = Current_state;
}
/*-----*/
----文件结束-----
/*-----*/
```

源程序清单 35.6 直流电动机 PID 控制例子的部分源程序

进阶阅读

- Atherton, D.P. (1999) 'PID controller tuning', *IEE Computing & Control Engineering Journal*, 10 (2): 44-50.
- Bennett, S. (1994) *Real-time Computer Control*, 2nd edn, Prentice Hall, New Jersey.
- Daley, S. and Liu, G.P. (1999) 'Optimal PID tuning using direct search algorithms', *IEE Computing & Control Engineering Journal*, 10 (2): 51-6.
- Dorf, R.C. and Bishop, R.H. (1998) *Modern Control Systems*, 8th edn, Addison-Wesley, CA.
- Doyle, F.J., Gatzke, E.P. and Parker, R.S. (1999) *Process Control Module: A Software Laboratory for Control Design: The MATLAB-based Process Control Guide for Chemical Engineering Professionals*, Prentice-Hall, New Jersey.
- Dutton, K., Thompson, S. and Barraclough, B. (1997) *The Art of Control Engineering*, Addison-Wesley Reading, MA.
- Franklin, G.F., Powell, J.D., and Emami-Naeini, A. (1994) *Feedback Control of Dynamic Systems*, 3rd edn, Addison-Wesley, Reading, MA.
- Franklin, G.F., Powell, J.D., and Workman, M. (1998) *Digital Control of Dynamic Systems*, 3rd edn, Addison-Wesley, CA.
- Nise, N.S. (1995) *Control Systems Engineering*, 2nd edn, Addison-Wesley, CA.
- Passino, K.M. and Yurkovich, S. (1998) *Fuzzy Control*, Addison-Wesley, CA.
- Ziegler, J.G. and Nichols, N.B. (1942) 'Optimal setting for automatic controllers', *Trans. ASME*, 64 (11), 759-68.
- Ziegler, J.G. and Nichols, N.B. (1943) 'Process lags in automatic control circuits', *Trans. ASME*, 65 (5), 433-44.

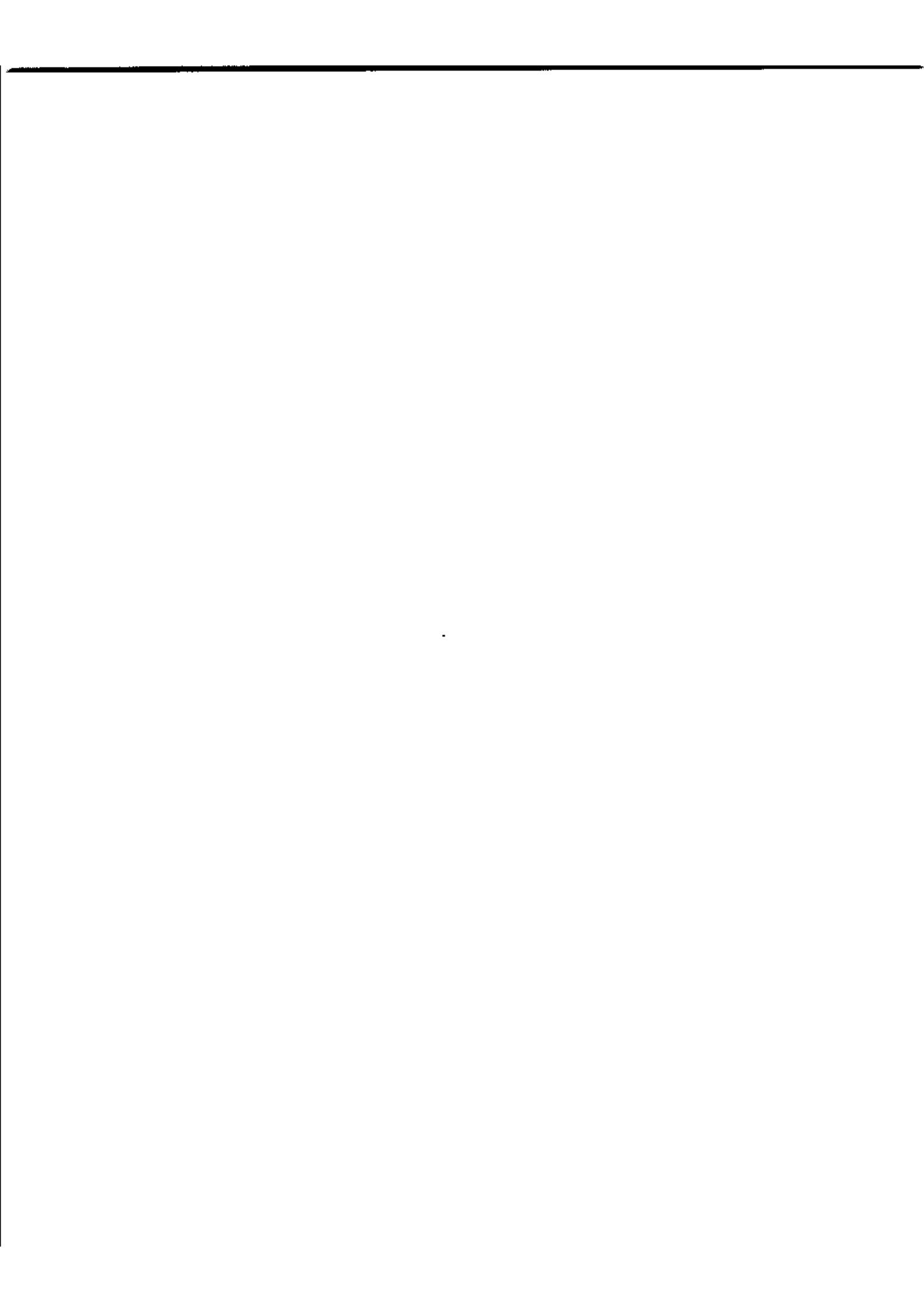


Part 8

特殊的时间触发结构

本部分通过讨论四个特殊的调度器来对本书介绍的各种模式做一个总结。事实上这些调度器全部都是合作式的，但是为满足特殊应用的需要而做了修改。例如，应用于要求低功耗或者需要非常精确的调度器定时的系统。更具体地说：

- 在第 36 章中，将讨论减少 CPU 及存储器开销，同时保持调度结构优点的方法。
- 在第 37 章中，将讨论当环境温度不可避免的产生波动时，提高调度器定时稳定性的简单而高性价比的技术。



减少系统开销

引言

在本章中，将讨论如何利用本书给出的通用调度器的灵活特性，来创建针对特殊应用需要而修改的专用调度器。本章将介绍以下模式：

- 255-时标调度器

一种设计用来运行多个任务的调度器，需要较少的存储器（和 CPU）开销。该调度器的运行方式与标准的合作式调度器一样，然而所有信息都保存在以字节为单位的变量中（而不是以字为单位）。这样，对于每个任务将减少大约 30% 的存储器开销。

- 单任务调度器

一种简化的合作式调度器，能够管理单个任务。这种非常简单的调度器能够很有效地使用硬件资源，CPU 和存储器开销最少。

- 一年调度器

一种设计用于非常低运行功耗的调度器。具体地说，设计用于通过一个小型的低成本电池供电支持一年以上运行的系统。

255-时标调度器

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。

问题

如何减少调度器所需占用的存储器？

背景知识

解决方案

正如在第14章讨论的，本书中的大多数调度器基于以下的数据结构：

```
// 如果可能的话，存储在 DATA 区，以供快速存取
// 每个任务的存储器总和是 7 个字节
typedef data struct
{
    // 指向任务的指针(必须是一个“void(void)”函数)
    void (code * pTask)(void);
    // 延迟(时标)直到函数将(下一次)运行
    // -详细说明参见 SCH_Add_Task()
    tWord Delay;
    // 在连续的运行之间的间隔(时标)。
    // -详细说明参见 SCH_Add_Task()
    tWord Period;
    // 当任务需要运行时(由调度器)加 1
    tByte RunMe;
} sTask;
```

这种数据结构支持任务的初始延迟和任务间隔最多为 65 535 个时标。这样，每个任务需要总共七个字节的存储器。

而 255-时标调度器的数据结构经过稍加改进：

```
// 如果可能的话，存储在 DATA 区，以供快速存取
// 每个任务的存储器总和是五个字节
typedef data struct
{
    // 指向任务的指针(必须是一个“void(void)”函数)
    void (code * pTask)(void);
    // 延迟(时标)直到函数将(下一次)运行
    // -详细说明参见 SCH_Add_Task()
    tByte Delay;
    // 在连续的运行之间的间隔(时标)。
    // -详细说明参见 SCH_Add_Task()
    tByte Period;
    // 当任务需要运行时(由调度器)加 1
    tByte RunMe;
} sTask;
```

这种数据结构支持任务的初始延迟和任务间隔为最多 255 个时标。这样，每个任务需要总共五个字节的存储器。

硬件资源

除了显著地节约了存储器之外，使用这种调度器还能够带来少量的性能改善。因为在 255-时标调度器中，处理器可以处理 8 位变量，比处理普通版本调度器的 16 位变量更快。

可靠性和安全性

这种调度器具有贯穿于本书讨论的合作式调度器的所有可靠性和安全性方面的特性。

可移植性

这种调度器的可移植性和贯穿于本书讨论的其他合作式调度器一样。

优缺点小结

- ◎ 提供合作式的调度环境。
- ◎ 减少对存储器的需求（和合作式调度器相比）。
- ◎ 少量的减少了 CPU 的负荷。
- ◎ 受限制的（255 个时标）初始延迟和任务间隔。

相关的模式和替代解决方案

参见合作式调度器。

例子：一个通用 255-时标调度器

在源程序清单 36.1~源程序清单 36.8 中将提供一个通用 255-时标调度器的例子。

```
/*
 *-----*
 * Port.H (v1.00)
 *
 *-----*
 * 项目 SCH_255 (参见第 36 章) 的“项目头文件”(参见第 10 章)
 *-----*
 */
// ----- Sch51.C -----
// 如果不需要错误报告, 将这一行注释掉
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P2
#endif
// ----- LED_Flas.C -----
// +5V 串接适当的电阻连接 LED 到这个引脚。
// [详细资料参见第 7 章]
sbit LED_pin = P1^5;
/*-----*
 *----- 文件结束 -----
 *-----*
```

源程序清单 36.1 通用 255-时标调度器例子的一部分

```
/*
 *-----*
 * Main.c (v1.00)
 *-----*
```

```

----- Demonstration program for:
----- 通用 255-时标调度器的演示程序。使用 T2。
----- 假定采用 12MHz 的振荡器 (-> 5ms 时标间隔)
----- ***所有定时单位为时标（而不是毫秒） ***
----- 要求的链接程序选项（详细资料参见第 14 章）
----- */

#include "Main.h"
#include "2_05_12g.h"
#include "LED_flas.h"
/* ..... */
/* ..... */
void main(void)
{
    // 设置调度器
    SCH_Init_T2();
    // 为“Flash_LED”任务作准备
    LED_Flash_Init();
    // 添加“Flash LED”任务 (1000ms 亮, 1000ms 灭)
    // -定时单位为时标 (5ms 间隔)
    // (最大的间隔/延迟是 255 个时标)
    SCH_Add_Task(LED_Flash_Update, 0, 200);
    // 开始调度器
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
----- 文件结束 -----
----- */

```

源程序清单 36.2 通用 255-时标调度器例子的一部分

```

----- 2_05_12g.h (v1.00)
----- // -详细资料参见 2_05_12g.C
----- */

#include "Main.h"
#include "SCH51a.H"
// -----公用的函数原型-----
void SCH_Init_T2(void);
void SCH_Start(void);
----- 文件结束 -----
----- */

```

源程序清单 36.3 通用 255-时标调度器例子的一部分

第36章 减少系统开销

```
/*
 2_05_12g.C (v1.00)

 *** 这里是用于标准 8051/8052 的 255-时标调度器 ***
 *** 使用 T2 定时，16 位自动重装 ***
 *** 12MHz 的振荡器 -> 5ms (精确) 时标间隔 ***
 */

#include "2_05_12g.h"
// -----公有变量声明-----
// 任务队列 (参见 Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量
//
// 将用来显示错误代码的端口以及错误代码的详细资料参见 Port.H
extern tByte Error_code_G;
/*-----*
 SCH_Init_T2()
 调度器初始化函数。准备调度器数据结构并且设置定时器以所需的频率中断
 必须在使用调度器之前调用这个函数
 *-----*/
void SCH_Init_T2(void)
{
    tByte i;
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
    // 复位全局错误变量
    // - SCH_Delete_Task () 将产生一个错误代码
    // (因为任务队列是空的)
    Error_code_G = 0;
    // 现在设置定时器 2
    // 自动重装的 16 位定时器函数
    T2CON = 0x04;      // 加载定时器 2 的控制寄存器
    T2MOD = 0x00;      // 加载定时器 2 的模式寄存器
    // 晶振假定为 12 MHz
    // 定时器 2 的精度是 0.000001 秒 (1 微秒)
    // 要求的定时器 2 溢出为 0.005 秒 (5ms)
    // -需要 5000 个定时器时标
    // 重装值为 65536 - 5000 = 60536 (十进制) = 0xEC78
    TH2      = 0xEC;      // 加载定时器 2 的高位字节
    RCAP2H   = 0xEC;      // 加载定时器 2 的重装捕捉寄存器的高位字节
    TL2      = 0x78;      // 加载定时器 2 的低位字节
    RCAP2L   = 0x78;      // 加载定时器 2 的重装捕捉寄存器的低位字节
    ET2      = 1;         // 使能定时器 2 中断
    TR2      = 1;         // 启动定时器 2
}
/*-----*
 SCH_Start()
 *-----*
```

通过允许中断来启动调度器

注意：通常在添加了所有定期的任务之后调用，从而使任务保持同步。

注意：应该只使能调度器中断!!!

```
-----*/
void SCH_Start(void)
{
    EA = 1;
}
/*-----
SCH_Update()
这是调度器的中断服务程序。其调用频率取决于 SCH_Init() 中的定时器设置
这个版本由定时器 2 中断触发：
定时器自动重装
-----*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    TF2 = 0; // 必须手工清除
    // 注意：计算单位为“时标”（不是毫秒）
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // 检测这里是否有任务
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // 任务需要运行
                SCH_tasks_G[Index].RunMe += 1; // 增加运行标志
                if (SCH_tasks_G[Index].Period)
                {
                    // 调度周期性的任务再次运行
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // 还没有准备好运行；延迟减 1
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}
/*-----
---- 文件结束 -----
-----*/
```

源程序清单 36.4 通用 256 时标调度器例子的一部分

```
-----*
SCH51a.h (v1.00)
```

```

-----详细资料参见 SCH51a.C -----
/*
#ifndef _SCH51_H
#define _SCH51_H
#include "Main.h"
// -----公用数据类型声明-----
// 如果可能的话，存储在 DATA 区，以供快速存取
// 每个任务的存储器总和是五个字节
typedef data struct
{
    // 指向任务的指针（必须是一个“void (void) ”函数）
    void (code * pTask)(void);
    // 延迟（时标）直到函数将（下一次）运行
    // -详细说明参见 SCH_Add_Task()
    tByte Delay;
    // 在连续的运行之间的间隔（时标）
    // -详细说明参见 SCH_Add_Task()
    tByte Period;
    // 当任务需要运行时（由调度器）加 1
    tByte RunMe;
} sTask;
// -----公用的函数原型-----
// 调度器内核函数
void SCH_Dispatch_Tasks(void);
tByte SCH_Add_Task(void (code*) (void), const tByte, const tByte);
bit SCH_Delete_Task(const tByte);
void SCH_Report_Status(void);
// -----公用的常数-----
// 程序运行的任一时刻所需的任务的最大数量
// 每个新建项目都必须调整
#define SCH_MAX_TASKS (3)
#endif
/*-----文件结束-----*/
*/

```

源程序清单 36.5 通用 255-时标调度器例子的一部分

```

/*
SCH51a.C(v1.00)

***这里是调度器内核函数***
***255-时标版本***
---这个函数可以用于所有的 8051 芯片
***SCH_MAX_TASKS 必须由用户设置***
---参见“Sch51.h ”
***包括省电模式***
---必须确认省电模式被修改以适用于所选定的芯片(通常只有在使用扩展 8051, 诸如 c515c、c509
---等等才需要) ---
*/

```

