

Patterns for Time-Triggered Embedded Systems

Michael J Pont

Building reliable applications
with the 8051 family of
microcontrollers

Foreword by Kent Beck

TTE Systems

Patterns for time-triggered embedded systems

ACM PRESS BOOKS

This book is published as part of the ACM Press Books – a collaboration between the Association for Computing Machinery and Addison-Wesley. ACM is the oldest and largest education and scientific society in the information technology field. Through its high-quality publications and services, ACM is a major force in advancing the skills and knowledge of IT professionals throughout the world. For further information about ACM contact:

ACM Member Services
1515 Broadway, 17th Floor
New York NY 10036-5701
Phone: +1 212 626 0500
Fax: +1 212 944 1318
Email: acmhelp@acm.org

ACM European Service Center
108 Cowley Road
Oxford OX4 1JF
United Kingdom
Phone: +44 1865 382338
Fax: +44 1865 381338
Email: acm-europe@acm.org
URL: <http://www.acm.org>

SELECTED ACM TITLES:

Software Requirements and Specification: A Lexicon of Software Practice, Principles and Prejudices *Michael Jackson*

Software Test Automation: Effective Use of Text Execution Tools *Mark Fewster and Dorothy Graham*

Test Process Improvement: A Practical Step-by-step Guide to Structured Testing *Tim Koomen and Martin Pol*

Mastering the Requirements Process *Suzanne Robertson and James Robertson*

Bringing Design to Software: Expanding Software Development to Include Design *Terry Winograd, John Bennett, Laura de Young, Bradley Hartfield*

Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design *Larry L. Constantine and Lucy A. D. Lockwood*

Problem Frames: Analyzing and Structuring Software Development Problems *Michael Jackson*

Software Blueprints: Lightweight Uses of Logic in Conceptual Modelling *David Robertson and Jaume Agustí*

Patterns for time-triggered embedded systems

**Building reliable applications with
the 8051 family of microcontrollers**

Michael J. Pont



The Keil compiler (demo) and associated files on the CD-ROM enclosed with this book have been authored and developed by Keil (UK) Ltd.
© Keil (UK) Ltd 2001.



An imprint of Pearson Education

Harlow, England · London · New York · Reading, Massachusetts · San Francisco
Toronto · Don Mills, Ontario · Sydney · Tokyo · Singapore · Hong Kong · Seoul
Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan



PEARSON EDUCATION LIMITED

Head Office:
Edinburgh Gate
Harlow CM20 2JE
Tel: +44 (0)1279 623623
Fax: +44 (0)1279 431059
Websites: www.it-minds.com
www.aw.com/cseng/

London Office:
128 Long Acre
London WC2E 9AN
Tel: +44 (0)20 7447 2000
Fax: +44 (0)20 7240 5771

First published in Great Britain in 2001

© ACM Press 2001

ISBN 0 201 33138 1

The right of Michael Pont to be identified as Author of this Work has been asserted by him in accordance with the Copyright, Designs, and Patents Act 1988.

All rights reserved; no part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without either the prior written permission of the Publishers or a licence permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1P 0LP. This book may not be lent, resold, hired out or otherwise disposed of by way of trade in any form of binding or cover other than that in which it is published, without the prior consent of the Publishers.

The programs in this book have been included for their instructional value. The publisher does not offer any warranties or representations in respect of their fitness for a particular purpose, nor does the publisher accept any liability for any loss or damage arising from their use.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Pearson Education Limited has made every attempt to supply trademark information about manufacturers and their products mentioned in this book.

The publishers wish to thank the following for permission to reproduce the material: Arizona Microchip Technology Ltd; Allegro Microsystems; Atmel Corporation; Infineon; Philips Semiconductors; Texas Instruments.

Two of the figures in this book (Figure 3.2 and 3.4) reproduce information provided by Atmel Corporation. Atmel® warrants that it owns these materials and all intellectual property related thereto. Atmel, however, expressly and explicitly excludes all other warranties, insofar as it relates to this book, including accuracy or applicability of the subject matter of the Atmel materials for any purpose.

British Library Cataloguing-in-Publication Data

A CIP catalogue record for this book can be obtained from the British Library.

Library of Congress Cataloging in Publication Data
Applied for.

10 9 8 7 6 5 4 3 2 1

Designed by Claire Brodmann Book Designs, Lichfield, Staffs

Typeset by Pantek Arts Ltd, Maidstone, Kent.

Printed and bound in the United States of America.

The Publishers' policy is to use paper manufactured from sustainable forests.

**This book is dedicated to my parents,
Barbara and Gordon Pont**

Contents

Foreword *page* xiv

Preface xvi

Introduction 1

1 What is a time-triggered embedded system? 3

- 1.1 Introduction 3
- 1.2 Information systems 3
- 1.3 Desktop systems 5
- 1.4 Real-time systems 6
- 1.5 Embedded systems 8
- 1.6 Event-triggered systems 10
- 1.7 Time-triggered systems 11
- 1.8 Conclusions 14

2 Designing embedded systems using patterns 15

- 2.1 Introduction 15
- 2.2 Limitations of existing software design techniques 17
- 2.3 Patterns 22
- 2.4 Patterns for time-triggered systems 24
- 2.5 Conclusions 25

Part A Hardware foundations 27

3 The 8051 microcontroller family 29

- STANDARD 8051 30
- SMALL 8051 41
- EXTENDED 8051 46

4 Oscillator hardware 53

- CRYSTAL OSCILLATOR 54
- CERAMIC RESONATOR 64

5 Reset hardware 67

RC RESET 68

ROBUST RESET 77

6 Memory issues 81

ON-CHIP MEMORY 82

OFF-CHIP DATA MEMORY 94

OFF-CHIP CODE MEMORY 100

7 Driving DC loads 109

NAKED LED 110

NAKED LOAD 115

IC BUFFER 118

BJT DRIVER 124

IC DRIVER 134

MOSFET DRIVER 139

SSR DRIVER (DC) 144

8 Driving AC loads 148

EMR DRIVER 149

SSR DRIVER (AC) 156

Part B Software foundations 159**9 A rudimentary software architecture** 161

SUPER LOOP 162

PROJECT HEADER 169

10 Using the ports 173

PORT I/O 174

PORT HEADER 184

11 Delays 193

HARDWARE DELAY 194

SOFTWARE DELAY 206

12 Watchdogs	215
HARDWARE WATCHDOG	217

Part C Time-triggered architectures for single-processor systems 229

13 An introduction to schedulers	231
13.1 Introduction	231
13.2 The desktop OS	231
13.3 Assessing the super loop architecture	233
13.4 A better solution	235
13.5 Example: Flashing an LED	239
13.6 Executing multiple tasks at different time intervals	243
13.7 What is a scheduler?	245
13.8 Co-operative and pre-emptive scheduling	246
13.9 A closer look at pre-emptive schedulers	250
13.10 Conclusions	253
14 Co-operative schedulers	254
CO-OPERATIVE SCHEDULER	255
15 Learning to think co-operatively	297
LOOP TIMEOUT	298
HARDWARE TIMEOUT	305
16 Task-oriented design	316
MULTI-STAGE TASK	317
MULTI-STATE TASK	322
17 Hybrid schedulers	332
HYBRID SCHEDULER	333

Part D The user interface 359

18 Communicating with PCs via RS-232	361
PC LINK (RS-232)	362

19	Switch interfaces	397
	SWITCH INTERFACE (SOFTWARE)	399
	SWITCH INTERFACE (HARDWARE)	410
	ON-OFF SWITCH	414
	MULTI-STATE SWITCH	423
20	Keypad interfaces	433
	KEYPAD INTERFACE	434
21	Multiplexed LED displays	449
	MX LED DISPLAY	450
22	Controlling LCD panels	465
	LCD CHARACTER PANEL	467

Part E Using serial peripherals 491

23	Using 'I²C' peripherals	493
	I ² C PERIPHERAL	494
24	Using 'SPI' peripherals	520
	SPI PERIPHERAL	521

Part F Time-triggered architectures for multiprocessor systems 537

25	An introduction to shared-clock schedulers	539
25.1	Introduction	539
25.2	Additional CPU performance and hardware facilities	539
25.3	The benefits of modular design	541
25.4	How do we link more than one processor	543
25.5	Why additional processors may not always improve reliability	550
25.6	Conclusions	552

26 Shared-clock schedulers using external interrupts	553
SCI SCHEDULER (TICK)	554
SCI SCHEDULER (DATA)	593
27 Shared-clock schedulers using the UART	608
SCU SCHEDULER (LOCAL)	609
SCU SCHEDULER (RS-232)	642
SCU SCHEDULER (RS-485)	646
28 Shared-clock schedulers using CAN	675
SCC SCHEDULER	677
29 Designing multiprocessor applications	711
DATA UNION	712
LONG TASK	716
DOMINO TASK	720

Part G Monitoring and control components 725

30 Pulse-rate sensing	727
HARDWARE PULSE COUNT	728
SOFTWARE PULSE COUNT	736
31 Pulse-rate modulation	741
HARDWARE PRM	742
SOFTWARE PRM	748
32 Using analogue-to-digital converters (ADCs)	756
ONE-SHOT ADC	757
ADC PRE-AMP	777
SEQUENTIAL ADC	782
A-A FILTER	794
CURRENT SENSOR	802
33 Pulse-width modulation	807
HARDWARE PWM	808

PWM SMOOTHER	818
3-LEVEL PWM	822
SOFTWARE PWM	831
34 Using digital-to-analog converters (DACs)	840
DAC OUTPUT	841
DAC SMOOTHER	853
DAC DRIVER	857
35 Taking control	860
PID CONTROLLER	861

Part H Specialized time-triggered architectures 891

36 Reducing the system overheads	893
255-TICK SCHEDULER	894
ONE-TASK SCHEDULER	911
ONE-YEAR SCHEDULER	919
37 Increasing the stability of the scheduling	931
STABLE SCHEDULER	932

Conclusions 941

38 What this book has tried to do	941
38.1 Introduction	943
38.2 What this book has tried to do	943
38.3 Conclusions	943
39 Collected references and bibliography	946
39.1 Complete list of publications	946
39.2 Other pattern collections	952
39.3 Design techniques for real-time/embedded systems	952
39.4 Design techniques for high-reliability systems	953
39.5 The 8051 microcontroller	954
39.6 Related publications by the author	954

Appendices 955

A	The design notation and CASE tool	957
Overview	957	
The CASE tool	957	
The notation	957	
B	Guide to the CD	980
Overview	980	
The basis of the CD	980	
The source code for this book	980	
C	Guide to the WWW site	982
Overview	982	
The URL	982	
Contents of the WWW site	982	
Bug reports and code updates	982	
	Index	985

Foreword

You hold in your hands a pattern language. If you could measure its distance from the topics of my patterns, you would find a sizeable gap. In spirit, however, Michael is right on target.

Ward Cunningham and I worked during the early days of the commercialization of Smalltalk. Smalltalk had been designed from the beginning to be a seamless environment. You could be using a word processor written in Smalltalk, start up a debugger, modify the program and continue typing.

Some of the first Tektronix customers for Smalltalk were pretty odd. We often talked about Ray, an old guy from a big chemical company who latched onto Smalltalk and really made it jump and run, presenting and manipulating experimental data. Watching one of his demos was a delight, because he was so proud of what he had accomplished.

Reading Ray's code was another matter entirely. He would do anything and everything, no matter how hideous, to get his programs to work. The result was a mess that was totally unmaintainable and only used a fraction of the power of Smalltalk.

We often used Ray as the personification of the audience we wanted for our software – people who have problems to solve and have to construct software to solve them. We could easily contrast this utilitarian attitude with our ‘no compromise engineering’ attitude towards software, where the simplicity and elegance of the solution were more important than the problem solved. We could see that if we wanted to affect the world, we couldn’t just pursue our visions of beauty, we would have to try to help Ray at the same time.

The resulting pattern language was a curious blend of high-minded advice ('never use a computer you can't personally turn off') and banal bookkeeping chores ('have the braces in your source code form rectangles'). The intent was to help Ray get more out of Smalltalk. In this we largely failed. Looking at my career since then, I have drifted more and more to giving advice to people coaching people who write programs for people who write programs for...

That's why I loved reading Michael's early draft. It brought back that feeling of opening up a field of endeavour to someone who just has a problem to solve and who doesn't want to be an expert in the solution. Now I'm Ray. I'd love to whip together little microcontrollers to solve various problems (okay, so I'm a nerd). Reading this pattern language gives me the confidence that I could do just that.

Far from just giving me the smell of rosin in my nose and the feel of a wire wrap gun in my hand, these patterns stand as an example of how much more can be done with patterns than is commonly attempted. Patterns at their best bridge the gap between

problem and solution. They connect human needs and emotions with technology. And they open up new possibilities for people who just have a problem to solve.

Fire up your soldering iron and enjoy.

Kent Beck

*Three Rivers Institute
Merlin, Oregon*

Preface

Embedded software is ubiquitous. It forms a core component of an enormous range of systems, from aircraft, passenger cars and medical equipment, to children's toys, video recorders and microwave ovens. This book provides a complete and coherent set of software patterns to support the development of this type of application.

The remainder of this preface attempts to provide answers to more detailed questions which prospective readers may have about the contents.

I What are the key features of this book?

- The focus is on the rapid development of software for time-triggered, embedded systems, using software patterns. The meaning of 'time triggered' is explained in Chapter 1; software patterns are introduced in Chapter 2.
- The systems are all based on microcontrollers, from the widely used 8051 family. This vast family of 8-bit devices is manufactured by a number of companies, including Philips, Infineon, Atmel, Dallas, Texas Instruments and Intel. The range of different 8051 microcontrollers available is reviewed in Chapter 3.
- Time-triggered techniques are the usual choice in safety-related applications, where reliability is a crucial design requirement. However, the need for reliability is not restricted to systems such as drive-by-wire passenger cars, aerospace systems or monitoring systems for industrial robots: even at the lowest level, an alarm clock that fails to sound on time or a video recorder that operates intermittently may not have safety implications but, equally, will not have high sales figures. The patterns presented here allow time-triggered techniques to be simply and cost-effectively applied in virtually any embedded project.
- The applications discussed in detail must carry out tasks or respond to events over time intervals measured in milliseconds. This level of response can be economically and reliably achieved, even with an 8-bit microcontroller, using the approaches discussed in this book.
- The software is implemented entirely in 'C'. All of the examples in the book appear, in full, on the enclosed CD.
- The book is supported by a WWW site which includes, among other features, a wide range of detailed case studies, additional technical information and links to sources of further information (<http://www.engg.lse.ac.uk/books/Pont>).

II How do you build time-triggered embedded systems?

- The time-triggered systems in this book are created using schedulers. Briefly, a scheduler is a very simple ‘operating system’ suitable for use in embedded applications (see Chapter 13 for a detailed introduction to this topic).
- A range of complete scheduler architectures for applications involving a single microcontroller is described and illustrated (Chapters 14 to 17). Complete source code for a number of different schedulers is included on the CD.
- Like an increasing number of applications, many of the systems presented here involve the use of more than one microcontroller: a range of shared-clock scheduler architectures that can support this type of application is described (Chapters 25 to 29). Many of these systems make use of popular serial standards, including the CAN bus and RS-485.
- A selection of more specialized scheduler architectures is also presented (in Part H). This includes a ‘stable’ scheduler that can provide very precise timing over long periods, a scheduler optimized to run a single task and general-purpose schedulers designed for low-power and/or low-memory applications (see Chapters 36 and 37).

III What other topics are discussed in the book?

- All embedded systems involve some hardware design and suitable hardware foundations are presented. These include designs for oscillator and reset circuits and techniques for connecting external ROM and RAM memory (see Chapters 4, 5 and 6). These also include interface circuits suitable for use with low- and high-voltage DC and AC loads (see Chapters 7 and 8).
- Suitable software foundations are also presented, including a simple architecture for embedded applications (Chapter 9), techniques for controlling port pins (Chapter 10), techniques for generating delays (Chapter 11) and techniques for using watchdog timers (Chapter 12).
- A key part of the user interface of some embedded applications is an RS-232 link to a desktop or notebook PC, while many other embedded systems have a user interface created using an LCD or LED display along with a small collection of switches and/or a keypad. Techniques for working with these different interface components are presented in Chapters 18 to 22.
- Many modern different peripheral devices (LCDs, LED displays, EEPROMs, A-D and D-A devices and so on) now have a serial interface, with the result that these devices can be connected to a microcontroller without consuming large numbers of port pins. Complete software libraries for the two main serial communication protocols (I²C and SPI) are presented in Chapters 23 and 24.
- Techniques suitable for use in condition monitoring and control applications are presented in Part G. This includes a discussion of ‘PID control’. Again, detailed code libraries are provided (Chapter 30 to Chapter 35).