```

#include "Main.h"
#include "Port.h"
#include "Sch51a.h"
// -----公有变量定义-----
// 任务队列
sTask SCH_tasks_G[SCH_MAX_TASKS];
// 用来显示错误代码
// 错误代码的详细资料参见 Main.H
// 错误显示端口的详细资料参见 Port.H
tByte Error_code_G = 0;
// -----私有函数原型-----
static void SCH_Go_To_Sleep(void);
// -----私有变量-----
// 记住自从上一次记录错误以来的时间(见下文)
static tWord Error_tick_count_G;
// 上次的错误代码(在1分钟之后复位)
static tByte Last_error_code_G;
/*-----*/
SCH_Dispatch_Tasks()
这是“调度”函数。当一个任务(函数)需要运行时, SCH_Dispatch_Tasks()将运行它。
这个函数必须被主循环(重复)调用。
*-----*/
void SCH_Dispatch_Tasks(void)
{
    tByte Index;
    // 调度(运行)下一个任务(如果有任务就绪)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        if (SCH_tasks_G[Index].RunMe > 0)
        {
            (*SCH_tasks_G[Index].pTask)();
            // 运行任务
            SCH_tasks_G[Index].RunMe -= 1; // 复位/降低 RunMe 标志
            // 周期性的任务将自动的再次运行
            // -如果这是个“单次”任务, 将它从队列中删除
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }
    // 报告系统状况
    SCH_Report_Status();
    // 这里调度器进入空闲模式
    SCH_Go_To_Sleep();
}
/*-----*/
SCH_Add_Task()
使任务(函数)每隔一定间隔或在用户定义的延迟之后运行
Fn_P-将被调度的函数的名称。

```

注意：所有被调度的函数必须是“void, void”即，函数必须没有参数，并且返回类型为 void。

DELAY-在任务第一次被运行之前的间隔（时标）

PERIOD-如果“PERIOD”为 0，该函数将只被调用一次，由“DELAY”确定调用的时间。如果 PERIOD 非 0，那么该函数将按 PERIOD 的值确定的间隔被重复调用（下面的例子将有助于理解这些）。

返回值：

返回被添加任务在任务队列中的位置。如果返回值是 SCH_MAX_TASKS，那么该任务不能被加到队列中（空间不够）。如果返回值< SCH_MAX_TASKS，那么该任务被成功添加。

注意：如果以后要删除任务，将需要这个返回值，参见 SCH_Delete_Task()。

例子：

```
Task_ID = SCH_Add_Task(Do_X,1000,0);
使函数 Do_X() 在 1000 个调度器时标之后运行一次。
```

```
Task_ID = SCH_Add_Task(Do_X,0,1000);
使函数 Do_X() 每隔 1000 个调度器时标运行一次。
```

```
Task_ID = SCH_Add_Task(Do_X,300,1000);
使函数 Do_X() 每隔 1000 个调度器时标运行一次。任务将首先在 T = 300 个时标时被执行，然后
```

1300 个时标、2300 个时标等等；

```
-----*/
tByte SCH_Add_Task(void * pFunction)(),
    const tByte DELAY,
    const tByte PERIOD)
{
    tByte Index = 0;
    // 首先在队列中找到一个空隙（如果有的话）
    while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
    {
        Index++;
    }
    // 是否已经到达队列结尾？
    if (Index == SCH_MAX_TASKS)
    {
        // 任务队列已满
        //
        // 设置全角错误变量
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
        // 同时返回错误代码
        return SCH_MAX_TASKS;
    }
    // 如果能运行到这里，说明任务队列中有空间
    SCH_tasks_G[Index].pTask = pFunction;
    SCH_tasks_G[Index].Delay = DELAY;
    SCH_tasks_G[Index].Period = PERIOD;
    SCH_tasks_G[Index].RunMe = 0;
    return Index; // 返回任务的位置（以便以后删除）
}
/*-----
SCH_Delete_Task()
// 从调度器删除任务。注意：并不从存储器中删除相关的函数，仅仅是不再由调度器调用这个任务。
```

TASK_INDEX-任务索引。由 SCH_Add_Task() 提供。

返回值：返回 ERROR 或者返回 NORMAL

```
-----*/
bit SCH_Delete_Task(const tByte TASK_INDEX)
{
    bit Return_code;
    if (SCH_tasks_G[TASK_INDEX].pTask == 0)
    {
        // 这里没有任务
        //
        // 设置全局错误变量
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;
        // 同时返回错误代码
        Return_code = RETURN_ERROR;
    }
    else
    {
        Return_code = RETURN_NORMAL;
    }
    SCH_tasks_G[TASK_INDEX].pTask    = 0x0000;
    SCH_tasks_G[TASK_INDEX].Delay    = 0;
    SCH_tasks_G[TASK_INDEX].Period   = 0;
    SCH_tasks_G[TASK_INDEX].RunMe   = 0;
    return Return_code;           // 返回状态
}
-----*/
```

SCH_Report_Status()

用来显示错误代码的简单的函数。

这个版本在连接到端口的 LED 上显示错误代码。

如果需要的话，可以修改为通过串行连接等方式来报告错误。

错误只在有限的时间内显示（1ms 时标间隔时，60000 时标= 1 分钟）

此后错误代码被复位为 0。

这些代码可以很容易的修改为“永远”显示最近的错误。这在系统中可能更为合理。

更加详尽的资料请参见第 10 章。

```
-----*/
void SCH_Report_Status(void)
{
#ifdef SCH_REPORT_ERRORS
    // 只在需要报告错误时适用
    // 检查新的错误代码
    if (Error_code_G != Last_error_code_G)
    {
        // 假定 LED 采用负逻辑
        Error_port = 255 - Error_code_G;
        Last_error_code_G = Error_code_G;
        if (Error_code_G != 0)
        {
            Error_tick_count_G = 60000;
        }
    }
}
-----*/
```

```

    else
    {
        Error_tick_count_G = 0;
    }
}
else
{
    if (Error_tick_count_G != 0)
    {
        if (--Error_tick_count_G == 0)
        {
            Error_code_G = 0; // 复位错误代码
        }
    }
}
#endif
}

/*
SCH_Go_To_Sleep()
本调度器在时钟时标之间将进入“空闲模式”来节省功耗。下一个时钟时标将使处理器返回到正常操作状态。
注意：如果这个函数由宏来实现，或简单的将这里的代码粘贴到“调度”函数中，可以带来少量的性能改善。
然而，通过采用函数调用的方式来实现，可以在开发期间更容易的使用 Keil 硬件模拟器中的“性能分析器”来估计调度器的性能。这方面的例子参见第 14 章。
***如果使用看门狗的话，可能需要禁止这个功能***
***根据硬件的需要修改***
*/
void SCH_Go_To_Sleep()
{
    PCON |= 0x01; // 进入空闲模式（通用 8051 版本）
    // 进入空闲模式需要两个连续的指令
    // 在 80c515/80c505 上，用来避免意外的触发
    // PCON |= 0x01; // 进入空闲模式 (#1)
    // PCON |= 0x20; // 进入空闲模式 (#2)
}
/*
----文件结束-----
*/

```

源程序清单 36.6 通用 255-时标调度器例子的一部分

```

/*
LED_flas.H (v1.00)
-----详细资料参见 LED_flas.C -----
// -----公用的函数原型-----
void LED_Flash_Init(void);
void LED_Flash_Update(void);

```

```
/*-----  
---- 文件结束  
-----*/
```

源程序清单 36.7 通用 255-时标调度器例子的一部分

```
/*-----  
 LED_flas.C (v1.00)  
-----  
 // 用于调度器简单的“闪烁 LED”的测试函数  
-----*/  
#include "Main.h"  
#include "Port.h"  
#include "LED_flas.h"  
// -----私有变量定义-----  
static bit LED_state_G;  
/*-----  
 LED_Flash_Init()  
 -参见下文。  
-----*/  
void LED_Flash_Init(void)  
{  
    LED_state_G = 0;  
}  
/*-----  
 LED_Flash_Update()  
 在指定端口引脚上闪烁 LED(或产生脉冲给蜂鸣器，等等)。  
 必须按需要的闪烁频率的两倍计时。这样，对于 1Hz 的闪烁(0.5 秒亮，0.5 秒灭)  
 必须以 2Hz 计时  
-----*/  
void LED_Flash_Update(void)  
{  
    // 使 LED 从灭变亮(反之亦然)  
    if (LED_state_G == 1)  
    {  
        LED_state_G = 0;  
        LED_pin = 0;  
    }  
    else  
    {  
        LED_state_G = 1;  
        LED_pin = 1;  
    }  
}  
/*-----  
---- 文件结束  
-----*/
```

源程序清单 36.8 通用 255-时标调度器例子的一部分

进阶阅读

单任务调度器

适用场合

- 用8051系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。

问题

如何使用最少的存储器和CPU资源创建只运行一个任务的系统？

背景知识

解决方案

合作式调度器可用作单任务系统的基础，常常用它来实现这种系统。然而，有时候系统很简单，而需要从8051微控制器所具有的有限的CPU和存储器资源中榨取尽可能多的可用资源。在这种情况下，使用完整的合作式调度器环境可能并不合适。

有一种与本书中的大多数模式兼容的简单解决方案：将系统基于单个任务并且通过实现一个定时器中断服务程序来直接调用这个任务。

源程序清单36.9说明了该方法。

```
-----*/  
Main.c  
-----*/  
简单的定时器中断服务程序的演示  
-----*/  
#include<AT89S53.h>  
#define INTERRUPT_Timer_2_Overflow 5  
// 函数原型  
// 注意：中断服务程序不被直接调用，因此不需要原型  
void Timer_2Init(void);  
/*-----*/  
void main(void)  
{  
    Timer_2_Init(); // 设置定时器2  
    EA=1;           // 允许所有中断  
    while(1);       // 一个空的超级循环  
}  
/*-----*/  
void Timer_2_Init(void)
```

```

{
// 定时器 2 配置为 16 位定时器
// 当溢出时自动重装
//
// 本代码（通用 8051/52）假定用在振荡周期为 12MHz 的系统。
// 因此定时器 2 的精度是 1.000 微秒
// （详细资料参见第 11 章）
//
// 重装值为 FC18（十六进制）= 64536（十进制）
// 当定时器（16 位）到达 65536（十进制）时溢出
// 这样，在这种设置下，定时器将每隔 1ms 溢出
T2CON = 0X0.4; // 加载定时器 2 的控制寄存器
T2MOD = 0X0.0; // 加载定时器 2 的模式寄存器
TH2 = 0XFC; // 加载定时器 2 的高位字节
RCAP2H = 0XFC; // 加载定时器 2 的重装捕捉寄存器的高位字节
TL2 = 0X18; // 加载定时器 2 的低位字节
RCAP2L = 0X0.18; // 加载定时器 2 的重装捕捉寄存器的低位字节
// 使能定时器 2 中断，并且将调用中断服务程序
// 每当定时器溢出时——如下
ET2 = 1;
// 启动定时器 2 运行
TR2 = 1;
}

/*-----*/
void X(void) interrupt INTERRUPT_Timer_2_Overflow
{
// 每隔 1ms 调用这个中断服务程序
// 所需的代码放在这里
}
/*-----文件结束-----*/
*/

```

源程序清单 36.9 使用定时器中断服务程序定期调用任务的系统结构

硬件资源

主要从三个方面来讨论有关硬件资源的问题：定时器、存储器和 CPU 开销。

定时器

本模式需要一个硬件定时器。如果可能的话，应该使用有自动重装能力的 16 位定时器，比如定时器 2（详细资料参见第 13 章）。

存储器和 CPU 开销

调度器将耗费不太大的 CPU 资源，少于用超级循环（具有这种基本结构的所有缺点）来实现该系统的开销，用高级语言来实现该系统通常没有更有效的方法。

可靠性和安全性

如果不是将多任务设计“硬塞进”这种单任务框架的话，这种方法将既安全又可靠。

优缺点小结

- ⊕ 是一种定期运行单个任务的有效环境。
- ⊖ 只适用于可以彻底使用单个任务实现的系统。

相关的模式和替代解决方案

这种“简化”调度器的主要替代方案是超级循环结构。

如果和多状态任务配合使用时，单任务调度器尤其有效。

例子：估计单任务调度器的开销

在源程序清单 36.10~源程序清单 36.11 中说明单任务调度器的使用。

```
/*
 *-----*
 * Port.H (v1.00)
 *
 *-----*
 * 项目 ONE_TASK (参见第 36 章) 的“项目头文件”(参见第 10 章)
 *-----*
 */
// ----- Sch51.C -----
// 如果不需要错误报告，将这一行注释掉
#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// 将用来显示错误代码的端口
// 只在报告错误时使用
#define Error_port P1
#endif
// ----- LED_Flas.C -----
// +5V 串接适当的电阻连接 LED 到这个引脚。
// [详细资料参见第 7 章]
sbit LED_pin = P1^5;
/*
 *-----*
 *   文件结束
 *-----*
 */
```

源程序清单 36.10 单任务调度器的部分实现

```
/*
 *-----*
 * Main.c (v1.00)
 *
 *-----*
 * 单任务调度器的演示程序-详细资料参见第 36 章
 *-----*
 #include "Main.H"
 #include "Port.H"
```

```

#define INTERRUPT_Timer_2_Overflow 5
// 全局变量
static tByte LED_state_G;
// 函数原型
// 注意：中断服务程序不被直接调用，因此不需要原型
void Timer_2_Init(void);
void LED_Flash_Init(void);
void Go_To_Sleep(void);
/* -----
void main(void)
{
    Timer_2_Init();      // 设置定时器 2
    LED_Flash_Init();    // 准备闪烁 LED
    EA = 1;              // 允许所有中断
    while(1)             // 超级循环
    {
        Go_To_Sleep();   // 进入空闲模式以节省功耗
    }
}
/* -----
void Timer_2_Init(void)
{
    // 定时器 2 配置为 16 位定时器
    // 当溢出时自动重装
    //
    // 本代码（通用 8051/52）假定用在振荡周期为 12MHz 的系统
    // 因此定时器 2 的精度是 1.000 微秒
    // （详细资料参见第 11 章）
    //
    // 重装值为 FC18（十六进制）= 64536（十进制）
    // 当定时器（16 位）到达 65536（十进制）时溢出
    // 这样，在这种设置下，定时器将每隔 1ms 溢出
    T2CON = 0x04;        // 加载定时器 2 的控制寄存器
    T2MOD = 0x00;        // 加载定时器 2 的模式寄存器
    TH2   = 0xFC;         // 加载定时器 2 的高位字节
    RCAP2H = 0xFC;        // 加载定时器 2 的重装捕捉寄存器的高位字节
    TL2   = 0x18;         // 加载定时器 2 的低位字节
    RCAP2L = 0x18;        // 加载定时器 2 的重装捕捉寄存器的低位字节
    // 使能定时器 2 中断，并且将调用中断服务程序
    // 每当定时器溢出时——如下
    ET2   = 1;
    // 启动定时器 2 运行
    TR2   = 1;
}
/* -----
LED_Flash_Init()
如下。
* -----
void LED_Flash_Init(void)
{

```

```

LED_state_G = 0;
}
/*
LED_Flash_Update()
在指定端口引脚上闪烁 LED (或产生脉冲给蜂鸣器, 等等)。
代码假定这个函数将每隔 1ms 调用,
LED 将以 0.5Hz 闪烁 (亮 1 秒, 灭 1 秒)
*/
void LED_Flash_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
// 每隔 1ms 调用这个中断服务程序
// -只需每隔一秒刷新 LED
static data tWord Call_count;
TF2 = 0; // 复位 T2 标志
if (++Call_count < 1000)
{
    return;
}
Call_count = 0;

// 使 LED 从灭变亮 (反之亦然)
// (每秒重复)
if (LED_state_G == 1)
{
    LED_state_G = 0;
    LED_pin = 0;
}
else
{
    LED_state_G = 1;
    LED_pin = 1;
}
}

/*
Go_To_Sleep()
该单任务调度器在时钟时标之间将进入“空闲模式”来节省功耗。
下一个时钟时标将使处理器返回到正常操作状态。
注意：如果这个函数由宏来实现，或简单的将这里的代码粘贴到“调度”函数中，
可以带来少量的性能改善。
然而，通过采用函数调用的方式来实现，可以在开发期间更容易的使用 Keil 硬件模拟器中的
“性能分析器”来估计调度器的性能。这方面的例子参见第 14 章。
***如果使用看门狗的话，可能需要禁止这个功能***
***根据硬件的需要修改***
*/
void Go_To_Sleep(void)
{
PCON |= 0x01; // 进入空闲模式 (通用 8051 版本)
// 进入空闲模式需要两个连续的指令
// 在 80c515/80c505 上，用来避免意外的触发
// PCON |= 0x01; // 进入空闲模式 (#1)
// PCON |= 0x20; // 进入空闲模式 (#2)
}

```