IV Who should read this book?

I had three main groups of people in mind as I wrote this book:

- Software engineers with previous experience of desktop systems now beginning to work with embedded systems.
- Hardware engineers who wish to understand more about the software issues involved in the development of embedded systems.
- University and college students on 'electronic and software engineering', 'software engineering', 'computer science', 'electronic engineering' or similar programmes who are taking advanced modules in embedded systems.

It must be emphasized that this book is not intended for those requiring an introduction to programming and it is expected that readers will have previously developed 'desktop' software applications, using C, C++ or a similar high-level language. Readers with less experience in this area may find it useful to have a copy of an introductory book on 'C', such as Herbert Schildt's *Teach Yourself C* (Schildt, 1997)¹ by their side as they read this book.

Similarly, some familiarity with the principles of software design is assumed. Here, some experience with 'object-oriented' design, and 'process-oriented' design ('structured analysis') will be useful. Readers with less experience in this area may find it useful to have a copy of my previous introductory book on software design (Pont, 1996) by their side.

Finally, some very basic electronics knowledge is also useful. Readers without hardware design experience may find it useful to have available a copy of *The Art of Electronics* (Horowitz and Hill, 1989).

In most cases, readers with previous desktop programming experience, some familiarity with 'dataflow diagrams' or 'UML' and some rudimentary hardware knowledge will have little difficulty with the material presented here. Please note that no knowledge of software patterns is assumed.

V What type of microcontroller hardware is used?

The market for microcontrollers is vast. Most current estimates suggest that, for every processor sold for a desktop PC, 100 microcontrollers are sold for embedded systems.

As the sub-title suggests, this book focuses on the 8051 family of microcontrollers, which was originally developed by Intel, but is now produced, in more than 300 different forms, by a wide range of companies, including Philips, Infineon, Atmel and Dallas. The use of the 8051 family is no accident. Together, sales of this vast family are estimated to account for more than 50% of the 8-bit microcontroller market and to have the largest share (around 30%) of the microcontroller market as a whole.

1. Details of sources referred to in the text are given in Chapter 39

Note that in this book I consider not only recent versions of the ‘standard’ 8051 (4 ports, 40/44 pins: e.g. the Atmel 89C52; Dallas 89C420; Infineon C501; Philips 89CRD2), but the full range of modern devices, including the ‘small’ 8051s (two ports, 20/24 pins: e.g. the Atmel 89C4051; Philips 87LPC764) and the ‘extended’ 8051s (up to ten ports, ~100 pins, CAN, ADC, etc. on chip: e.g. Infineon C509; Infineon C515c; Dallas 80c390).

Please note: *The code associated with this book is written entirely in ‘C’: you will find it straightforward to translate the code for use on a different hardware platform should you wish to do so.*

VI What’s on the CD?

The CD includes complete source code files for all the software patterns: as mentioned above, **all** of this code is in the ‘C’ programming language.

The source code for these patterns is fully compatible with the industry-standard Keil C compiler. An evaluation version of this compiler, and a complete hardware simulator, is also included on the CD: this allows the majority of the patterns to be explored on a desktop PC without the need to purchase or construct any hardware at all.

Finally, data sheets (in PDF format) for a large number of 8051 microcontroller are also included on the CD.

VII What about the WWW site?

There is a WWW site associated with this book, at the following URL:

<http://www.engg.lse.ac.uk/books/Pont>

On this site you will find:

- A set of detailed case studies describing the application of the techniques discussed in this book in a series of small and large projects.
- Bug reports and code updates (please see section X, which follows).
- Further code samples.
- Links to other relevant sites.

VIII Is the code ‘free ware’?

The code included in this book took many years to produce. It is not ‘free ware’ and is subject to some simple copyright restrictions. These are as follows:

- Having purchased a copy of this book, you are entitled to use the code listed in this book and included on the CD in your projects, should you choose to do so. If you use the code in this way, then no run-time royalties are due. However, I would appreciate it if you acknowledged the source of the code in the product documentation.

- If there are ten developers in your team using code adapted from this book, please purchase ten copies of the book.
- You may not, **under any circumstances**, publish any of the source code included in the book or on the CD, in any form or by any means, without explicit written authorization from me. If you wish to publish limited code fragments then, in most circumstances, I will grant this permission, subject only to an appropriate acknowledgement accompanying the published material. If you wish to publish more substantial code listings, then payment of a fee may be required. Please contact me for further details.

IX How should this book be read?

While writing this book, I had two types of reader in mind: those who like to read a book from cover to cover and those who prefer to treat a book like this as a reference source, to be first skim read and then opened, as needed, during the course of a project.

To match the needs of the cover-to-cover readers, the material follows in a logical order, from the introductory and foundation material, through to more advanced material. To make it easy to read in this way, I have tried to ensure that the delivery of information is as sequential as possible: that is, that the material needed to understand (say) Chapter 14 is presented in Chapters 1 to 13.

For use as a work of reference, I suggest that readers first read (or at least skim) the introductory chapters (1 and 2, plus 3, 9, 13 and 25): together, these chapters will provide a good overview of the material presented elsewhere in the book.

X What about bug reports and code updates?

There is huge amount of code involved in this project, both in the book itself and on the associated CD. I have personally tested all of the code that appears here. Nonetheless, errors can creep in.

If you think you have found a bug, please first check the WWW site (see earlier section VII), to see if anyone else has picked up the error: if they have, a code correction will have been made available.

If you have found a bug not listed on the WWW site, please send me an e-mail (the address is at the end of this preface) and I will do my best to help.

I will be also be pleased to mention anyone who spots a bug in subsequent editions.

XI What about other reader comments?

I began my first 8051 project in 1986 and I have tried to write the book that I needed at this time. Only you can tell me if I have succeeded.

I would appreciate your comments and feedback. For example, should the book be longer? Shorter? What other areas should I cover? What should I miss out? Would you like to see a future edition focusing on a different family of microcontrollers? If so, which one?

To ensure that any future editions continue to provide the information you need, I would be delighted to hear of your experiences (good or bad) using the book. I can be contacted either by post (via the publishers, please), or much more efficiently by e-mail at the address given at the end of this Preface.

I'll do my best to respond personally and promptly to every communication.

XII Credit where credit is due

The material presented here has evolved substantially in the three years since I began work on this project. The creation and subsequent development of this material would not have been possible without the help and support of a great many people.

In particular, I would like to thank:

- Kent Beck (Three Rivers Institute) for providing the Foreword and introducing me to Ray.
- The Engineering and Physical Sciences Research Council (EPSRC) and the (then) Science and Engineering Research Council (SERC), which have funded most of my research in this area.
- Staff at a range of UK and European organizations who have employed me as a consultant and / or attended my training courses in software development over the last decade and from whom I – in turn – have learned an enormous amount about embedded systems, software design and programming.
- Various people associated with the EuroPlop (1999) conference:
 - Fiona Kinnear (then at Addison-Wesley) for suggesting that I should attend.
 - My ‘shepherd’, Ward Cunningham, for making me revise my submission to take into account more of the ideas and philosophy of this book: as Ward predicted, the revised version provoked much useful debate.
 - All the people who took the time to comment on my draft patterns: of these people, Kent Beck deserves a particular mention as he provided numerous constructive comments and general support.
- The members of the Midlands Patterns Group for numerous helpful suggestions and ideas.
- Various people who have acted as reviewers during the evolution of this text:
 - Michael Jackson (University of Wolverhampton) for invaluable comments on my early ideas for the first version of this book.
 - Chris Hills (Keil Software), Niall Murphy (PanelSoft) and David Ward (The Motor Industry Research Association), who provided many useful comments on the first complete draft of this book.