```

    }
/*
----- 文件结束 -----
*/

```

源程序清单 36.11 单任务调度器的部分实现

如果将这个例子中的调度器开销（如图 36.1 所示）与第 14 章中的单任务、12MHz 的 8051 的相应数据（例如图 14.4）相比，可以看到这个单任务版本更加有效率。具体地说，即使在基本 8051 上，单任务调度器也将允许使用 0.1ms 时标间隔。



图 36.1 使用 Keil 硬件模拟器来估计运行在一个 12MHz（每个指令 12 个振荡周期）的 8051 上的 1ms 时标的单任务调度器占用的 CPU 开销

注意，试验显示 CPU 是 97% 空闲的，因此可允许的任务最大运行时间约为 0.97ms。

进阶阅读

一年调度器

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。
- 该系统由电池供电。

问题

如何创建一个由电池供电的系统，在更换电池之前可以运行长达 12 个月？

背景知识

在这个模式中，将讨论如何使用调度器来创建由电池供电，在更换电池之前可以运行长达

12个月的嵌入式系统。

在本节中将提供一些主要的详细背景资料。

原电池和蓄电池

当我们谈论电池时，指的是一种通过一个或多个原电池将化学能转换为电能的电化学器件^①。原电池是一种由两个电极（阳极和阴极）和电解质溶液组成的相对简单的装置。第一块这样的电池由意大利人 Alessandro Volta 于 1800 年发明（如图 36.2 所示），该电池由铜棒阴极、锌棒阳极和稀硫酸电解液组成。

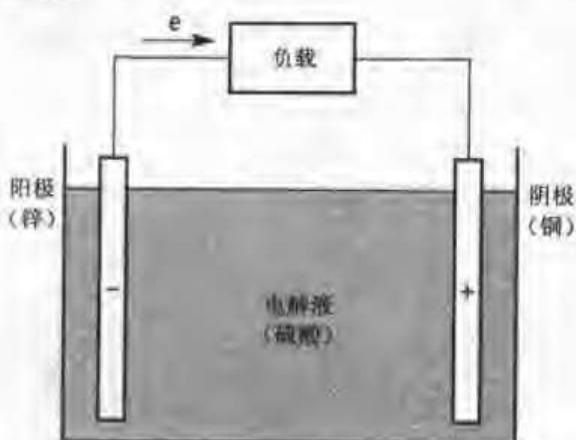


图 36.2 电池的最初形式：由 Alessandro Volta 于 1800 年发明的原电池

简而言之，该电池运行如下：锌阳极中的金属被氧化（即“腐蚀”或者“溶解”）进入电解质溶液，而电解液中的氢分子储存在阴极上。当阳极完全被氧化时（或者阴极完全被还原时），化学反应将停止而认为电池被放电。

有时候，有可能对电池“重新充电”。再充电通常是在电极两端加电压，使电池的化学过程反向。然而，一些化学反应难以反向或者根本就不可能。反应不可逆的电池通常被称为原电池，而反应可逆的电池通常被称为蓄电池。虽然有时小心操作也可以对原电池实现部分的再充电，但是这种方法往往效率很低而且很危险。

电池的特性

现代电池设计中使用了很多种不同的电极和电解液组合。这些组合导致电池具有不同的容量。电池容量常常通过“毫安时”(mAh)来表示。例如，某个电池的容量为 1 000mAh。这意味着，在理想条件下，该电池在放电到不再能够为系统提供能源之前，可以持续 1 小时产生 1 000mA 电流。或者，这个理想电池在到达放电状态之前可以持续 1 000 小时产生 1mA 电流。实际上，电池并不能在所有输出电流下都能达到同样的额定功率。例如，1.5V (D型) 碱性电池一般在（连续）10mA 输出时具有 10 000mAh 的额定功率，而在 100mA 输出时容

^① 严格说，电池由至少两个原电池单元组成（“多个单元组合而成电池”）。然而，通常使用该术语来描述包含一个或多个电池单元的器件。

量减少为 8000mAh，在 1000mA 输出时减少为 4000mAh。当然，许多嵌入式系统不会连续运行，在使用过程中对电流的需求将不断地变化。因此，对电流需求变化很大的设备中的电池使用寿命做出预计将特别困难，惟一实用的解决方案是从厂家发布的数据手册估计可能的电流需求，然后在真实环境下对系统样机进行测试。

电池的第二个主要特性是放电曲线（如图 36.3 所示）。理论上，在有关微控制器的系统中，希望得到平坦的放电曲线，这样电池在其工作寿命内将保持输出电压不变。如果放电特性不平坦而是倾斜的，电池电压可能很快就低于微控制器运行所需的电平。

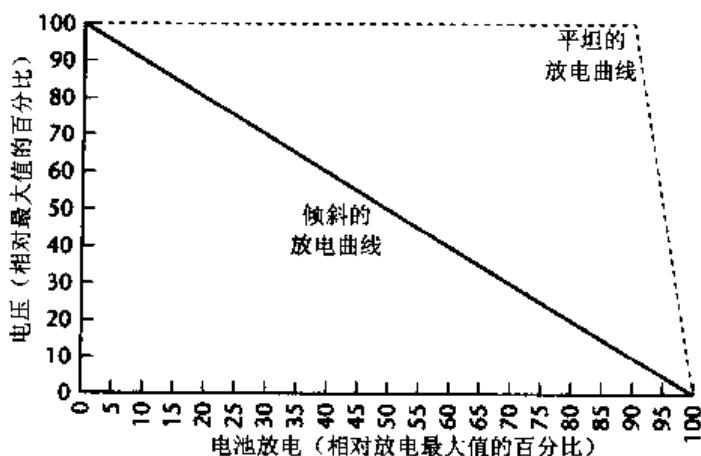


图 36.3 理想化的“平坦”和“倾斜”放电特性

注意，实际上大多数电池的特性在这两个极端之间。

碱性电池

碱性电池在现代嵌入式系统中使用得特别广泛。这种电池非常通用，因为和传统的 Leclanche 干电池相比（普通电池），它具有更平坦的放电曲线和更大的电池容量。

普通的 1.5V 碱性电池运行时间的典型数据在表 36.1 中给出（由 Duracell 的数据手册改编而来）。

表 36.1 碱性电池的典型工作寿命

电池类型	平均电流	工作寿命（小时）
碱性 AAA	245	3
	40	30
	15	80
碱性 AA	275	5
	50	50
	15	200
碱性 D 型	475	20
	235	50
	115	130

两个有用的简单参考：

- 碱性电池的储藏寿命为 2~4 年。
- AA 电池在全世界都通用并且随处可得。这种电池适用于很多系统，例如，无所不在的 Duracell 的 MN1500 具有 1850mAh 的额定值。工作在小电流（平均大约 0.3mA），这种电池的寿命至少为一年。

获得所需的电压

每个电池提供 1.5V（标称），因此四个电池将提供 6V。注意，这并不会改变电流的额定值。1 个 MN1500 AA 电池具有 1.5V、1850mAh 的额定值，而四个这样的电池具有 6V、1850mAh 的额定值。

解决方案

这里将集中讨论使用 AA 碱性电池作为电源，因为使用这种电池是一种通用而有效的方案。由于“背景知识”中给出的原因，要实现该方案需要将消耗的平均电流减少到 0.3mA。

正如在第 3 章中说明的（参见表 3.1），在正常运行情况下，标准 8051 微控制器的电流消耗约为 15mA。在这种电流下，电池将持续 200 小时左右——大约 8 天。即使系统大多数时间处于空闲模式，电流消耗也将达到大约 5mA。理想情况下，系统几乎可以运行一个月。

然而，这些数据并不足以代表所有的可能。特别是忽略了电流消耗在很大程度上取决于振荡器频率和工作电压这一事实。例如，CD 上包含的 Atmel 的 89553 的数据手册表明其电流消耗随电压和振荡器频率而变。更具体地说，89S53 或者各种其他 8051 芯片的数据手册都表明，为了获得平均 0.3mA 的电流消耗，必须运行在 3V 左右的电压，以及非常低的振荡器频率下（大约 1MHz，尽可能的低）。此外，必须使系统在大多数时间内都处于空闲模式。

获得所需振荡器频率的一种高性价比的方法是使用表晶。这种器件便宜，小并且随处可得，且具有 32.768kHz 的频率。注意，要使用这种频率，必须使用可以运行在这种频率范围内的 8051，需要核对数据手册。

使用 AAA 电池

在这个模式中详细讨论的 AA 电池具有 23.6 克或者 0.83 盎司的重量（Duracell 的 MN1500 的数据手册）。要减少系统的重量（和体积），可以考虑使用 AAA 电池。这种电池具有 11.0 克或者 0.39 盎司的重量（Duracell 的 MN2400 的数据手册），这种电池的容量为 1150mAh（Duracell 的 MN2400 的数据手册）。

1150mAh 的容量当平均电流约为 0.13mA 时，使用可以超过一年。目前，使用精简 8051 以及较新的 8051 芯片（尤其是由 Philips 生产的），可以达到这样的电流消耗水平。正如（CD ROM 上包含的）数据手册表明的，Philips 的 87LPC764 在活动和空闲状态都具有非常低的电流消耗，尤其是工作在较低频率时。

从数据手册可以看到，即使是在正常运行模式下（再次假设使用 32kHz 的表晶），87LPC768 也只消耗大约 0.1mA，在空闲模式下，下降约为 0.05mA。如果不需要驱动大功率的外部模块，

那么以这种芯片作为轻便的嵌入式系统的基础十分理想。

使用 9V 电池

另一种常用的轻便电池是 9V “无线电” 电池，这种电池的终端接插件很有用。其容量约为 500 mAh (Duracell 的 MN1604 的数据手册)，即平均电流为 0.06mA 时可以超过一年。最新的精简 8051 (前面说明的) 差不多可以满足这种要求。然而，在大部份基于 8051 的系统中，这种电池的寿命很难长于六个月。

使用 D 型电池

如果需要提供比 AA 电池更长的工作寿命及更大的电流，可以采用 D 型电池的替代方案。这种相对较大的电池容量约为 15 000mAh (Duracell 的 MN1300 的数据手册)。需要工作超过一年时，这相当于 1.7mA 的平均电流，超过两年时，相当于 0.8mA。

嵌入式系统中使用 D 型电池可以可靠地获得两年寿命，当要求的电流非常有限时甚至可以使用 3 年。对于那些不要求便于携带的系统这是一种不错的解决方案。

D 型电池尤其适用于诸如长期自动数据记录方面的系统。虽然 D 型电池的尺寸比较大、重量比较重，然而其替代方案铅酸蓄电池和太阳能充蓄电池的尺寸更大、更复杂而且更加昂贵。

使用锂电池

锂电池的推出相对较晚，它具有许多很好的特性，用作由电池供电的嵌入式系统的原电池非常合适。

这种电池的主要特性有：

- ⑤ 锂电池具有极低的自放电速度，这意味着在储藏状态下它们能够保存许多年（即使十年也很普通）。
- ⑥ 在使用过程中，锂电池能够提供同样大小的碱性电池三倍以上的时间。这使它们很适合用作台式计算机的 CMOS 备用电池。
- ⑦ 锂电池的重量大约为相应碱性电池的 30% 以下。
- ⑧ 与大多数其他的电池技术不同，锂电池能够在很宽的温度范围内有效运行。在 0°C 下运行时，碱性电池大约为正常水平的 50%~60%，而锂电池在该温度下仍能正常运行。即使在 -20°C 以下的温度，锂电池也能提供正常水平的 80% 左右，而碱性电池在这种温度下将无法使用。
- ⑨ 锂电池为标准 (AA) 尺寸、1.5V。
- ⑩ 因为在与水接触时锂将产生剧烈反应，所以基于这种金属的电池有安全性问题。最新的设计大大缓解了这种问题，然而必须了解这种潜在的安全性问题。
- ⑪ 锂电池不适用于大电流的系统（例如，驱动电动机或者其他类似的负载）。
- ⑫ 锂电池的成本大约是同样大小的碱性电池的两倍，因此可能并不是经济合算的。
- ⑬ 目前，锂电池并不像碱性电池一样随处可得。

总的说来，由于有这些特性，随着价格的下降，锂电池将不可避免地在嵌入式系统中越来

越常见。

使用蓄电池

可再充电的电池在许多嵌入式系统很有用，比如移动电话。然而它们的电能无法保持足够长的时间以满足这里模式的需要。

硬件资源

这个模式只适用于需要很有限 CPU 处理，大部分使用时间都处于空闲模式的系统。

可靠性和安全性

典型的一年应用系统将每隔（比方说）200ms 从空闲模式中唤醒一次，以检查某个开关或者完成其他的处理。当处于空闲模式下时，将无法对外部事件做出反应。

这类结构常见于电视遥控器等应用中，并且适用于类似这方面的应用。这种结构不适用于要求反应非常迅速并与安全相关的系统。

可移植性

这是一种合作式调度器，可移植性与合作式调度器一样。

优缺点小结

- ⑤ 将电池消耗减少至一个较低的水平。
- ⑥ 由于时钟频率较低，处理能力有限。
- ⑦ 可能不适用于与安全相关的系统。

相关的模式和替代解决方案

这个模式常常可以与 255-时标调度器或者单任务调度器一起使用，以更进一步减少所需的资源。

例子：自动照明

要求开发一种由电池供电的橱柜（壁橱）照明，要求如下运行：

- 一旦按下开关，点亮一个小灯泡。
- 再次按下开关，熄灭该灯泡。

当然，这些要求并不需要借助于使用微控制器就可以非常容易地实现。然而，很多人在向橱柜中放入物品后会忘记关掉照明。于是，电池将在几个小时后耗尽。在这类系统中，电池的价格将在成本中占有相当大的比重，而更换电池的频率也是需要考虑的因素之一。

因此希望创建一个运行如下的系统：

- 当照明熄灭时按下开关，打开照明。

- 当照明打开时按下开关，关闭照明。
- 如果灯泡被点亮而不再按下开关，照明将在 30 秒之后熄灭。
- 电池寿命应该“尽可能的长”。

图 36.4 展示了一种可行的硬件设计。

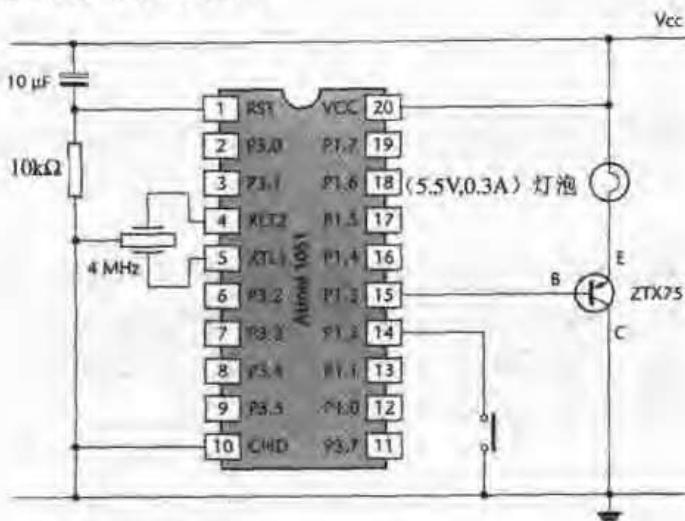


图 36.4 自动照明例子的硬件原理图

整个系统的代码在源程序清单 36.12 中给出。

```
/*
Main.c (v1.00)

自动照明的例子
*/
#include "Main.h"
// -----端口引脚-----
// 在这个简单的(单文件)例子中省略 Port.h
// 不使用引脚 1^0、1^1——没有内部上拉电阻
sbit Switch_pin_G = P1^2;
sbit Light_pin_G = P1^3;
// -----私有函数原型-----
// 函数原型
// 注意：中断服务程序不被直接调用，因此不需要原型
void Timer_1_Init(void);
void Timer_1_Manual_Reload(void);
void Light_Init(void);
// -----私有的常数-----
#define SWITCH_PRESSED 0
#define LIGHT_ON 0
#define LIGHT_OFF 1
// -----私有变量定义-----
```