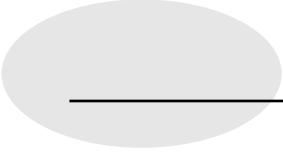
- Mark Banner (University of Leicester) for providing useful comments on several of the final draft chapters.
- Various people at the University of Leicester:
 - Members of the ‘Frankenstein’ group for inviting me to give my first talk on patterns and, through their enthusiasm and feedback, first convincing me that the ideas presented here had some validity.
 - Royan Ong, who has taught me a great deal about hardware design over the last two years.
 - Dave Dryden and Andy Willby for feedback on my hardware designs.
 - Andrew Norman, for creating the first version of the SPI library in Chapter 24 and – more generally – for finding numerous ‘features’ in my designs and code since 1992.
 - James Andrew, Adrian Banks, Mark Banner, Mathew Hubbard, Andrei Lesiapeto, Hitesh Mistry, Alastair Moir, Royan Ong, Chinmay Parikh, Keiron Skillett, Robert Smith, Thomas Sorrel, and Neil Whitworth, for destructive testing of many of the code examples.
 - The people who ‘saved my life’ when my computer went up in smoke in March 2000, when the first draft of this book was (over)due at the publishers, in particular Andy Willby and Jason Palmer.
 - Other members of staff for help and advice during the course of this project, including Declan Bates, Dave Dryden, Chris Edwards, Ian Jarvis, Fernando Schlindwein and Maureen Strange.
 - Ian Postlethwaite, for allowing me time to complete this large project.
- Bob Damper (University of Southampton) who introduced me to the challenges of speech recognition using the 8051 family in the mid-1980s.
- People at Keil Software:
 - Reinhard Keil, for his support and for providing an updated CD at the last minute.
 - Chris Hills, for much useful advice.
- The members of various e-mail pattern and microcontroller lists for numerous helpful comments and suggestions.
- Various people at Addison-Wesley Longman and Pearson Education:
 - Sally Mortimore (then of AWL) for letting me constantly change my mind about the contents of this book.
 - Alison Birtwell for stepping courageously into Sally’s shoes when Sally could take it no longer.
 - Katherin Ekstrom for answering all my e-mails.
 - Penelope Allport, for smooth management of the final production process.
 - Helen Baxter, for careful copy editing.
 - George Moore, for proof reading the final, vast, document.
 - Isobel McLean, for the index.
 - Everyone at Pantek, for the typesetting.

- Gordon Pont and Andrew Pont for proof reading.
- Last, but not least:
 - Sarah, for supporting me throughout the last three years.
 - Fiona, Mark, Siobhan and Clare, for teaching me how to fly kites.
 - Anna, Nick and Ella, for numerous Friday nights.
 - Lisa and Mike, for Tuscany.
 - Cass and Kynall Washington, for always being there.
 - Radiohead, for keeping me sane.

Michael J. Pont

Great Dalby, May 2001

M.Pont@leicester.ac.uk



Introduction

The chapters in this introductory section are intended to answer the following questions:

- What is an embedded system?
- What is a time-triggered system and what are the alternatives?
- Why are time-triggered systems generally considered to be more reliable than systems based on different architectures?
- What is a software pattern?
- How can patterns assist in the creation of reliable embedded applications?

1

chapter

What is a time-triggered embedded system?

In this introductory chapter, we consider what is meant by the phrases ‘embedded system’ and ‘time-triggered system’ and we examine how these important areas overlap.

1.1 Introduction

Current software applications are often given one of a bewildering range of labels:

- Information system
- Desktop application
- Real-time system
- Embedded system
- Event-triggered system
- Time-triggered system

There is considerable overlap between the various areas. We will therefore briefly consider all six types of application in this chapter, to put our discussions of time-triggered embedded systems in the remainder of this book in context.

1.2 Information systems

Information systems (ISs), and particularly ‘business information systems’, represent a huge number of applications. Although many of the challenges of information system development are rather different from those we will be concerned with in this book, a

basic understanding of such systems is useful, not least because most of the existing techniques for real-time and embedded development have been adapted from those originally developed to support the IS field.

As an example of a basic information system, consider the payroll application illustrated schematically in Figure 1.1.

This application will, we assume, be used to print the pay slips for a company, using employee data provided by the user and stored in the system. The printing of the cheques might take several hours: if a particularly complex set of calculations are required at the end of a tax year, and the printing is consequently delayed by a few minutes, then this is likely to be, at most, inconvenient. We will contrast this ‘inconvenience’ with the potentially devastating impact of delays in a real-time application in later examples.

ISs are widely associated with storage and manipulation of large amounts of data stored in disk files. Implementations in file-friendly languages, such as COBOL, were common in the 1960s and 1970s and such systems remain in widespread use, although most such systems are now in a ‘maintenance’ phase and new implementations in such languages are rare.

Modern IS implementations make far greater use of relational databases, accessed and manipulated using the SQL language. Relational database technology is well proven, safe and built on a formal mathematical foundation. While the design and implementation of large, reliable, relational database systems is by no means a trivial activity, the range of skills required to develop applications for use in a home or small business is limited. As a consequence, the implementation of such small relational

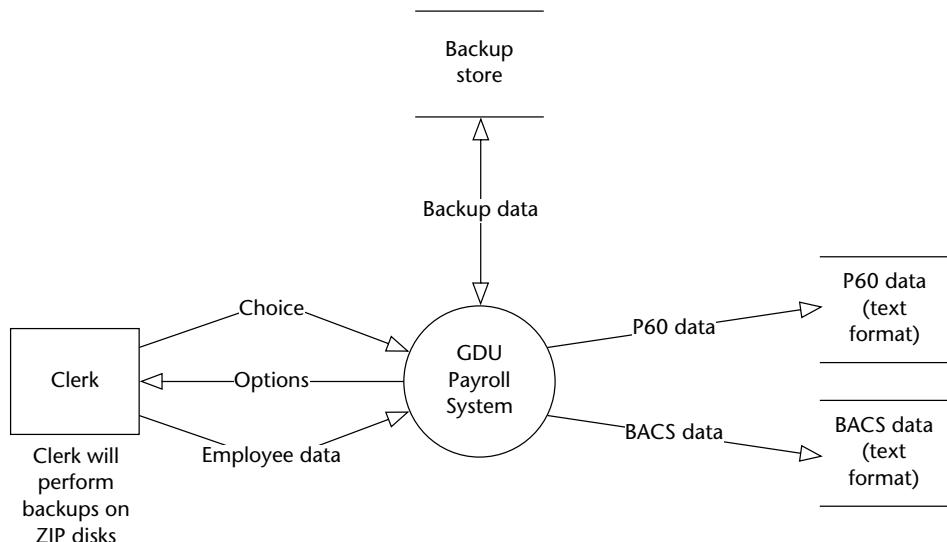


FIGURE 1.1 A high-level schematic view (dataflow diagram) of a simple payroll system.
Refer to Appendix A for details of this notation

database systems has ceased to be a specialized process and relational database design tools are now available to, and used by, many desktop computer users as part of standard 'office' packages.

However, new demands are being placed on the designers of information systems. Many hospitals, for example, wish to store waveforms (for example, ECGs or auditory evoked responses) or images (for example, X-rays or magnetic resonance images) and other complex data from medical tests, alongside conventional text records. An example of an ECG trace is shown in Figure 1.2.

For the storage of waveforms, images or speech relational databases systems, optimized for handling a limited range of data types (such as strings, characters, integers and real numbers), are not ideal. This has increased interest in object-oriented database systems ('object databases'), which are generally considered to be more flexible.

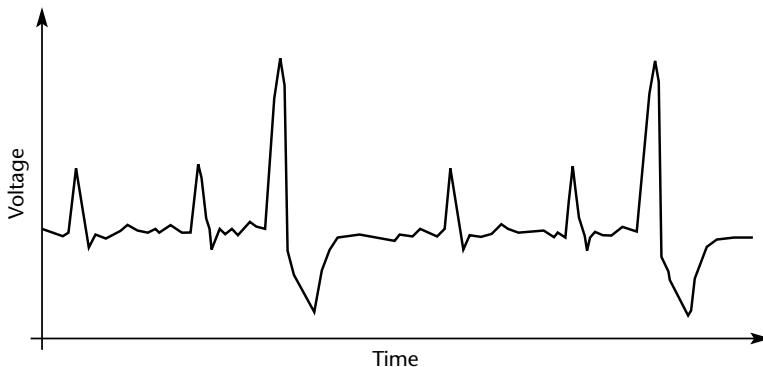


FIGURE 1.2 An example of an electrocardiogram (ECG) signal

1.3 Desktop systems

The desktop / workstation environment plays host to many information systems, as well as general-purpose desktop applications, such as word processors. A common characteristic of modern desktop environments is that the user interacts with the application through a high-resolution graphics screen, plus a keyboard and a mouse (Figure 1.3).

In addition to this sophisticated user interface, the key distinguishing characteristics of the desktop system is the associated operating system, which may range from DOS through to a version of Windows or the UNIX operating system.

As we will see, the developer of embedded applications rarely has an operating system, screen, keyboard or mouse available.

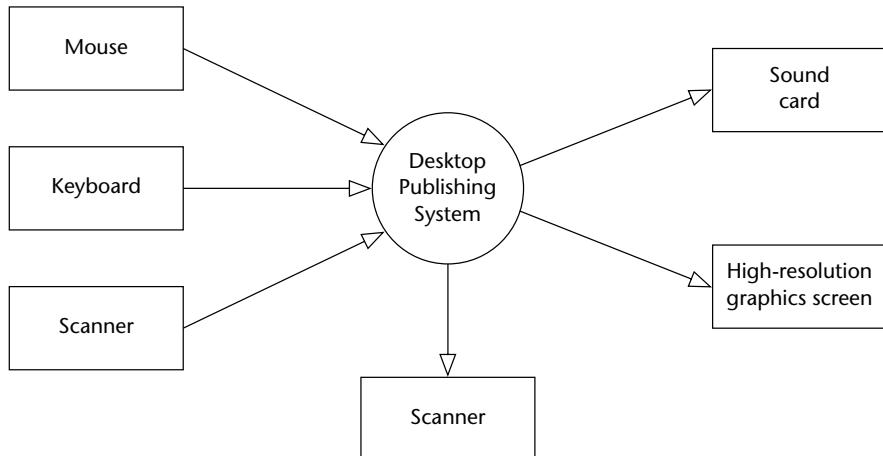


FIGURE 1.3 An example of part of the design for a GUI-driven desktop application

Reminder

1 second (s)	= 1.0 second (10^0 seconds) = 1000 ms.
1 millisecond (ms)	= 0.001 seconds (10^{-3} seconds) = 1000 μ s.
1 microsecond (μ s)	= 0.000001 seconds (10^{-6} seconds) = 1000 ns.
1 nanosecond (ns)	= 0.000000001 seconds (10^{-9} seconds).

1.4 Real-time systems

Users of most software systems like to have their applications respond quickly: the difference is that in most information systems and general desktop applications, a rapid response is a *useful* feature, while in many real-time systems it is an *essential* feature.

Consider, for example, the greatly simplified aircraft autopilot application illustrated schematically in Figure 1.4.

Here, we assume that the pilot has entered the required course heading and that the system must make regular and frequent changes to the rudder, elevator, aileron and engine settings (for example) in order to keep the aircraft following this path.

An important characteristic of this system is the need to process inputs and generate outputs very rapidly, on a time scale measured in milliseconds. In this case, even a slight delay in making changes to the rudder setting (for example) may cause the plane to oscillate very unpleasantly or, in extreme circumstances, even to crash. As a consequence of the need for rapid processing, few software engineers would argue with a claim that the autopilot system is representative of a broad class of real-time systems.

In order to be able to justify the use of the aircraft system in practice, it is not

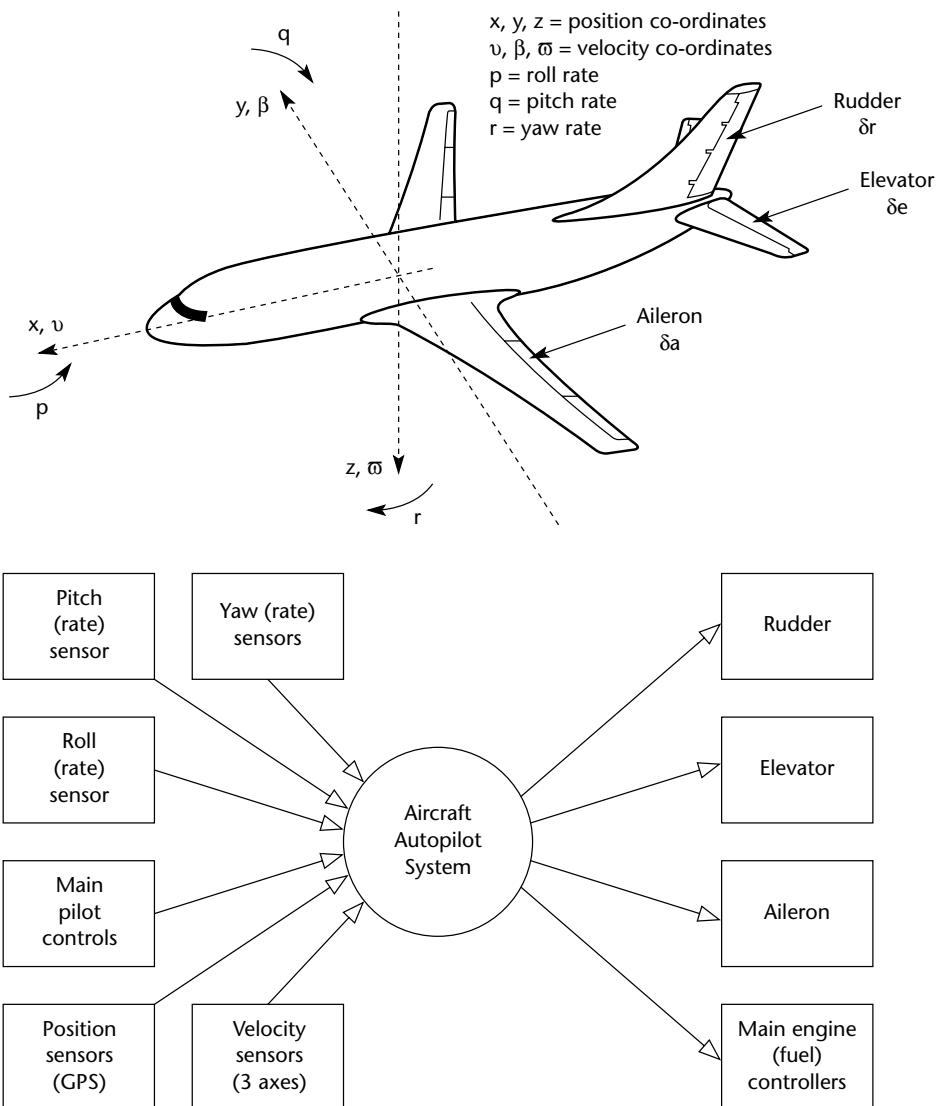


FIGURE 1.4 A high-level schematic view of a simple autopilot system

enough simply to ensure that the processing is ‘as fast as we can make it’: in this situation, as in many other real-time applications, the key characteristic is *deterministic* processing. What this means is that in many real-time systems we need to be able to *guarantee* that a particular activity will always be completed within (say) 2 ms, or at precisely 6 ms intervals: if the processing does not match this specification, then the application is not simply slower than we would like, it is *useless*.

Tom De Marco has provided a graphic description of this form of hard real-time requirement in practice, quoting the words of a manager on a software project:

'We build systems that reside in a small telemetry computer, equipped with all kinds of sensors to measure electromagnetic fields and changes in temperature, sound and physical disturbance. We analyze these signals and transmit the results back to a remote computer over a wide-band channel. Our computer is at one end of a one-meter long bar and at the other end is a nuclear device. We drop them together down a big hole in the ground and when the device detonates, our computer collects data on the leading edge of the blast. The first two-and-a-quarter milliseconds after detonation are the most interesting. Of course, long before millisecond three, things have gone down hill badly for our little computer. We think of *that* as a real-time constraint.'

(De Marco, writing in the foreword to Hatley and Pirbhai, 1987)

In this case, it is clear that this real-time system must complete its recording on time: it has no opportunity for a 'second try'. This is an extreme example of what is sometimes referred to as a 'hard' real-time system.