```

static tByte Switch_count_G = 0;
static tByte Auto_switch_off_count_G = 0;
static tByte Switch_blocked_G = 0;
static bit LED_state_G = 0;
static tByte Call_count_G = 0;
/* -----
void main(void)
{
    Timer_1_Init(); // 设置定时器 2
    Light_Init(); // 准备闪烁 LED
    EA = 1; // 全局使能中断
    while(1)
    {
        PCON |= 0x01; // 进入睡眠（空闲模式）
    }
}
/* -----
void Timer_1_Init(void)
{
    // 定时器 1 配置为 16 位定时器
    // 当溢出时手工重装
    TMOD &= 0x0F; // T1 需要的位置 1 (T0 不变)
    TMOD |= 0x10; // 清除 T1 所有位 (T0 不变)
    // 设置定时器重装值
    Timer_1_Manual_Reload();
    // 使能定时器 1 中断
    ET1 = 1;
}
/* -----
Timer_1_Manual_Reload()
这个“一年调度器”使用（手动重装）16 位定时器、
手动重装意味着所有定时是近似的。
这个调度器不适用于要求精确定时的应用场合！
在这个函数中执行定时器重装。
*/
void Timer_1_Manual_Reload(void)
{
    // 停止定时器 1
    TR1 = 0;
    // 本代码（通用 8051/52）假定用在振荡周期为 4 MHz 的系统
    // 这样定时器 1 的精度是 0.000003 秒
    // （详细资料参见第 11 章）
    //
    // 希望每隔（大约）200ms 产生一个中断：
    // 这需要 0.2/0.000003 定时器增量
    // 即 66666 定时器增量
}

```

```

// 重装值 0x00 对应 65536 个增量，对于这里的要求来说已经足够接近（大约 2% 的偏差）
TL1 = 0x00;
TH1 = 0x00;
// 启动定时器 1
TR1 = 1;
}
/*-----*
Light_Init()
*-----*/
void Light_Init(void)
{
    Switch_count_G = 0;
    Auto_switch_off_count_G = 0;
    Switch_blocked_G = 0;
    // 向开关引脚写 1 (设置为高用于读取)
    Switch_pin_G = 1;
}
/*-----*
Check_Switch()
*-----*/
void Check_Switch(void) interrupt INTERRUPT_Timer_1_Overflow
{
    // 这个函数是通断开关模式的具体实现
    // 如果照明已打开，“Auto_switch_off_count” 将 > 0
    // 这里减 1，当达到 0 时关闭照明
    if (Auto_switch_off_count_G > 0)
    {
        Auto_switch_off_count_G--;
        if (Auto_switch_off_count_G == 0)
        {
            Light_pin_G = LIGHT_OFF;
        }
    }
    // 在每次按下开关之后“阻塞”该开关，使用户有时间移开手指
    // 如果不这样做，当用户按下开关长于 0.4 秒时照明将再次关闭
    //
    // 如果开关被阻塞，阻塞计数递减，不检查开关引脚状态而返回
    if (Switch_blocked_G > 0)
    {
        Switch_blocked_G--;
        return;
    }
    // 现在读取开关引脚
    if (Switch_pin_G == SWITCH_PRESSED)
    {

```

```
// 如果开关引脚被按下，增加开关计数。  
// 如果 Switch_count_G==2，则意味着引脚有效  
// 连续两次调用这个任务，也就是说，是一次真正的开关按下，而不是一次抖动，  
// 变量 Auto_switch_off_count_G 既用作照明点亮的指示（如果它非零）  
// 又用作保持照明期间任务调用数量的计数器  
if (Auto_switch_off_count_G > 0)  
{  
    // 现在照明是点亮的  
    // ->关闭它。  
    Light_pin_G = LIGHT_OFF;  
    Auto_switch_off_count_G = 0;  
}  
else  
{  
    // 照明现在是关闭的  
    // -> 它，并且设置计数器为 150  
    // (任务每隔 0.2s 调用，因此延迟 30 秒)  
    Light_pin_G = LIGHT_ON;  
    Auto_switch_off_count_G = 150;  
}  
// 复位开关计数，并且阻塞开关直到下一秒（5 次调用这个任务），  
Switch_count_G = 0;  
Switch_blocked_G = 5;  
}  
}  
else  
{  
    Switch_count_G = 0;  
}  
}  
/*-----  
--- 文件结束  
-----*/
```

源程序清单 36.12 用于控制自动照明的代码

进阶阅读

Chapter 37

提高调度的稳定性

引言

所有的晶体振荡器和陶瓷谐振器的频率输出都随温度而变化。因此，由这样的时钟源驱动的调度器的定时特性也随温度变化。稳定调度器是一种经过温度补偿的调度器，它将根据环境温度的变化而调整其特性。

稳定调度器

适用场合

- 用 8051 系列微控制器开发一种嵌入式系统。
- 该系统使用调度器构造一种时间触发结构。

问题

如何保证调度系统即使在环境温度有变化时也仍然能正确地定时运行？

背景知识

温度对晶体振荡器电路性能影响的有关讨论参见晶体振荡器。

解决方案

在大多数要求精确定时的实际系统中，系统的振荡时钟来自某种形式的晶体振荡器。不幸的是，这些振荡器的稳定性随温度而变化。为了获得稳定的调度，主要有两种选择：

- 使系统保持在某个不变的、已知的温度，根据这个温度设定所有定时器设置。这可以通过将该系统放在某种形式的工业烘箱或者冰箱中实现。然而，对于大多数系统来说，这种方法是不切实际的。

- 测量环境温度并且根据温度的改变来调整系统定时。

稳定调度器采用第二种方法。

硬件资源

主要的硬件问题是需要某种形式的温度传感器。在大多数情况下，数字传感器（比如 Dallas Semiconductor 的 1620 系列）是一种高性价比的选择，因为它很便宜并且只需要至多三个微控制器引脚。

一种很好的可选方案是采用片内温度传感器。Cygnal^① 和 Analog Devices^② 公司的一些 8051 微控制器就包含有这样的传感器，它们用于这些系统很理想。

以较长的间隔（一般为 1 分钟）进行温度测量时，调度软件占用的开销极小。

可靠性和安全性

如果系统要求长时间精确定时，使用稳定调度器可以提高其可靠性。

可移植性

大多数微控制器能够连接某种形式的外扩温度传感器，因此可以使用这个模式。

优缺点小结

- ⊕ 提高调度器的稳定性。
- ⊗ 通常将增加成本。

相关的模式和替代解决方案

例子：使用外扩的 Dallas 的 DS1621 I²C 温度传感器

在这个例子中，给出了一个简单的稳定调度器，它使用外扩（DS1621）温度传感器（源程序清单 37.1 和源程序清单 37.2）。

演示程序将在 LED 上显示时间，相应的显示硬件参见第 21 章的内容，完整的源程序清单参见随书 CD。

要求的链接程序选项（详细资料参见第 13 章）：

```
/*-----*
Main.c (v1.00)
-----*
用于稳定调度器的演示程序
驱动 4 个多路复用的多段 LED
显示消逝的时间
```

① www.cygnal.com

② www.analog.com

需要的链接选项（详细内容参见第14章）

```

OVERLAY
(main ~ (CLOCK_LED_Time_Update,LED_MX4_Display_Update,
SCH_Calculate_Ave_Temp_DS1621),
SCH_Dispatch_Tasks !(CLOCK_LED_Time_Update,LED_MX4_Display_Update,
SCH_Calculate_Ave_Temp_DS1621))
#include "Main.h"
#include "2_01_12s.h"
#include "LED_Mx4.h"
#include "Cloc_Mx4.h"
#include "I2C_1621.h"
/* ..... */
/* ..... */
void main(void)
{
    // 设置调度器
    SCH_Init_T2();
    // 为温度测量做准备
    I2C_Init_Temperature_DS1621();
    // 添加“Time Update”任务（每秒一次）
    // -定时单位为时标（1ms间隔）
    // (最大的间隔/延迟是65535个时标)
    SCH_Add_Task(CLOCK_LED_Time_Update,100,10);
    // 添加“Display Update”任务（每秒一次）
    // 必须每隔（大约）3ms刷新4段显示
    // 必须每隔（大约）6ms刷新2段显示
    SCH_Add_Task(LED_MX4_Display_Update,0,3);
    // 每分钟调度一次
    SCH_Add_Task(SCH_Calculate_Ave_Temp_DS1621,33,60000);
    // 启动调度器
    SCH_Start();
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
/*-----文件结束-----*/

```

源程序清单37.1 简单的稳定调度器的实现的一部分，使用外扩的（DS1621）温度传感器

```

/*
2_01_12s.C (v1.00)

***这里是用于标准8051/8052的稳定调度器***
***使用T2定时，16位自动重装***
***12MHz的振荡器->1ms（精确）时标间隔***
***假定使用DS1621温度传感器***
#include "2_01_12s.h"

```

```

#include "I2C_1621.h"
// -----公有变量定义-----
// 当前的温度，每小时记录
tByte Temperature_G;
// -----公有变量声明-----
// 任务队列（参见 Sch51.C）
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
// 错误代码变量
//
// See Port.H for port on which error codes are displayed
// 将用来显示错误代码的端口参见 Port.H
// 以及错误代码的详细资料
extern tByte Error_code_G;
// 运行合计/平均温度（每隔 24 小时计算一次）
static int Temperature_average_G = 0;
// 每分钟调用一次，每个小时读取一次
// (每小时调用一次需要修改调度器，并增加了每个任务所需的存储器)
static tByte Minute_G;
static tByte Hour_G;
// 温度补偿数据
//
// 定时器 2 的重装值（低位和高位字节）取决于当前的平均温度而变化
// 注意(1)：
//
// 这个版本只考虑 18~30 摄氏度之间的温度值
// 注意(2)：
// 调整这些值以匹配你的硬件！
tByte code T2_reload_L[21] =
    // 10   11   12   13   14   15   16   17   18   19
    {0xBA,0xB9,0xB8,0xB7,0xB6,0xB5,0xB4,0xB3,0xB2,0xB1,
     // 20   21   22   23   24   25   26   27   28   29   30
     0xB0,0xAF,0xAE,0xAD,0xAC,0xAB,0xAA,0xA9,0xA8,0xA7,0xA6};
tByte code T2_reload_H[21] =
    // 10   11   12   13   14   15   16   17   18   19
    {0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,
     // 20   21   22   23   24   25   26   27   28   29   30
     0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,0x3C,0x3C}.*;
*-----*
SCH_Init_T2()
调度器初始化函数。准备调度器数据结构并且设置定时器以所需的频率中断。
必须在使用调度器之前调用这个函数
*-----*
void SCH_Init_T2(void)
{
    tByte i;
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
}

```

```

// 复位全局错误变量
// -SCH_Delete_Task()将产生一个错误代码
// (因为任务队列是空的)
Error_code_G = 0;
// 现在设置定时器 2
// 自动重装的 16 位定时器函数
// 晶振假定为 12MHz
// 定时器 2 的精度是 0.000001 秒 (1μs)
// 要求的定时器 2 溢出为 0.001 秒 (1ms)
// -需要 1000 个定时器时标
// 重装值为 65536 - 1000 = 64536 (十进制) = 0xFC18
T2CON = 0x04;      // 加载定时器 2 的控制寄存器
T2MOD = 0x00;      // 加载定时器 2 的模式寄存器
TH2    = 0xFC;      // 加载定时器 2 的高位字节
RCAP2H = 0xFC;      // 加载定时器 2 的重装捕捉寄存器的高位字节
TL2    = 0x18;      // 加载定时器 2 的低位字节
RCAP2L = 0x18;      // 加载定时器 2 的重装捕捉寄存器的低位字节
ET2    = 1;         // 使能定时器 2 中断
TR2    = 1;         // 启动定时器 2
}

/*-----*
SCH_Start()
通过允许中断来启动调度器
注意：通常在添加了所有定期的任务之后调用，从而使任务保持同步。
注意：应该只使能调度器中断!!!
-----*/
void SCH_Start(void)
{
EA = 1;
}
/*-----*
SCH_Update()
这是调度器的中断服务程序。
初始化函数中的定时器设置决定了它的调用频率。
这个版本由定时器 2 中断触发；
定时器自动重装
-----*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
tByte Index;
TF2 = 0; // 必须手工清除
// 注意：计算单位为“时标”（不是毫秒）
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
// 检测这里是否有任务
if (SCH_tasks_G[Index].pTask)
{
if (SCH_tasks_G[Index].Delay == 0)

```

```
{  
    // 任务需要运行  
    SCH_tasks_G[Index].RunMe += 1; // “RunMe” 标志加 1  
    if (SCH_tasks_G[Index].Period)  
    {  
        // 调度周期性的任务再次运行  
        SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;  
    }  
    }  
    else  
    {  
        // 还没有准备好运行：延迟减 1  
        SCH_tasks_G[Index].Delay -= 1;  
    }  
}  
}  
/*-----*  
SCH_Calculate_Ave_Temp_DS1621()  
这个函数应该每分钟调度一次。  
这个函数以每小时一次的测量值为基础，用前 24 小时的平均温度刷新变量。  
该刷新每隔 24 小时执行一次  
-----*/  
void SCH_Calculate_Ave_Temp_DS1621(void)  
{  
    if (++Minute_G == 60)  
    {  
        Minute_G = 0;  
        // 已经过去一个小时——读取温度读数  
        I2C_Read_Temperature_DS1621();  
        // 把当前的读数加到运行合计上  
        Temperature_average_G += Temperature_G;  
        if (++Hour_G == 24)  
        {  
            // 已经过去 24 小时——得到平均温度  
            Hour_G = 0;  
            Temperature_average_G /= 24;  
            // 刷新调度器  
            SCH_Perform_Temperature_Adjustment();  
        }  
    }  
}  
/*-----*  
SCH_Perform_Temperature_Adjustment()  
调度器根据环境温度的变化调整定时  
-----*/  
void SCH_Perform_Temperature_Adjustment(void)  
{  
    static int Previous_temperature_average_G;
```

```

if ((Previous_temperature_average_G - Temperature_average_G) != 0)
{
    // 这个版本只考虑 10~30 摄氏度之间的温度（容易校准）
    if (Temperature_average_G < 10)
    {
        Temperature_average_G = 10;
    }
    else
    {
        if (Temperature_average_G > 30)
        {
            Temperature_average_G = 30;
        }
    }
}
ET2 = 0; // 禁止中断
TR2 = 0; // 停止定时器 2

// 重装定时器
TL2 = T2_reload_L[Temperature_average_G-10];
RCAP2L = T2_reload_L[Temperature_average_G-10];
TH2 = T2_reload_H[Temperature_average_G-10];
RCAP2H = T2_reload_H[Temperature_average_G-10];

ET2 = 1;
TR2 = 1;
}
Previous_temperature_average_G = Temperature_average_G;
Temperature_average_G = 0;
}