Note that, unlike this military example, many applications (like the aircraft system outlined earlier), involve repeated sampling of data from the real world (via a transducer and analog-to-digital converter) and, after some (digital) processing, creating an appropriate analog output signal (via a digital-to-analog converter and an actuator). Assuming that we sample the inputs at 1000 Hz then, to qualify as a real-time system, we must be able to process this input and generate the corresponding output, before we are due to take the next sample (0.001 seconds later).

To summarize, consider the following 'dictionary' definition of a real-time system:

'[A] program that responds to events in the world as they happen. For example, an automatic-pilot program in an aircraft must respond instantly in order to correct deviations from its course. Process control, robotics, games, and many military applications are examples of real-time systems.'

(Hutchinson New Century Encyclopedia (CD ROM edition, 1996))

It is important to emphasize that a *desire* for rapid processing, either on the part of the designer or on the part of the client for whom the system is being developed, is not enough, on its own, to justify the description 'real time'. This is often misunderstood, even by developers within the software industry. For example, Waites and Knott have stated:

'Some business information systems also require real-time control ... Typical examples include airline booking and some stock control systems where rapid turnover is the norm.'

(Waites and Knott, 1996, p.194)

In fact, neither of these systems can sensibly be described as a real-time application.

1.5 Embedded systems

Although it is widely associated with real-time applications, the category 'embedded systems', like that of desktop systems includes, for example, both real-time and, less

commonly, information systems. The distinguishing characteristic of an embedded application is loosely summarized in the box:

An embedded system is an application that contains at least one programmable computer (typically in the form of a microcontroller, a microprocessor or digital signal processor chip) and which is used by individuals who are, in the main, unaware that the system is computer-based.

Typical examples of such embedded applications include common domestic appliances, such as video recorders, microwave ovens and fridges. Other examples range from cars through combine harvesters to aircraft and numerous defence systems. Please note that this definition *excludes* applications such as 'palm computers' which – from a developer's perspective – are best viewed as a cut-down version of a desktop computer system.

While the desktop market is driven by a need to provide ever more performance, in order to support sophisticated operating systems and applications, the embedded market has rather different needs. For example, recent economic, legislative and technological developments in the automotive sector mean that an increasing number of road vehicles contain embedded systems. In some cases, such systems have been introduced primarily as a means of reducing production costs: for example, in modern vehicles, expensive (~£600.00) multi-wire looms have now been replaced by a two-wire controller area network (CAN) computer bus at a fraction of the cost (Perier and Coen, 1998). In other situations, such as the introduction of active suspension systems, the embedded systems have been introduced to improve ride quality and handling (Sharp, 1998).

Consider a very simple example which may help to illustrate some of the requirements of the embedded market: the indicator light circuit for a passenger car. In this application we need to be able to control six or more indicator lights from a switch behind the steering wheel, allowing the driver to tell other road users that he or she intends to turn a corner, change lane or park. For the US (and some other) markets, we expect the indicator circuit to interact with the rear lights (so that one light flashes to indicate the direction of a turn); in Europe, we expect indicator and rear lights to operate separately. Furthermore, in some countries, we wish to use the indicator lights as 'parking lights', to avoid having people run into our (parked) car at night.

The Volvo 131 ('Amazon') demonstrates the traditional solution to this problem. This classic European car from the 1960s uses a considerable amount of wire and some mechanical switches to provide indicator and braking behaviour: if we wanted to adjust this car to operate in the US style, we would have to make substantial changes to the wiring loom. To avoid this type of expensive, labour-intensive conversion, more modern cars use a microcontroller to provide the required behaviour. Not only does the microcontroller solution result in a simpler, and cheaper, collection of wires, it can also be converted between US and European indicator standards by flicking a switch or changing a memory chip.

This simple application highlights four important features of many embedded applications.

- Like this indicator system, many applications employ microcontrollers not because the processing is complex, but because the microcontroller is flexible and, crucially, because it results in a cost-effective solution. As a result, many products have embedded microcontrollers sitting almost idle for much of their operational life. The fact that many commonly used microcontrollers are, by comparison with modern desktop microprocessors, often rather slow, is seldom of great concern.
- Unlike most microprocessors, microcontrollers are required to interact with the outside world, not via keyboards and graphical user interfaces, but via switches, small keypads, LEDs and so on. The provision of extensive I/O facilities is a key driving force for many microcontroller manufacturers.
- Like the indicator system, most embedded applications are required to execute particular tasks at precise time intervals or at particular instants of time. In this case, for example, the indicators lights must flash on at a precise frequency and duty cycle in order to satisfy legal requirements. This type of application is considered in greater detail in Chapter 2.
- Unlike many desktop applications (for example) many embedded applications have safety implications. For example, if the indicator lights fail while the car is in use, this could result in an accident. As a result, reliability is a crucial requirement in many embedded applications.

1.6 Event-triggered systems

Many applications are now described as ‘event triggered’ or ‘event driven’. For example, in the case of modern desktop applications, the various running applications must respond to events such as mouse clicks or mouse movements. A key expectation of users is that such events will invoke an ‘immediate’ response.

In embedded systems, event-triggered behaviour is often achieved through the use of interrupts (see following box). To support these, event-triggered system architectures often provide multiple interrupt service routines.

What is an interrupt?

From a low-level perspective, an interrupt is a hardware mechanism used to notify a processor that an ‘event’ has taken place: such events may be ‘internal’ events (such as the overflow of a timer) or ‘external’ events (such as the arrival of a character through a serial interface).

Viewed from a high-level perspective, interrupts provide a mechanism for creating multitasking applications: that is applications which, apparently, perform more than one

task at a time using a single processor. To illustrate this, a schematic representation of interrupt handling in an embedded system is given in Figure 1.5.

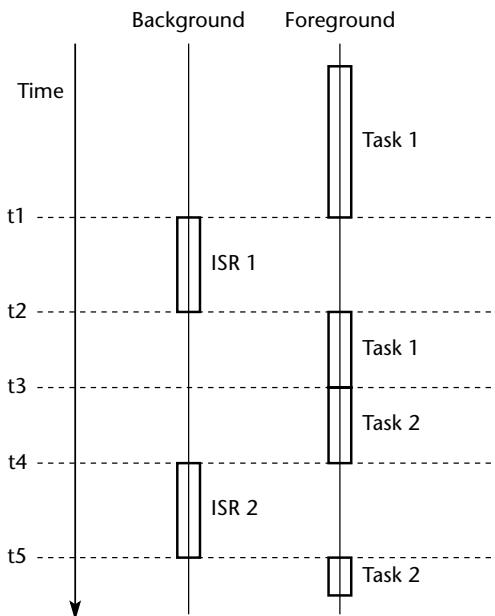


FIGURE 1.5 A schematic representation of interrupt handling in an embedded system

In Figure 1.5 the system executes two (background) tasks, Task 1 and Task 2. During the execution of Task 1, an interrupt is raised, and an ‘interrupt service routine’ (ISR1) deals with this event. During the execution of Task 2, another interrupt is raised, this time dealt with by ISR2.

Note that, from the perspective of the programmer, an ISR is simply a function that is ‘called by the microcontroller’, as a result of a particular hardware event.

1.7 Time-triggered systems

The main alternative to event-triggered systems architectures are time-triggered architectures (see, for example, Kopetz, 1997). As with event-triggered architectures, time-triggered approaches are used in both desktop systems and in embedded systems.

To understand the difference between the two approaches, consider that a hospital doctor must look after the needs of ten seriously ill patients overnight, with the support of some nursing staff. The doctor might consider two ways of performing this task:

- The doctor might arrange for one of the nursing staff to waken her, if there is a significant problem with one of the patients. This is the ‘event-triggered’ solution.

- The doctor might set her alarm clock to ring every hour. When the alarm goes off, she will get up and visit each of the patients, in turn, to check that they are well and, if necessary, prescribe treatment. This is the ‘time-triggered’ solution.

For most doctors, the event-triggered approach will seem the more attractive, because they are likely to get a few hours of sleep during the course of the night. By contrast, with the time-triggered approach, the doctor will inevitably suffer sleep deprivation.