/*
-----文件结束-----
*/

```

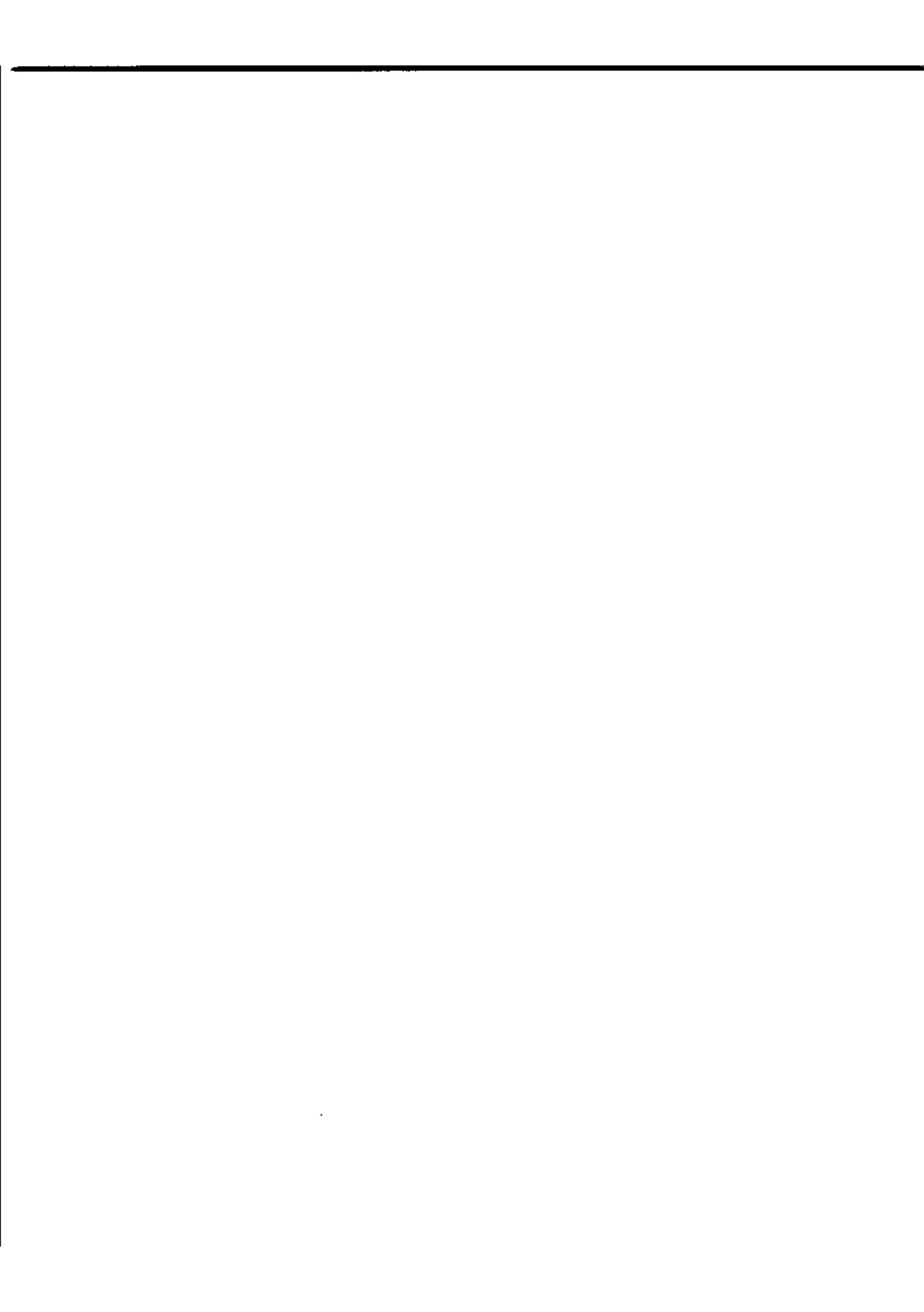
源程序清单 37.2 简单的稳定调度器的实现的一部分，使用外扩的（DS1621）温度传感器

进阶阅读

结 论

将以最后这两章结束本书。

- 在第 38 章中，将回顾通过介绍这些模式我们试图实现了什么样的目标。
- 在第 39 章中，将给出正文中直接引用的，以及相关书籍和论文的列表。



Chapter 38

本书试图实现的目标

38.1 引言

在本章中，将回顾通过介绍这些模式我们试图实现什么样的目标。

38.2 本书试图实现的目标

很多现有的嵌入式系统都采用以下两种方式之一来开发：

- 不使用特定的系统总体结构，而常常使用多个中断服务程序。
- 使用商业的、抢占式的操作系统。

这两种开发方式导致大多数研究机构和其他研发组织在许多方面都会遇到以下问题：

- 大多数用于嵌入式系统的公开代码（例如在互联网上）以及厂家针对微控制器和相关模块的应用手册，其代码的书写形式往往是非正规的，并且非常强调中断的使用。
- 这一领域的学术研究往往关注对复杂的抢占式应用的技术上和理论上的分析（详细资料参见第 13 章）。这些研究所给出的例子常常和实际项目所面对的问题脱节。

贯穿于本书，将设法在这两个领域之间寻找一条中间路径。具体而言，将试图展示：对于一系列实际的嵌入式系统，即使是基于 CPU 和存储器资源非常有限的多个 8 位微控制器的分布式应用，使用（每个微控制器）有单个中断源的合作式调度器也是一种实用的方法。

我们还试图非正式地展示：基于合作式结构的系统具有很好的可预测特性，因此适用于作为与安全相关的系统的基础。我们论证了可预测性和可靠性不仅仅是与安全相关的系统的基本要求，只要嵌入式系统的设计开发人员关注设计出的产品的质量，这些特性就不可忽略。

这些调度器简单易用的特性还带来了其他的好处。例如，这意味着开发人员自己能够非常快地将调度器移植到新的微控制器平台下。同时也意味着基本结构可以很容易地根据特殊应用的需要而修改，而不需改变基本（合作式的）的方法（正如在第 8 篇中试图展示的）。

然而，使用这些简单调度器的最重要的副作用也许是（与传统的实时操作系统不同）它们成为系统（如图 38.1 所示）的一部分，而不是成为用户代码和系统应用之间的分离的代码层（如图 38.2 所示）。

根据我们的经验，调度器和系统的紧密结合意味着开发人员能够迅速理解和掌握调度器代码。这是很重要的，因为它避免了“不是在这里创制的”或者“把它归咎于操作系统”的哲学。在开发人员必须将他们的代码和一个庞大而复杂的实时操作系统接口时，可能会出现这种情况。因为这样的实时操作系统的特性可能永远不会为开发人员完全理解。



图 38.1 (本书中讨论的) 合作式调度器成为应用系统的一部分

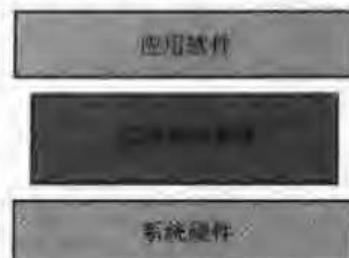


图 38.2 “实时操作系统”仍然是一种分离的应用系统

38.3 小结

在本章中，简要地回顾了我们通过介绍的模式试图实现什么样的目标。

本书到此结束，如需要更进一步探讨本章以及前面章节中讨论的主题，请参考第 39 章中列出的资料。

收集的参考文献和书目

在本章中，将给出正文中直接引用的以及相关的书籍和论文列表。

39.1 出版书刊一览表

- Acarnley, P.P. (1984) *Stepping Motors: A Guide to Modern Theory and Practice*, Peter Peregrinus Ltd, UK.
- Ackermann, J. (1998) 'Active steering for better safety, handling and comfort', Proceedings of Advances in Vehicle Control and Safety, Amiens, France, 1-3 July.
- Alexander, C. (1979) *The Timeless Way of Building*, Oxford University Press, New York.
- Alexander, C., Ishikawa, S., Silverstein, M. with Jacobson, M., Fisksdahl-King, I. and Angel, S. (1977) *A Pattern Language*, Oxford University Press, New York.
- Allworth, S.T. (1981) *An Introduction to Real-Time Software Design*, Macmillan, London.
- Atherton, D.P. (1999) 'PID controller tuning', *IEE Computing & Control Engineering Journal*, 10 (2): 44-50.
- Awad, M., Kuusela, J. and Ziegler, J. (1996) *Object-oriented Technology for Real-time Systems*, Prentice-Hall, New Jersey.
- Axelson, J. (1998) *Serial Port Complete*, Lakeview Research.
- Axelson, J. (1999) *USB Complete*, Lakeview Research.
- Ayala, K. (2000) *The 80251 Microcontroller*, Prentice Hall, New Jersey.
- Barnett, R.H. (1995) *The 8051 Family of Microcontrollers*, Prentice Hall, New Jersey.
- Barrenscheen, J. (1996) 'On-board communication via CAN without Transceiver', Infineon (Siemens) Application Note AP2921. [Available from www.infineon.com]
- Bates, I. (2000) 'Introduction to scheduling and timing analysis', in *The Use of Ada in Real-Time System*, IEE Conference Publication 00/034.
- Bennett, S. (1994) *Real-Time Computer Control*, 2nd edn, Prentice Hall, New Jersey.

- Bignell, V. and Fortune, J. (1984) *Understanding System Failures*, Manchester University Press, Manchester.
- Bishop, C.M (1995) *Neural Networks for Pattern Recognition*, Oxford University Press Oxford.
- Boehm, B.W. (1981) *Software Engineering Economics*, Prentice Hall, New Jersey.
- Booch, G (1994) *Object-Oriented Analysis and Design*, Benjamin Cummings.
- Bowerman, B.L. and O'Connell, R.T. (1987) *Time Series Forecasting: Unified Concepts and Computer Implementation*, Duxbury Press, Boston, MA.
- Brooks, E.P. (1975) *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, MA.
- Brooks, E.P. (1986) 'No silver bullet- essence and accidents of software engineering', in H.J. Kugler (ed.) *Information Processing 86*, Elsevier Science, Amsterdam.
- Broughton, J. (1994) 'Assessing the safety of new vehicle control systems', Proceedings of the First World Congress on Applications of Transport Telematics and Intelligent Vehicle-Highway Systems, Paris, November 1994.
- BS IEC 61508 (1999) 'Functional safety of electrical / electronic / programmable electronic safety-related systems'. [Available from BSI, London]
- Burns, A. and Wellings, A. (1997) *Real-time Systems and Programming Languages*, Addison-Wesley, Reading, MA.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996) *Pattern- Oriented Software Architecture: A System of Patterns*, Wiley, Chichester.
- Cahill, S.J. (1994) *C for the Microprocessor Engineer*, Prentice Hall, New Jersey.
- Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F and Jeremes, P.(1994) *Object-oriented Development: The Fusion Method*, Prentice Hall, New Jersey.
- Cooling, J.E. (1991) *Software Design for Real-time Systems*, Chapman & Hall, London.
- Cunningham, W. and Beck, K. (1987) 'Using pattern languages for object-oriented programs', Proceedings of OOPSLA'87, Orlando, FL.
- Daley, S. and Liu, G.P. (1999) 'Optimal PID tuning using direct search algorithms', IEE Computing & Control Engineering Journal, 10 (2): 51-56.
- DeMarco, T. (1978) *Structured Analysis and System Specification*, Prentice-Hall, New Jersey.
- Dorf, R.C. and Bishop, R.H. (1998) *Modem Control Systems*, 8th edn, Addison-Wesley, Reading, MA.
- Douglass, B.P. (1998) *Real-time UML*, Addison-Wesley, Reading, MA.
- Doyle, F.J., Gatzke, E.P. and Parker, R.S. (1999) *Process Control Modules: A Software Laboratory for Control Design: The MATLAB-based Process Control Guide for Chemical Engineering Professionals*, Prentice Hall, New Jersey.
- Dutton, K., Thompson, S. and Barraclough, B. (1997) *The Art of Control Engineering*.

- Addison-Wesley, Reading, MA.
- Ebinger, B., Sienel, W. and Sporl, T. (1998) 'A hardware-in-the-loop test environment for interconnected ECUs for passenger cars and commercial vehicles', Proceedings of Advances in Vehicle Control and Safety, Amiens, France, 1-3 July.
- Elgar, P. (1998) Sensors for Measurement and Control, Longman, London.
- Falla, M. (1997) Advances in Safety-Critical Systems, University of Lancaster Press, Lancaster.
- Fenton, N. and Pfleeger, S.L. (1996) Software Metrics, Thomson, London.
- Fowler, M. and Scott, K. (2000) UML Distilled, 2nd edn, Addison-Wesley, Reading, MA.
- Franco, S. (1998) Design with Operational Amplifiers and Analog Integrated Circuits, 2nd edn, McGraw-Hill, Boston, MA.
- Franklin, G.E, Powell, J.D. and Emami-Naeini, A. (1994) Feedback Control of Dynamic Systems, 3rd edn, Addison-Wesley, Reading, MA.
- Franklin, G.F, Powell, J.D. and Workman, M. (1998) Digital Control of Dynamic Systems, 3rd edn, Addison-Wesley, Reading, MA.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley, Reading, MA.
- Ganssle, J. (1992) The Art of Programming Embedded Systems, Academic Press, San Diego.
- Gergeleit, M. and Streich, H. (1994) 'Implementing a distributed high-resolution real-time clock using the CAN-bus', Proceedings 1st International CAN Conference, Mainz, Germany, September 1994.
- Goldie, J. (1991) 'Comparing EIA-485 and EIA-422-A line drivers and receivers in multi-point applications', National Semiconductor Application Note 759. [Available from www.national.com]
- Goldie, J. (1996) 'Ten ways to bulletproof RS-485 interfaces', National Semiconductor Application Note 1057. [Available from www.national.com]
- Goldsmith, S. (1993) A Practical Guide to Real-Time Systems Development, Prentice Hall, New Jersey.
- Graham, I. (1994) Object-Oriented Methods, 2nd edn, Addison-Wesley, Reading, MA.
- Haney, P.R., Richardson, M.J., Clarke, N.J. and Barber, P.A. (1998) 'Development of adaptive cruise control systems for motor vehicles', Proceedings of Control '98, Swansea (Mini Symposium on Mechatronics).
- Hank, P. and Jbhnk, E. (1997) 'SJA1000 stand-alone CAN controller', Philips Application Note AN97076. [Available from www.philips.com]
- Hatley, D.J. and Pirbhai, I.A. (1987) Strategies for Real-time System Specification, Dorset House.
- Hatton, L. (1994) Safer C: Developing Software for High-integrity and Safety-critical Systems', McGraw-Hill, Maidenhead.

- Haykin, S. (1994) Neural networks: A Comprehensive Foundation, Macmillan College Publishing Company, New York.
- Horowitz, P. and Hill, W. (1989) The Art of Electronics, 2nd edn, Cambridge University Press, Cambridge.
- Huang, H-W (2000) Using the MCS-51 Microcontroller, Oxford University Press, New York.
- Intel (1985) Microcontroller Handbook 1986, Intel Corporation.
- Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1993) Object-Oriented Software Engineering, revised edn, Addison-Wesley, Reading, MA.
- Kenjo, T. (1984) Stepping Motors and their Microprocessor Circuits, Clarendon Press, Oxford.
- Kopetz, H. (1997) Real-time Systems: Design Principles for Distributed Embedded Applications, Kluwer Academic, New York.
- Labrosse, J.J. (1998) uC/OS II: The Real-Time Kernel, R&D Books.
- Lander, C.W. (1993) Power Electronics, 3rd edn, McGraw-Hill, Maidenhead.
- Lawrence, P.D. and Mauch, K. (1988) Real-Time Microcomputer System Design: An Introduction, McGraw-Hill, Maidenhead.
- Lawrenz, W. (1997) CAN System Engineering, Springer-Verlag, Heidelberg.
- Leen, G., Heffernan, D. and Dunne, A. (1999) Digital networks in the automotive vehicle', Computing and Control, 10 (6): 257-66.
- Leveson, N.G. (1995) Safeware: System Safety and Computers, Addison-Wesley, Reading, MA.
- Li, Y., Pont, M.J. and Jones, N.B. (1999) 'A comparison of the performance of radial basis function and multi-layer Perceptron networks in a practical condition monitoring application', Proceedings of Condition Monitoring, Swansea, UK, April 12-15, 1999.
- Li, Y., Pont, M.J., Parikh, C.R. and Jones, N.B. (2000) 'Using a combination of RBFN, MLP and kNN classifiers for engine misfire detection', in R. John and R. Birkenhead (eds) Advances in Soft Computing: Soft Computing Techniques and Applications, Springer-Verlag, Heidelberg.
- Lippmann, P. (1987) 'An introduction to computing with neural networks', Institute of Electrical and Electronic Engineers (USA), Acoustics, Speech and Signal Processing, April, 1987.
- Liu, J.W.S. and Ha, R. (1995) 'Methods for validating real-time constraints', Journal of Systems and Software, 30 (1-2'): 85-98.
- Locke, C.D. (1992) 'Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives', Journal of Real-Time Systems, 4: 37-53.
- Lynn, P. and Fuerst, W. (1998) Introductory Digital Signal Processing with Computer Applications, Wiley, Chichester.
- Mariutti, P. (1999) 'Crystal oscillator of the C500 and C166 microcontroller families', Infineon (Siemens) Application Note AP242005. [Available from www.infineon.com]
- MISRA (1994) 'Development guidelines for vehicle-based software', Motor Industry Software

Reliability Report, November, 1994. [These guidelines, plus nine supporting reports, are available from MIRA]

MISRA (1998) Guidelines for the use of the C language in vehicle-based software', Motor Industry Software Reliability Report. [Available from MIRA]

Nelson, T. (1995) 'The practical limits of RS-485', National Semiconductor Application Note 979. [Available from www.national.com]

NHTSA (1996) 'Effectiveness of occupant safety systems and their use'. National Highway Traffic Safety Administration (US) Third Report to Congress, December 1996.

NHTSA (1999) 'NHTSA light vehicle antilock brake system research program task 4', National Highway Traffic Safety Administration (US) Report, January 1999.

Nise, N.S. (1995) Control Systems Engineering, 2nd edn, Addison-Wesley, Reading, MA.

Nissanke, N. (1997) Realtime Systems, Prentice Hall, New Jersey.

Ong, H.L.R., Pont, M.J. and Peasgood, W. (2001) 'Do software-based techniques increase the reliability of embedded applications in the presence of EMI?', Microprocessors and Microsystems, 24 (10), 481-91.

Oppenheim, A.V., Schafer, R.W. and Buck, J.R. (1999) Discrete-time Signal Processing, Prentice Hall, New Jersey.

Palacheria, A. (1997) 'Using PWM to generate analog output', Microchip Application Note AN538.