However, in the case of many embedded systems – which do not need sleep – the time-triggered approach has many advantages. Indeed, within industrial sectors where safety is an obvious concern, such as the aerospace industry and, increasingly, the automotive industry, time-triggered techniques are widely used because it is accepted, both by the system developers and their certification authorities, that they help improve reliability and safety (see, for example, Allworth, 1981; MISRA, 1994; Storey, 1996; Nissanke, 1997; Bates, 2000 for discussion of these issues).

The main reason that time-triggered approaches are preferred in safety-related applications is that they result in systems which have very *predictable* behaviour. If we revisit the hospital analogy, we can begin to see why this is so.

Suppose, for example, that our ‘event-triggered’ doctor is sleeping peacefully. An apparently minor problem develops with one of the patients and the nursing staff decide not to awaken the doctor but to deal with the problem themselves. After another two hours, when four patients have ‘minor’ problems, the nurses decide that they will have to wake the doctor after all. As soon as the doctor sees the patients, she recognizes that two of them have a severe complications, and she has to begin surgery. Before she can complete the surgery on the first patient, the second patient is very close to death.

Consider the same example with the ‘time-triggered’ doctor. In this case, because the patient visits take place at hourly intervals, the doctor sees each patient before serious complications arise and arranges appropriate treatment. Another way of viewing this is that the workload is spread out evenly throughout the night. As a result, all the patients survive the night without difficulty.

In embedded applications, the (rather macabre) hospital situation is mirrored in the event-driven application by the occurrence of several events (that is, several interrupts) at the same time. This might indicate, for example, that two different faults had been detected simultaneously in an aircraft or simply that two switches had been pressed at the same time on a keypad.

To see why the simultaneous occurrence of two interrupts causes a problem, consider what happens in the 8051 architecture in these circumstances. Like many microcontrollers, the original 8051 architecture supports two different interrupt priority levels: low and high. If two interrupts (we will call them Interrupt 1 and Interrupt 2) occur in rapid succession, the system will behave as follows:

- If Interrupt 1 is a low-priority interrupt and Interrupt 2 is a high-priority interrupt:

The interrupt service routine (ISR) invoked by a low-priority interrupt can be interrupted by a high-priority interrupt. In this case, the low-priority ISR will be paused, to allow the high-priority ISR to be executed, after which the operation of the low-priority ISR will be

completed. In most cases, the system will operate correctly (provided that the two ISRs do not interfere with one another).

- If Interrupt 1 is a low-priority interrupt and Interrupt 2 is also a low-priority interrupt:

The ISR invoked by a low-priority interrupt cannot be interrupted by another low-priority interrupt. As a result, the response to the second interrupt will be at the very least delayed; under some circumstances it will be ignored altogether.

- If Interrupt 1 is a high-priority interrupt and Interrupt 2 is a low-priority interrupt:

The interrupt service routine (ISR) invoked by a high-priority interrupt cannot be interrupted by a low-priority interrupt. As a result, the response to the second interrupt will be at the very least delayed; under some circumstances it will be ignored altogether.

- If Interrupt 1 is a high-priority interrupt and Interrupt 2 is also a high-priority interrupt:

The interrupt service routine (ISR) invoked by a high-priority interrupt cannot be interrupted by another high-priority interrupt. As a result, the response to the second interrupt will be at the very least delayed; under some circumstances it will be ignored altogether.

Note carefully what this means! There is a common misconception among the developers of embedded applications that interrupt events will never be lost. This simply is not true. If you have multiple sources of interrupts that may appear at 'random' time intervals, interrupt responses can be missed: indeed, where there are several active interrupt sources, it is practically impossible to create code that will deal correctly with all possible combinations of interrupts.

It is the need to deal with the simultaneous occurrence of more than one event that both adds to the system complexity and reduces the ability to predict the behaviour of an event-triggered system under all circumstances. By contrast, in a time-triggered embedded application, the designer is able to ensure that only single events must be handled at a time, in a carefully controlled sequence.

As already mentioned, the predictable nature of time-triggered applications makes this approach the usual choice in safety-related applications, where reliability is a crucial design requirement. However, the need for reliability is not restricted to systems such as fly-by-wire aircraft and drive-by-wire passenger cars: even at the lowest level, an alarm clock that fails to sound on time or a video recorder that operates intermittently, or a data monitoring system that – once a year – loses a few bytes of data may not have safety implications but, equally, will not have high sales figures.

In addition to increasing reliability, the use of time-triggered techniques can help to reduce both CPU loads and memory usage: as a result, as we demonstrate throughout this book, even the smallest of embedded applications can benefit from the use of this form of system architecture.

1.8 Conclusions

The various characteristics of time-triggered embedded systems introduced in this chapter will be explored in greater depth throughout this book.

In the next chapter, we consider why ‘traditional’ software design techniques provide only limited support for the developers of this type of application and argue that software patterns can provide a useful adjunct to existing approaches.

chapter 2

Designing embedded systems using patterns

In this second introductory chapter, we consider why ‘traditional’ software design techniques provide only limited support for the developers of embedded applications and argue that software patterns can provide a useful adjunct to such techniques.

2.1 Introduction

Most branches of engineering have a long history. Work in the area of control systems, for example, might be said to have begun with the seminal studies by James Watt on the flywheel governor in the 1760s, while work in electrical engineering can be dated back to the work of Michael Faraday, who is generally credited with the invention of the electric motor in 1821. It can be argued that the practice of civil engineering has the longest history of all, originating, perhaps, with the building of the Egyptian pyramids, or to Greek or Roman times: certainly the Institution of Civil Engineers was founded in England (UK) in 1818 and is the oldest professional engineering institution in the world.

For the software engineer, a different situation applies. The first mass-produced minicomputer, the PDP-8, was launched only in 1965 and the first microprocessor only in 1971. As a result of the comparatively late introduction, and subsequent rapid evolution, of small programmable computer systems, the field of software engineering has had little opportunity to mature. In the limited time available, much work on software engineering has focused on the design process and, in particular, on the development and use of various graphical notations, including process-oriented notations, such as dataflow diagrams (Yourdon, 1989: see Figure 2.1), and object-oriented notations, such as the ‘Unified Modelling Language’ (Fowler and Scott, 2000). The use of such notations is supported by ‘methodologies’: these are collections of ‘recipes’ for

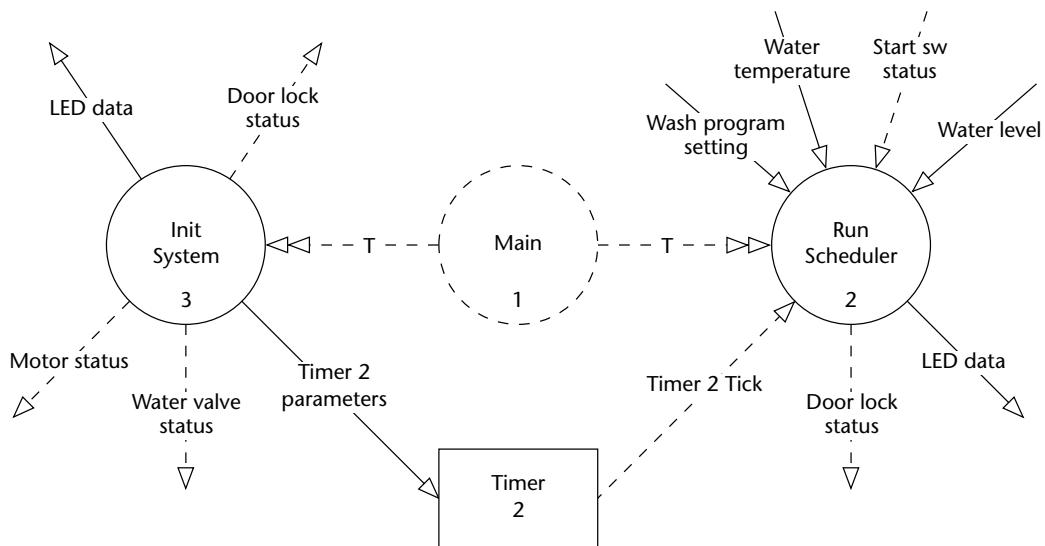
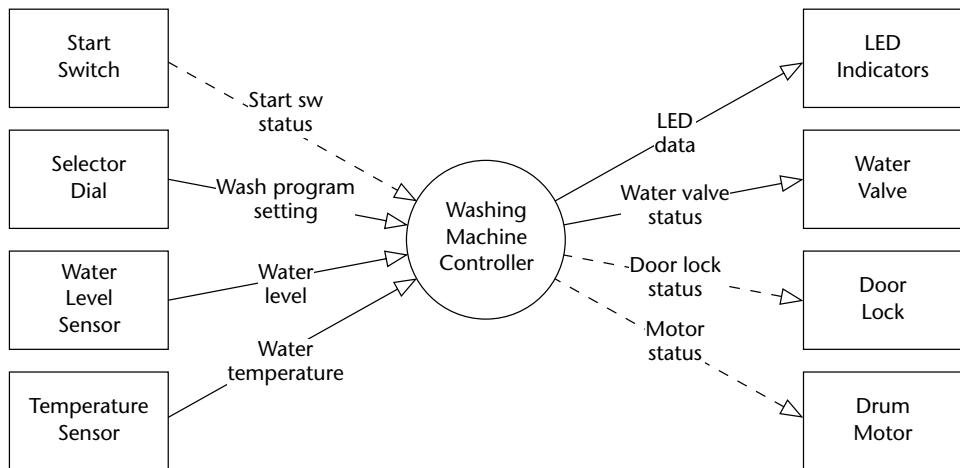