Parikh, C.R., Pont, M.J., Li, Y. and Jones, N.B. (1999) 'Improving the performance of multi-layer Perceptrons where limited training data are available for some classes', Proceedings of the IEE International Conference on Neural Networks, Edinburgh, September 1999.

Parikh C.R., M.J. Pont and N.B. Jones (2001) 'Application of Dempster-Shafer theory in condition monitoring applications - a case study', Pattern Recognition Letters, 22 (6-7), 777-85.

Passino, K.M. and Yurkovich, S. (1998) Fuzzy Control, Addison-Wesley, Reading, MA.

Perier, L. and Coen, A. (1998) 'CAN-do solutions for car multiplexing', Proceedings of the 5th International CAN Conference, San Jose, California, November 1998.

Plauger, P.J. (1992) The Standard C Library, Prentice Hall, New Jersey.

Pont, M.J. (1996) Software Engineering with C++ and CASE Tools, Addison-Wesley, Reading, MA.

Pont, M.J. (1998) 'Control system design using real-time design patterns', Proceedings of Control '98, Swansea, UK, September.

Pont, M.J. (in press) 'Designing and implementing reliable embedded systems using patterns', in P. Dyson (ed.) Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999, to be published by Springer-Verlag.

Pont, M.J., Ong, H.L.R., Parikh, C.R., Kureemun, R., Wong, C.P., Peasgood, W. and Li, Y.

- (1999a) 'A selection of patterns for reliable embedded systems', original paper presented at EuroPlop '99, Kloster Irsee, Germany.
- Pont, M.J., Li, Y., Parikh, C.R. and Wong, C.P. (1999b) 'The design of embedded systems using software patterns', Proceedings of Condition Monitoring 1999, Swansea, UK, 12-15 April.
- Press, W.H., Teulolsky, S.A., Vettering, W.T. and Flannery, B. P. (1992) Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, Cambridge.
- Pressman, R. (1992) Software Engineering: A Practitioner's Approach, 3rd edn. McGraw-Hill, Maidenhead.
- Ralston, A. and Meek, C.L. (1976) Encyclopaedia of Computer Science, Petrocelli/Charter.
- Rashid, M.H. (1993) Power Electronics: Circuits, Devices and Applications, 2nd edn, Prentice Hall, New Jersey.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) Object- Oriented Modeling and Design, Prentice Hall, New Jersey.
- Schildt, H. (1997) Teach Yourself C, 3rd edn, McGraw-Hill, Maidenhead.
- Scott, G (1995) 'Interfacing an MCS 51 Microcontroller to an 82527 CAN Controller', Intel Application Note AP-724. [Available from www.intel.com]
- Selic, B., Gullekson, G and Ward, P.T. (1994) Real-time Object-oriented Modeling, Wiley, New York.
- Sharp, R.S. (1998) 'Variable geometry active suspension for cars', IEE Computing and Control Engineering Journal, 9 (5): 217-22.
- Shaw, A.C. (2001) Real-Time Systems and Software, Wiley, New York.
- Sivasothy, S. (1998) 'Transceivers and repeaters meeting the EIA RS-485 interface standard', National Semiconductor Application Note 409. [Available from www.national.com]
- Smith, S.W. (1999) The Scientist and Engineer's Guide to Digital Signal Processing, 2nd edn, California Technical Publishing. [Available from www.DSPguide.com]
- Somerville, I. (1996) Software Engineering, 5th edn, Addison-Wesley, Reading, MA.
- Storey, N. (1996) Safety-critical Computer Systems, Addison-Wesley, Reading, MA.
- Tindell, K. (1998) 'Embedded systems in the automotive industry', Proceedings of the 1998 Embedded Systems Conference, San Jos(~, CA.
- Waites, N. and Knott, G (1996) Computing, 2nd edn, Business Education Publishers, Sunderland.
- Ward, N.J. (1991) 'The static analysis of a safety-critical avionics control system', in D.E. Corbyn and N.P. Bray (eds) Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, 1991, SaRS.
- Ward, P.T. and Melior, S.J. (1985) Structured Development for Real-Time Systems, Prentice Hall, New Jersey.

- Warnes, L. (1998) Electronic and Electrical Engineering: Principles and Practice, Macmillan, London.
- Wong, C. P. and Pont, M. J. (2000) 'An overview of an evolutionary algorithm pattern language', in R. John and R. Birkenhead (eds) Advances in Soft Computing: Soft Computing Techniques and Applications, Springer-Verlag, Heidelberg.
- Yalamanchili, S. (2001) Introductory VHDL: From Simulation to Synthesis, Prentice Hall, New Jersey.
- Yourdon, E.N. (1989) Modern Structured Analysis, Prentice Hall, New Jersey.
- Ziegler, J.G and Nichols, N.B. (1942) 'Optimal setting for automatic controllers', Trans. ASME, 64(11), 759-68.
- Ziegler, J.G and Nichols, N.B. (1943) 'Process lags in automatic control circuits', Trans. ASME, 65(5), 433-44.

39.2 其他模式

- Alexander, C. (1979) The Timeless Way of Building, Oxford University Press, New York.
- Alexander, C., Ishikawa, S., Silverstein, M. with Jacobson, M. Fisksdahl-King, I., and Angel, S. (1977) A Pattern Language, Oxford University Press, New York.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996) Pattern-oriented Software Architecture: A System of Patterns, Wiley, Chichester.
- Cunningham, W. and Beck, K. (1987) 'Using pattern languages for object-oriented programs', Proceedings of OOPSLA'87, Orlando, FL.
- Douglass, B.P. (1998) Real-time UML, Addison-Wesley, Reading, MA.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley, Reading, MA.
- Wong, C. P. and Pont, M. J. (2000) 'An overview of an evolutionary algorithm pattern language', in R. John, and R. Birkenhead (eds) Advances in Soft Computing: Soft Computing Techniques and Applications, Springer-Verlag, Heidelberg.
- Ziegler, J.G and Nichols, N.B. (1942) 'Optimal setting for automatic controllers', Trans. ASME, 64 (11), 759-68.
- Ziegler, J.G and Nichols, N.B. (1943) 'Process lags in automatic control circuits', Trans. ASME, 65 (5), 433-44.

39.3 实时嵌入式系统设计技术

- Allworth, S.T. (1981) An Introduction to Real-Time Software Design, Macmillan, London.
- Awad, M., Kuusela, J. and Ziegler, J. (1996) Object-oriented Technology for Real-time Systems,

Prentice Hall, New Jersey.

Cooling, J.E. (1991) *Software Design for Real-time Systems*, Chapman & Hall, London.

Douglass, B.P. (1998) *Real-time UML*, Addison-Wesley, Reading, MA.

Goldsmith, S. (1993) *A Practical Guide to Real-Time Systems Development*, Prentice Hall, New Jersey.

Hatley, D.J. and Pirbhai, I.A. (1987) *Strategies for Real-time System Specification*, Dorset House.

Kopetz, H. (1997) *Real-time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic, New York.

Lawrence, P.D. and Mauch, K. (1988) *Real-Time Microcomputer System Design: An Introduction*, McGraw-Hill, Maidenhead.

Nissanke, N. (1997) *Realtime Systems*, Prentice Hall, New Jersey.

Selic, B., Gullekson, G and Ward, P.T. (1994) *Real-time Object-oriented Modeling*, Wiley, New York.

Shaw, A.C. (2001) *Real-Time Systems and Software*, Wiley, New York.

Ward, P.T. and Melior, S.J. (1985) *Structured Development for Real-Time Systems*, Prentice Hall, New Jersey.

Ward, N.J. (1991) 'The static analysis of a safety-critical avionics control system', in D.E. Corbyn and N.P. Bray (eds) *Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference*, 1991, SaRS.

39.4 高可靠性系统设计技术

Bignell, V. and Fortune, J. (1984) *Understanding System Failures*, Manchester University Press, Manchester.

Broughton, J. (1994) 'Assessing the safety of new vehicle control systems', *Proceedings of the First World Congress on Applications of Transport Telematics and Intelligent Vehicle-Highway Systems*, Paris, November.

BS IEC 61508 (1999) 'Functional safety of electrical/electronic/programmable electronic safety-related systems'. [Available from BSI, London]

Falla, M. (1997) *Advances in Safety-Critical Systems*, University of Lancaster Press, Lancaster.

Hatton, L. (1994) *Safer C: Developing Software for High-integrity and Safety-critical Systems*, McGraw-Hill, Maidenhead.

Leveson, N.G (1995) *Safeware: System Safety and Computers*, Addison-Wesley, Reading, MA.

MISRA (1994) 'Development guidelines for vehicle-based software', *Motor Industry Software Reliability Report*, November, 1994. [These guidelines, plus nine supporting reports, are available from MIRA.]

- MISRA (1998) 'Guidelines for the use of the C language in vehicle-based software', Motor Industry Software Reliability Report. [Available from MIRA.]
- Ong, H.L.R., Pont, M.J. and Peasgood, W. (2001) 'Do software-based techniques increase the reliability of embedded applications in the presence of EMI?', *Microprocessors and Microsystems*, 24 (10), 481-91.
- Storey, N. (1996) *Safety-critical Computer Systems*, Addison-Wesley, Reading, MA.

39.5 8051微控制器

Barnett, R.H. (1995) *The 8051 Family of Microcontrollers*, Prentice Hall, New Jersey.

Huang, H-W (2000) *Using the MCS-51 Microcontroller*, Oxford University Press, New York.

39.6 作者的相关著作

Li, Y., Pont, M.J., and Jones, N.B. (1999) 'A comparison of the performance of radial basis function and multi-layer Perceptron networks in a practical condition monitoring application', *Proceedings of Condition Monitoring 1999*, Swansea, UK, April 12-15.

Li, Y., Pont, M.J., Parikh, C.R. and Jones, N.B. (2000) 'Using a combination of RBFN, MLP and kNN classifiers for engine misfire detection', in R. John, and R. Birkenhead (eds) *Advances in Soft Computing: Soft Computing Techniques and Applications*, Springer-Verlag, Heidelberg.

Ong, H.L.R., Pont, M.J. and Peasgood, W. (2001) 'Do software-based techniques increase the reliability of embedded applications in the presence of EMI?', *Microprocessors and Microsystems*, 24 (10), 481-91.

Parikh, C.R., Pont, M.J., Li, Y. and Jones, N.B. (1999) 'Improving the performance of multi-layer Perceptrons where limited training data are available for some classes', *Proceedings of the IEE International Conference on Neural Networks*, Edinburgh, September.

Parikh C.R., M.J. Pont and N.B. Jones (2001) 'Application of Dempster-Shafer theory in condition monitoring systems', *Pattern Recognition Letters*, 22 (6-7) 777-85.

Pont, M.J. (1996) *Software Engineering with C++ and CASE Tools*, Addison-Wesley, Reading, MA.

Pont, M.J. (1998) 'Control system design using real-time design patterns', *Proceedings of Control '98*, Swansea, UK, September 1998.

Pont, M.J., Ong, H.L.R., Parikh, C.R., Kureemun, R., Wong, C.P., Peasgood, W. and Li, Y. (1999a) 'A selection of patterns for reliable embedded systems', original paper presented at EuroPlop '99, Kloster Irsee, Germany.

Pont, M.J., Li, Y., Parikh, C.R. and Wong, C.P. (1999b) 'The design of embedded systems using software patterns', *Proceedings of Condition Monitoring 1999*, Swansea, UK, 12-15 April.

Pont, M.J. (in press) 'Designing and implementing reliable embedded systems using patterns', in Dyson, P. (Ed.) Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999, to be published by Springer- Verlag.

Wong, C. P. and Pont, M. J. (2000) 'An overview of an evolutionary algorithm pattern language', in: R. John, and R. Birkenhead, (eds) Advances in Soft Computing: Soft Computing Techniques and Applications, Springer-Verlag, Heidelberg.

附录

共有三个附录：

附录 A 提供了贯穿于本书所使用的设计表示法。

附录 B 提供了有关 CD 内容的介绍。

附录 C 提供了与本书有关的互联网站点的介绍。



A

设计表示法以及 CASE 工具

概述

本附录提供了对贯穿于本书所使用的设计表示法的简要描述，以及用来生成相关图表的 CASE 工具。

CASE 工具

贯穿于本书的图表使用“Yourdon”CASE 工具创建，这个产品现在由 Aonix 公司提供。^①

注意，这里的 CASE 工具的拷贝包含在“C++ 软件工程与 CASE 工具”一书中(Font, 1996)。这是该产品的完整拷贝，然而其许可证是教育性质的，不能用于商业目的。

表示法

本书使用的表示法在其他地方 (Font, 1996) 有详细说明。简而言之，设计由一系列层次来描述，从初始的高级设计开始 (以顶层数据流图的形式)，并以某种形式的过程说明结束 (一般将由 C 函数来实现)。

设计过程也包括创建用户界面和模块测试等方面，这里没有讨论这些问题。

以下一系列图表 (图 A1.1~图 A1.6) 取自 Font (1996)，用来说明用于银行自动出纳机设计的一些主要文档 (图 A1.1)，以及代码框架 (在这个例子中为桌面 C++，源程序清单 A1.1)。

^① www.aonix.com

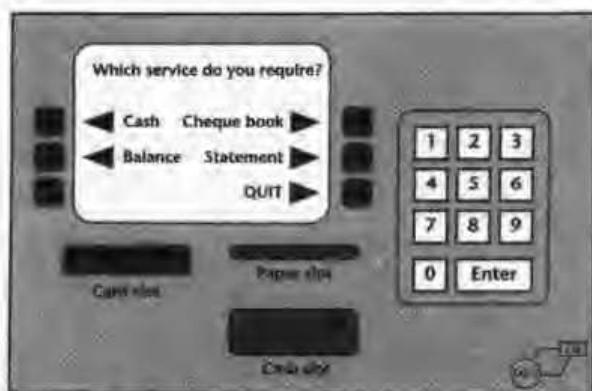


图 A1.1 自动出纳机接口

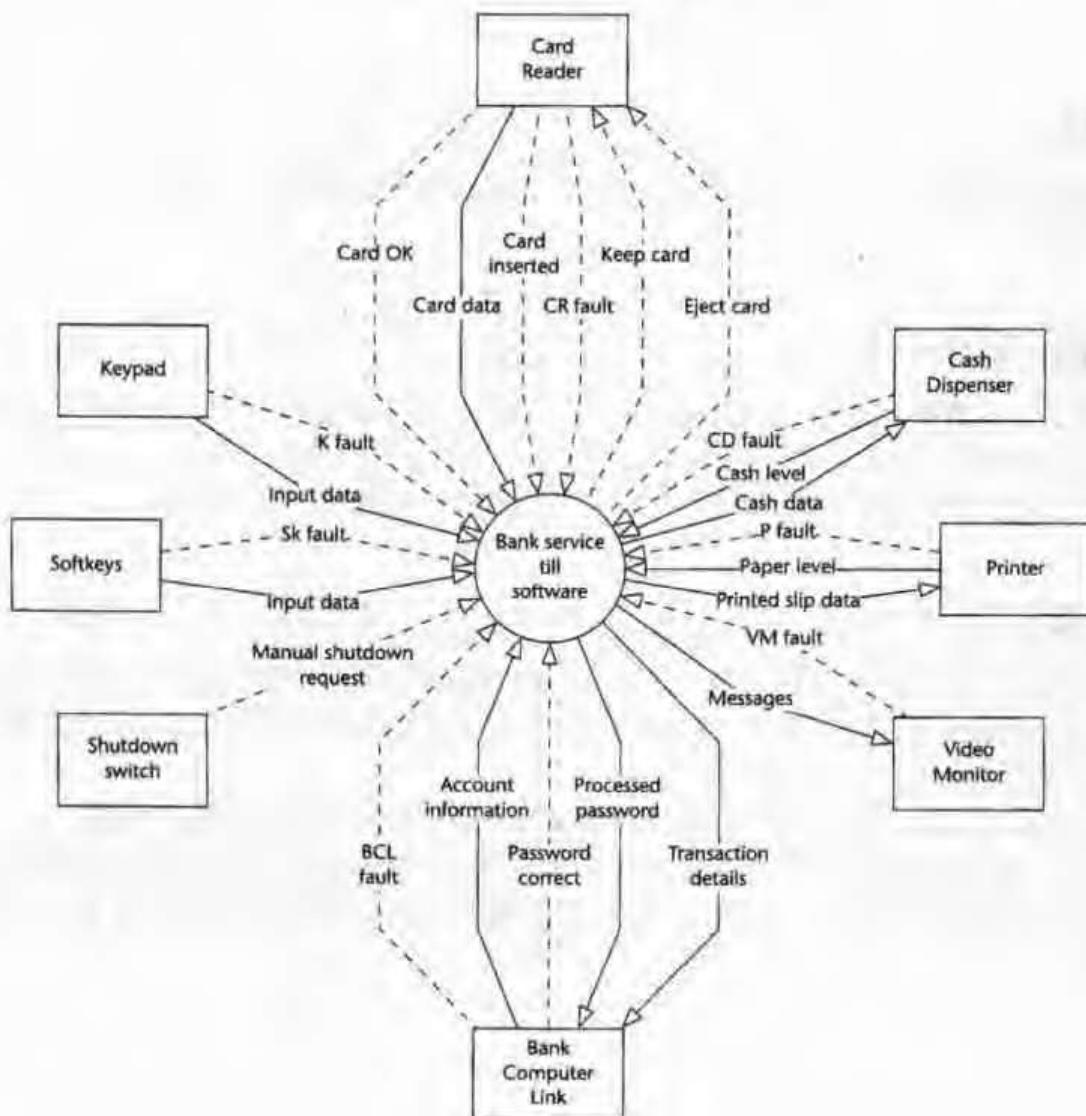


图 A1.2 顶层数据流图

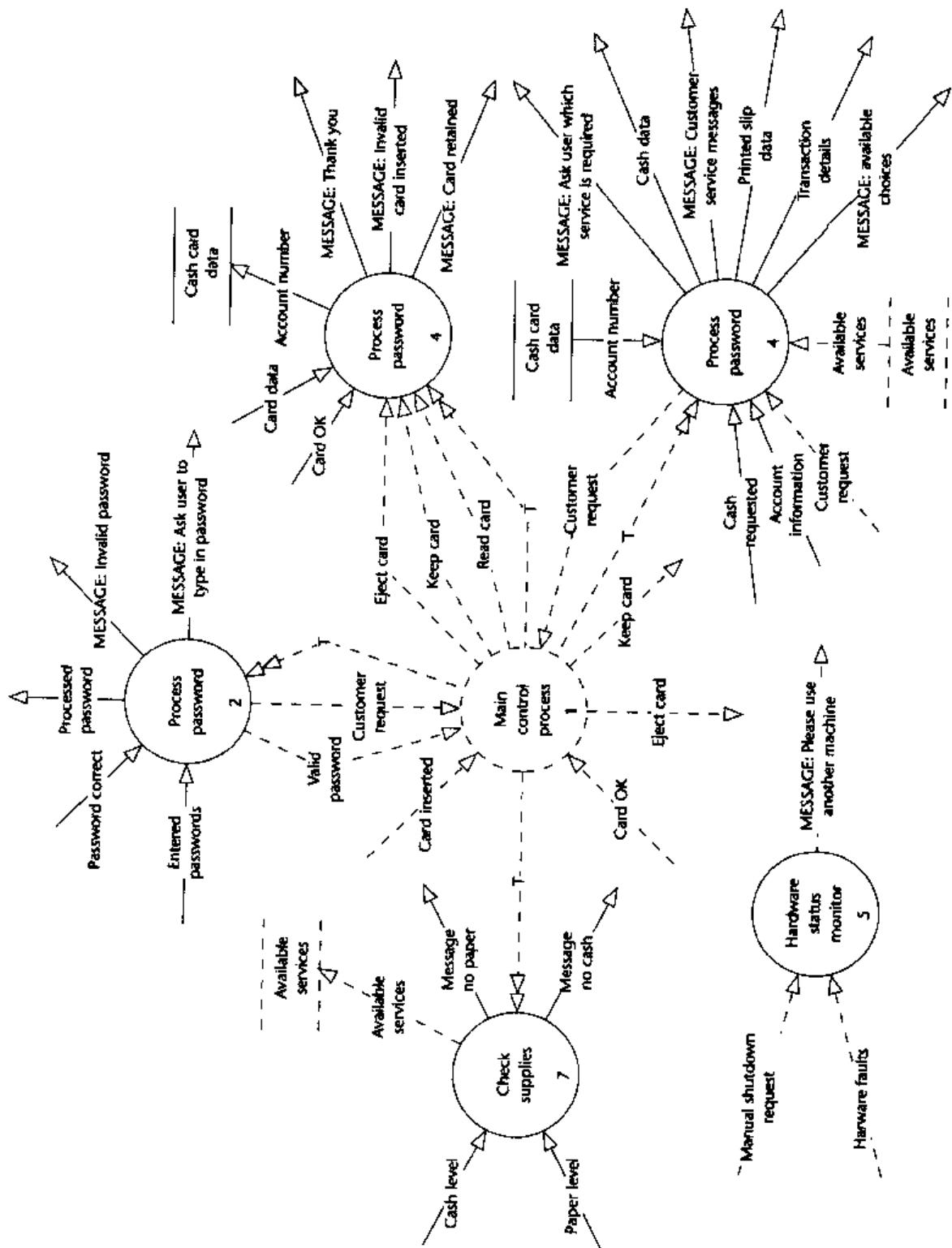


图 A1.3 第一层 DID

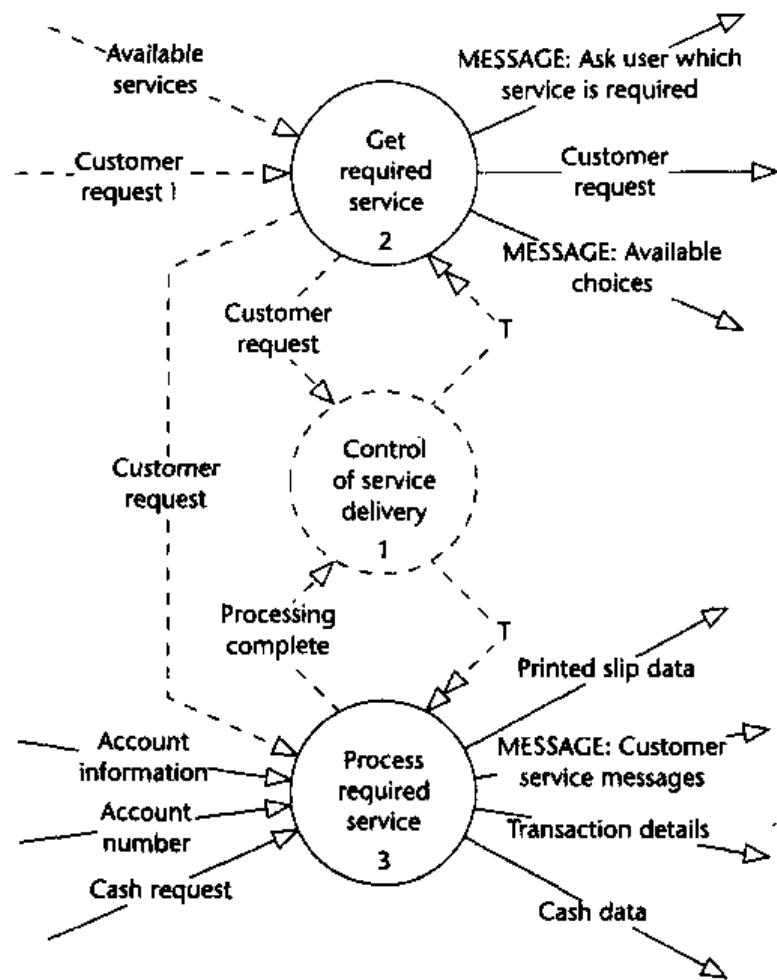


图 A1.4 DfD 1-3



图 A1.5 DFD 1-3-3

@IN=可选的服务
 @IN=顾客请求 I
 @OUT=顾客请求
 @OUT=信息，询问用户需要哪种服务
 @OUT=信息，可选的选择
 @PSpec 得到所需的服务
 //产生简单的菜单（只包含可选的选择）并返回一个有效的选择
前置条件: None
 (信息：询问用户需要哪种服务) = 需要哪种服务?
 (信息：可选的选择) = (可选的服务) 列表
 (顾客请求) = (顾客请求 I)
后续条件:
 (顾客请求) 是一个有效的选择
 @

图 A1.6 PSpec 1-3-2

```

*****  

*  

* (面向过程的)银行服务(自动出纳机)模拟  

*  

*  

*****  

#include <assert.h>  

#include <iostream.h>  

#include <stdlib.h>  

#include <time.h>  

// 尝试输入正确口令的最大允许次数  

const int MAX_NUM_PASSTRIES = 3;  

// 各种标志  

const int TRUE = 1;  

const int FLASH = 0;  

const int OUT_OF_PAPER = 10;  

const int OUT_OF_CASH = 11;  

const int CHECK_CARD = 12;  

const int KEEP_CARD = 13;  

const int EJECT_CARD = 14;  

const int ORDER_CHEQUE_BOOK = 15;  

const int PRINT_BALANCE = 16;  

const int PRINT_MINI_STATEMENT = 17;  

const int DISPENSE_CASH = 18;  

const int QUIT = 19;  

// 函数原型

```

```
void Check_cash_reserves(int&);  
void Check_paper_reserves(int&);  
void Check_supplies (int&, int&, int&, int&);  
void Deliver_services(int&, int&, int&, int&);  
void Dispense_cash_to_customer(int&);  
void Get_password(int&, int&, int&);  
void Get_required_service(int&, int, int, int, int);  
int Hardware_fault_detected();  
void Order_cheque_book_from_bank(int&);  
void Print_customer_balance(int&);  
void Print_customer_mini_statement(int&);  
void Process_cash_card(int, int* = 0);  
void Process_required_service(int, int&, int&, int&, int&);  
void Shutdown(void);  
  
*****  
*  
*      FUNCTION: main()  
*  
*****  
int main(void)  
{  
    int cheque_OK,  
        balance_OK,  
        statement_OK,  
        cash_OK,  
    int valid_password_entered, valid_card_entered;  
    int user_wants_quit, tries;  
    // 为模拟设置不重复的随机数  
    srand((unsigned)time(NULL));  
    // 模拟对列中有(最多)五个用户, 然后人工关闭  
    for (int customer = 1; customer <= 5; customer++)  
    {  
        // 欢迎标识  
        cout << "-----\n";  
        cout << "PROTOTYPE BANK AUTO TELLER MACHINE\n";  
        cout << "-----\n\n";  
        Check_supplies(cash_OK, balance_OK,  
                        statement_OK, cheque_OK);  
        Process_cash_card(CHECK_CARD, &valid_card_entered);  
        if (valid_card_entered)  
        {  
            tries = 0;  
            user_wants_to_quit = FALSE;  
            Get_password(valid_password_entered,  
                         user_wants_to_quit, tries);  
            if ((valid_password_entered)  
                && (!user_wants_to_quit))  
            {
```

```

    // 连续提供服务直到用户要求停止（或者硬件故障）
    Deliver_services(cheque_OK, balance_OK,
                      statement_OK, cash_OK);
    Process_cash_card(EJECT_CARD);
}
else
{
    // 没有正确的口令
    if (user_wants_to_quit)
    {
        // 礼貌的拒绝
        Process_cash_card(EJECT_CARD);
    }
}
}

}

// 模拟人工关闭
Shutdown();
return 0;
}

/*
*      FUNCTION: Shutdown()
*
*      概述: 退出信息, 并且退出程序。
*
*      PRE:      None.
*      POST:     None.
*
*      RETURNS: void.
*
*/
void Shutdown(void)
{
    // Pre - none
    cout << "Please use another machine.\n";
    cerr << "*** SIMULATING MACHINE SHUTDOWN ***\n";
    exit(1);
    // Post- none(!)
}

/*
*      FUNCTION: Check_supplies()
*
*      概述:      检查现金和纸张储量
*
*      PRE:      没有硬件故障。
*      POST:     得到有效数据。
*
*/

```

附录 A 设计表示法以及 CASE 工具

```
*      RETURNS: void.          *
*
***** ****
void Check_supplies(int& cash_OK_REF,
                     int& balance_OK_REF,
                     int& statement_OK_REF,
                     int& cheque_OK_REF)
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }
    int paper_level_OK; // 纸张高度是否正确? (标志)
    cout << "Next customer please wait...\\n\\n";
    Check_cash_reserves(cash_OK_REF);
    Check_paper_reserves(paper_level_OK);
    if (!paper_level_OK)
    {
        balance_OK_REF = FALSE;
        statement_OK_REF = FALSE;
    }
    else
    {
        // 复位这些用于新的顾客
        balance_OK_REF = TRUE;
        statement_OK_REF = TRUE;
    }
    // 复位标志用于新的顾客 (始终可用)
    cheque_OK_REF = TRUE;
    // Post - 得到有效数据
    assert((cash_OK_REF == FALSE)
           || (cash_OK_REF == TRUE));
    assert((balance_OK_REF == FALSE)
           || (balance_OK_REF == TRUE));
    assert((statement_OK_REF == FALSE)
           || (statement_OK_REF == TRUE));
    assert((cheque_OK_REF == FALSE)
           || (cheque_OK_REF == TRUE));
}
*****
*
*      FUNCTION: Hardware_fault_detected()          *
*
*      概述: 确信硬件没有损坏。                      *
*
*      PRE:      None.                            *
*      POST:     None.                           *
*
*      RETURNS:   TRUE (1) 如果故障, FALSE (0) 否则  *
*
*****
```

```

int Hardware_fault_detected()
{
    // Pre - none
    // 以 1/100 的概率模拟硬件故障
    if ((rand() % 100) == 0)
    {
        cerr << "*** SIMULATING HARDWARE FAULT ***\n";
        return 1;
    }
    else
    {
        return 0;
    }
    // Post - none
}

*****
*
*      FUNCTION: Check_cash_reserves()
*
*      概述: 检查现金高度。
*
*      PRE:      没有硬件故障
*      POST:     得到有效数据
*
*      RETURNS:   void
*
*****
void Check_cash_reserves(int& cash_OK_REF)
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }

    // 模拟 1/6 可能的现金不足
    if ((rand() % 6) == 0)
    {
        cash_OK_REF = FALSE;
        cerr << "*** SIMULATING LOW CASH LEVEL ***\n";
    }
    else
    {
        cash_OK_REF = TRUE;
        // Post - 得到有效数据
        assert((cash_OK_REF == FALSE)
               || (cash_OK_REF == TRUE));
    }
}

*****
*
*      FUNCTION: Check_paper_reserves()
*
*      概述: 检查是否有纸张供打印。
*
```

```

*
*      PRE:      没有硬件故障
*      POST:     得到有效数据
*
*      RETURNS:   void
*
*****void Check_paper_reserves(int& paper_level_OK_REF)
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }
    // 模拟 1/6 可能的纸张不足
    if ((rand() % 6)==0)
    {
        paper_level_OK_REF = FALSE;
        cerr << "**** SIMULATING LOW PAPER LEVEL **\n";
    }
    else
    {
        paper_level_OK_REF = TRUE;
    }
    // Post - 得到有效数据
    assert((paper_level_OK_REF == FALSE)
           || (paper_level_OK_REF == TRUE));
}
*****/*      FUNCTION: Process_cash_card()
*
*      概述: 验证、弹出或者保持自动提款卡。
*
*      PRE:      没有硬件故障, 有效操作请求
*      POST:     None.
*
*      RETURNS:  void.
*
*****void Process_cash_card(int required_op, int* card_OK_PTR)
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }
    // Pre - 有效操作请求
    assert((required_op == CHECK_CARD) ||
           (required_op == KEEP_CARD) ||
           (required_op == EJECT_CARD));
    /* 注意: Process_cash_card() 是一个有默认参数的函数, 能够如下调用:
       Process_cash_card(CHECK_CARD, card_OK);
       或
       Process_cash_card(KEEP_CARD); */
}

```

```

Process_cash_card(EJECT_CARD);    */
switch (required_op)
{
case CHECK_CARD:
    // 使用“额外的”参数
    cout << "Please insert your card.\n";
    // 验证用户的提款卡
    *card_OK_PTR = (((rand() % 6) != 0)
                    ? (TRUE) : (FALSE));
    if (*card_OK_PTR)
    {
        cerr <<
        "**** SIMULATING CORRECT CARD INSERTED ***\n";
        cout << "Thank you.\n\n";
    }
    else
    {
        cerr <<
        "**** SIMULATING *INCORRECT* CARD INSERTED ***\n";
        cout << "Incorrect card.\n";
        Process_cash_card(EJECT_CARD);
    }
    break;
case KEEP_CARD:
    cout << "Password incorrect. Card retained.\n";
    cout << "Please contact your branch.\n";
    cerr << "**** SIMULATING SWALLOWING CARD ***\n\n";
    break;
case EJECT_CARD:
    cout << "Thank you for using this machine.\n";
    cout << "**** SIMULATING EJECTING CARD ***\n\n";
    break;
}
// Post - none
}

/*********************************************
*
*   FUNCTION: Get_password()
*
*   概述: 得到并验证用户的口令 PRE: 没有硬件故障。
*
*   PRE:      没有硬件故障, 有效操作请求
*   POST:     得到有效数据
*
*   RETURNS:   void
*
*****************************************/
void Get_password(int& password_OK_REF,
                  int& want_to_quit_REF,

```

```
int& tries_REF)
{
// Pre -没有硬件故障
if (Hardware_fault_detected()) { Shutdown(); }
int password;
tries_REF++;
cout << "Please enter your password"
    << "\n(or 9999 to simulated QUIT button): ";
cin >> password;
if (!(password == 9999))
{
    //
    // 用户没有要求停止
    // 模拟向银行计算机呼叫
    cerr << "**** SIMULATING CALL TO BANK COMPUTER ****\n";
    // 模拟一个正确的口令
    // 实际上向银行计算机发送帐号和口令
    // 计算机返回“YES”或者“NO”
    if (password == 1234)
    {
        password_OK_REF = TRUE;
    }
    else
    {
        password_OK_REF = FALSE;
    }
    if (!(password_OK_REF))
    {
        if (tries_REF < MAX_NUM_PASS_TRIES)
        {
            cout << "Password incorrect.\n";
            // 循环调用 Get_password()
            Get_password(password_OK_REF, want_to_quit_REF, tries_REF);
        }
        else
        {
            // 尝试次数太多-是否是偷来的卡?
            Process_cash_card(KEEP_CARD);
        }
    }
    else
    {
        count << "Thank you.\n\n";
    }
}
else
{
    // 用户希望离开
    want_to_quit_REF = TRUE;
}
// Post - 得到有效数据。
```

```

assert((password_OK_REF == FALSE)
    || (password_OK_REF == TRUE));
assert((want_to_quit_REF == FALSE)
    || (want_to_quit_REF == TRUE));
assert(tries_REF <= MAX_NUM_PASS_TRIES);
}

*****
*
*      FUNCTION: Deliver_serivces()
*
*      概述:询问用户所要求的服务和传送它。
*
*      PRE:      没有硬件故障得到有效数据
*      POST:     None.
*
*      RETURNS: void.
*
*****
```

void Deliver_services(int& cheque_OK_REF,
 int& balance_OK_REF,
 int& statement_OK_REF,
 int& cash_OK_REF)

{

// Pre - 没有硬件故障

if (Hardware_fault_detected()) { Shutdown(); }

// Pre - 得到有效数据

assert((cash_OK_REF == FALSE)
 || (cash_OK_REF == TRUE));
assert((cheque_OK_REF == FALSE)
assert((balance_OK_REF == FALSE))
 || (balance_OK_REF = TRUE));
 || (cheque_OK_REF == TRUE));
assert((statement_OK_REF == FALSE)
 || (statement_OK_REF == TRUE));
int user_choice;
Get_required_service(user_choice, cheque_OK_REF,
 balance_OK_REF, statement_OK_REF, cash_OK_REF);
while (use)choice != QUIT)
{
 Process_required_service(user_choice, cheque_OK_REF,
 balance_OK_REF, statement_OK_REF, cash_OK_REF);
 Get_required_service(user_choice, cheque_OK_REF,
 balance_OK_REF, statement_OK_REF, cash_OK_REF);
}
// Post - none
}

*
* FUNCTION: Get_required_service()

```

/*
 * 概述：没有硬件故障得到有效的选择。
 *
 * PRE:      没有硬件故障
 * POST:     Got valid choice
 *
 * RETURNS:  void
 *
 *****/
void Get_required_service(int& service_REF,
                           int& cheque_OK,
                           int& balance_OK,
                           int& statement_OK,
                           int& cash_OK)
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }

    int service;
    cout << "Which service do you require: \n";
    cout << "Quit                         Type 1\n";
    if (cheque_OK)   cout << "Order cheque book           Type 2\n";
    if (balance_OK)  cout << "Print balance                 Type 3\n";
    if (statement_OK) cout << "Print statement                Type 4\n";
    if (cash_OK)     cout << "Cash                          Type 5\n";
    do {
        cout << "Please enter the appropriate number : ";
        cin >> service;
    } while ((service > 5)
             || (service < 1)
             || (Service == 2) && (!cheque_OK))
             || (Service == 3) && (!balance_OK))
             || (Service == 4) && (!statement_OK))
             || (Service == 5) && (!cash_OK));
    cout << "Thank you.\n\n";
    switch (service)
    {
        case 1: service_REF = QUIT; break;
        case 2: service_REF = ORDER_CHEQUE_BOOK; break;
        case 3: service_REF = PRINT_BALANCE; break;
        case 4: service_REF = PRINT_MINI_STATEMENT; break;
        case 5: service_REF = DISPENSE_CASH; break;
    }
    // Post - 得到有效的选择
    assert((service_REF == QUIT) ||
           (service_REF == ORDER_CHEQUE_BOOK) ||
           (service_REF == PRINT_BALANCE) ||
           (service_REF == PRINT_MINI_STATEMENT) ||
           (service_REF == DISPENSE_CASH));
}

```

```

*****
*
*      FUNCTION: Process_required_service()
*
*      概述: 执行用户请求的服务。
*
*      PRE:          没有硬件故障得到有效的选择
*      POST:         None.
*
*      RETURNS:      void.
*
*****
void Process_required_service(int user_choice,
                               int& cheque_OK_REF,
                               int& balance_OK_REF,
                               int& statement_OK_REF,
                               int& cash_OK_REF,
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }
    // Pre - 得到有效的选择
    assert((user_choice == QUIT) ||
           (user_choice == ORDER_CHEQUE_BOOK) ||
           (user_choice == PRINT_BALANCE) ||
           (user_choice == PRINT_MINI_STATEMENT) ||
           (user_choice == DISPENSE_CASH));
    switch (user_choice)
    {
        case ORDER_CHEQUE_BOOK:
            Order_cheque_book_from_bank(cheque_OK_REF);
            break;
        case PRINT_BALANCE:
            Print_customer_balance(balance_OK_REF);
            break;
        case PRINT_MINI_STATEMENT:
            Print_customer_mini_statement(statement_OK_REF);
            break;
        case DISPENSE_CASH:
            Dispense_cash_to_customer(cash_OK_REF);
            break;
    }
    // Post - none
}
*****
*
*      FUNCTION: Order_cheque_book_from_bank()
*
*      概述: 要求支票本。
*

```

```

*      PRE:      没有硬件故障得到有效的选择      *
*      POST:     返回有效数据      *
*
*      RETURNS:   void      *
*      ****
void Order_cheque_book_from_bank(int& cheque_OK_REF)
{
    // Pre - 没有硬件故障
    if(Hardware_fault_detected()) { Shutdown(); }
    // Pre - 得到有效数据
    assert((cheque_OK_REF == TRUE));
    cerr << "**** SIMULATING ORDER FOR CHEQUE BOOK ***\n";
    cout << "Your cheque book will be posted to you.\n\n";
    cheque_OK_REF = FALSE;
    // Post - 返回有效数据
    assert((cheque_OK_REF == FALSE));
}
/***
*      FUNCTION: Print_customer_balance()      *
*
*      概述: 打印顾客的结余。      *
*
*      PRE:      没有硬件故障得到有效选择      *
*      POST:     返回有效数据      *
*
*      RETURNS:   void      *
*      ****
void Print_customer_balance(int& balance_OK_REF)
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }
    // Pre - 得到有效数据
    assert((balance_OK_REF == TRUE));
    cout << "Your balance is: 99.99 (** SIMULATED **)\n\n";
    balance_OK_REF = FALSE;
    // Post - 返回有效数据
    assert((balance_OK_REF == FALSE));
}
/***
*      FUNCTION: Print_customer_mini_statement()      *
*
*      概述: 打印 m - in - i 结单。      *
*
*      PRE:      没有硬件故障得到有效选择      *
*      POST:     返回有效数据      *

```

```

*
*      RETURNS:    void
*
***** ****
void Print_customer_mini_statement(int& statement_OK_REF)
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }
    // Pre - 得到有效数据
    assert((statement_OK_REF == TRUE));
    cerr << "**** SIMULATING PRINTING OF MINI STATEMENT ****";
    cout << "\nPlease take your mini statement.\n\n";
    statement_OK_REF = FALSE;
    // Post - 返回有效数据
    assert((statement_OK_REF == FALSE));
}
***** ****
*
*      FUNCTION: Print_Dispense_cash_to_customer()
*
*      概述: 交付现金。
*
*      PRE:        没有硬件故障得到有效选择
*      POST:       返回有效数据
*
*      RETURNS:    void
*
***** ****
void Dispense_cash_to_customer(int& cash_OK_REF)
{
    // Pre - 没有硬件故障
    if (Hardware_fault_detected()) { Shutdown(); }
    // Pre - 得到有效数据
    assert((cash_OK_REF == TRUE));
    cerr << "**** SIMULATING CASH DELIVERY ***\n";
    cout << "Please take your cash.\n\n";
    cash_OK_REF = FALSE;
    // Post - 返回有效数据
    assert((cash_OK_REF == FALSE));
}
***** ****
*          *** 程序结束 ***
***** ****

```

源程序清单 A1.1 自动出纳机模型的系统软件

[G e n e r a l I n f o r m a t i o n]

书名 = 时间触发嵌入式系统设计模式 8051系列微控制器开发可靠应用

作者 =

页数 = 762

S S号 = 0

出版日期 =

封面
书名
版权
前言
目录
目录

前言
序言

绪 言

第1章什么是时间触发的嵌入式系统

- 1 . 1 引言
- 1 . 2 息系统
- 1 . 3 桌面系统
- 1 . 4 实时系统
- 1 . 5 嵌入式系统
- 1 . 6 事件触发系统
- 1 . 7 时间触发系统
- 1 . 8 小结

第2章使用模式来设计嵌入式系统

- 2 . 1 引言
- 2 . 2 现有软件设计技术的局限
- 2 . 3 模式
- 2 . 4 时间触发嵌入式系统模式
- 2 . 5 小结

第1篇硬件基础

第3章8051系列微控制器

- 引言
- 标准8051
- 精简8051
- 扩展8051

第4章振荡器硬件

- 引言
- 晶体振荡器
- 陶瓷谐振器

第5章硬件复位

- 引言
- 阻容复位
- 可靠的复位

第6章存储器问题

- 引言
- 片内存储器

片外数据存储器

片外程序存储器

第 7 章 直流负载驱动

引言

直接 L E D 驱动

直接负载驱动

I C 缓冲放大器

B J T (双极结型三极管) 驱动器

I C 驱动器

M O S F E T 驱动器

固态继电器驱动 (直流)

第 8 章 交流负载驱动

引言

电磁继电器驱动

固态继电器驱动 (交流)

第 2 篇 软件基础

第 9 章 基本的软件体系结构

引言

超级循环

项目头文件

第 10 章 使用端口

引言

端口输入 / 输出

端口头文件

第 11 章 延迟

引言

硬件延迟

软件延迟

第 12 章 看门狗

引言

硬件看门狗

第 3 篇 单处理器系统的时间触发结构

第 13 章 调度器的介绍

13.1 引言

13.2 桌面操作系统

13.3 对超级循环结构的评价

13.4 更好的解决方案

13.5 例子：闪烁 L E D

13.6 在不同的时间间隔执行多个任务

13.7 什么是调度器

13.8 合作式调度和抢占式调度

1 3 . 9 抢占式调度器详解

1 3 . 1 0 小结

1 3 . 1 1 进阶阅读

第 1 4 章 合作式调度器

引言

合作式调度器

第 1 5 章 学会以合作的方式思考

引言

循环超时

硬件超时

第 1 6 章 面向任务的设计

引言

多级任务

多状态任务

第 1 7 章 混合式调度器

引言

混合式调度器

第 4 篇 用户界面

第 1 8 章 通过 R S - 2 3 2 与 P C 通信

引言

P C 连接 (R S - 2 3 2)

第 1 9 章 开关接口

引言

开关接口 (软件)

开关接口 (硬件)

通断开关

多状态开关

第 2 0 章 键盘接口

引言

键盘接口

第 2 1 章 多路复用 L E D 显示

引言

多路复用 L E D 显示

第 2 2 章 控制 L C D 显示面板

引言

字符型 L C D 板

第 5 篇 使用串行外围模块

第 2 3 章 使用 I 2 C 外围模块

引言

I 2 C 外围模块

第 2 4 章 使用 S P I 外围模块

引言

S P I 外围模块

第6篇 多处理器系统的时间触发体系结构

第25章共享时钟调度器的介绍

25.1 引言

25.2 额外 C P U 性能和外围硬件

25.3 模块化设计的优点

25.4 怎样连接多个处理器

25.5 为什么增加处理器并不一定能改善可靠性

25.6 小结

第26章使用外部中断的共享时钟调度器

引言

共享时钟中断调度器(时标)

共享时钟中断调度器(数据)

第27章使用U A R T(通用异步收发器)的共享时钟调度器

引言

使用U A R T的共享时钟调度器(本地)

使用U A R T的共享时钟调度器(R S - 2 3 2)

使用U A R T的共享时钟调度器(R S - 4 8 5)

第28章使用C A N的共享时钟调度器

引言

共享时钟C A N调度器

第29章多处理器系统的设计

引言

数据联合

长任务

多米诺骨牌任务

第7篇监视与控制组件

第30章脉冲频率检测

引言

硬件脉冲计数

软件脉冲计数

第31章脉冲频率调制

引言

硬件脉冲频率调制

软件脉冲频率调制

第32章模拟-数字转换器(A D C)的应用

引言

单次模数转换

模数转换前置放大器

序列模数转换

A - A 濾波器

电流传感器

第3 3章脉冲宽度调制

引言

硬件脉宽调制

脉宽调制信号平滑滤波

3 级脉宽调制

软件脉宽调制

第3 4章数模转换器的应用 (D A C)

引言

数模转换输出

数模转换平滑滤波

数模转换驱动

第3 5章进行控制

引言

P I D 控制器

第8篇特殊的时间触发结构

第3 6章减少系统开销

引言

2 5 5 - 时标调度器

单任务调度器

一年调度器

第3 7章提高调度的稳定性

引言

稳定调度器

结 论

第3 8章本书试图实现的目标

3 8 . 1 引言

3 8 . 2 本书试图实现的目标

3 8 . 3 小结

第3 9章收集的参考文献和书目

3 9 . 1 出版书刊一览表

3 9 . 2 其他模式

3 9 . 3 实时嵌入式系统设计技术

3 9 . 4 高可靠性系统设计技术

3 9 . 5 8 0 5 1 微控制器

3 9 . 6 作者的相关著作

附 录

A 设计表示法以及C A S E 工具

概述

C A S E 工具

表示法
B C D 指南
概述
C D 的主要内容
本书的源代码
C 互联网站点指南
概述
U R L
互联网站点上的内容
错误报告和程序代码更新