FIGURE 2.1 An example of part of the design for a washing machine, showing (top) the context diagram and (bottom) the Level 1 dataflow diagram

[Note: that in this example, and throughout most of this book, we have used a process-oriented (dataflow) notation¹ to record the design solutions: this remains the most popular approach for embedded applications, in part because process-oriented languages (notably 'C') are popular in this area. Although object-oriented languages (like C++) are comparatively uncommon in microcontroller-based embedded projects at the present time, object-oriented design notations could equally well be used to record the design presented here.]

1. Please refer to Appendix A for details of this notation.

software design, detailing how and when particular notations should be used in a project (see, for example, Pont, 1996). The designs that result from the application of these techniques consist of a set of linked diagrams, each following a standard notation, and accompanied by appropriate supporting documentation (Yourdon, 1989; Booch, 1994; Pont, 1996; Fowler and Scott, 2000).

As the title suggests, we are concerned in this book with the development of software for embedded systems. In the past, despite the ubiquitous nature of embedded applications, the design of such systems has not been a major focus of attention within the software field. Indeed, in almost all cases, software design techniques have been developed first to meet the needs of the developers of desktop business systems (DeMarco, 1978; Rumbaugh *et al.*, 1991; Coleman *et al.*, 1994), and then subsequently 'adapted' in an attempt to meet the needs of developers of real-time and / or embedded applications (Hatley and Pirbhai, 1987; Selic *et al.*, 1994; Awad *et al.*, 1996; Douglass, 1998). We will argue (in Section 2.2) that the resulting software design techniques, although not without merit, cannot presently meet the needs of the designers of embedded systems. We then go on to propose (Sections 2.2 and 2.4) that the use of software patterns as an adjunct to existing techniques represents a promising way to alleviate some of the current problems.

2.2 Limitations of existing software design techniques

We begin the main part of this chapter by considering two examples which illustrate the limitations of standard design techniques when used for embedded system development.

Cruise-control system

As a first example, we will consider a cruise-control system (CCS) for a road vehicle. A CCS is often used to demonstrate the effectiveness of real-time software design methodologies (for example, see Hatley and Pirbhai, 1987; Awad *et al.*, 1996). Such a system is usually assumed to be required to take over the task of maintaining the vehicle at a constant speed even while negotiating a varying terrain, involving, for example, hills or corners in the road. Subject to certain conditions (typically that the vehicle is in top gear and exceeding a preset minimum speed), the cruise control is further assumed to be engaged by the driver via a push switch on the dashboard and disengaged by touching the brake pedal.

As an example, an outline process-oriented ('structured') design for such a system is illustrated in Figure 2.2 to Figure 2.5. This design is adapted from that suggested by Hatley and Pirbhai (1987) in a standard text on real-time software design. Starting at the highest level of abstraction, Figure 2.2 shows the 'context diagram' for the system. Figure 2.3 shows the corresponding Level 1 dataflow diagram, which describes in more detail the processing carried by the process 'simple cruise control' in Figure 2.2. Figure 2.4 in turn shows the state-transition diagram associated with 'main control process' in Figure 2.3.

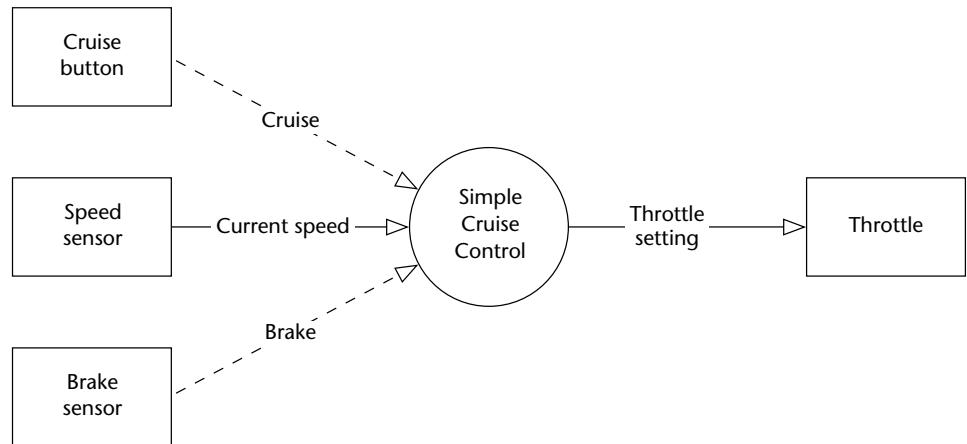


FIGURE 2.2 The context diagram for the simple cruise-control system

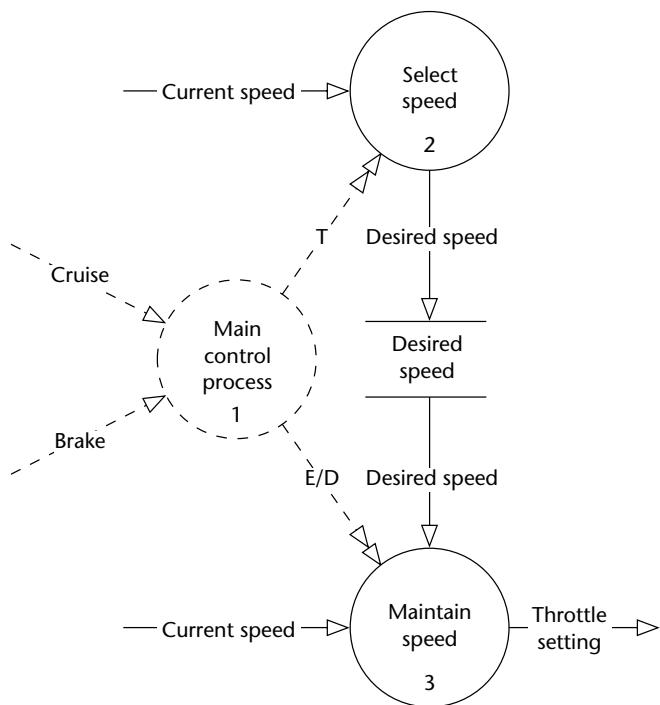


FIGURE 2.3 The Level 1 dataflow diagram for the cruise-control system

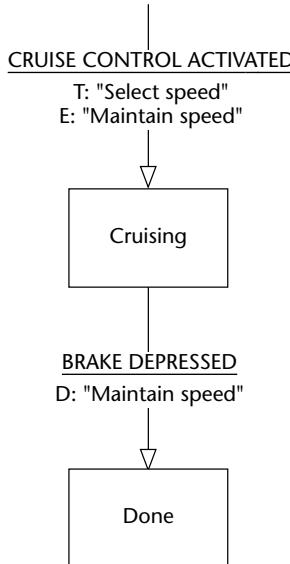


FIGURE 2.4 The state-transition diagram associated with the control process in Figure 2.2

To complete this simple design, we need to define more precisely the operation of the two processes 'set speed' and 'maintain speed'. Of these, the second is the more complex and will be discussed here. Hatley and Pirbhai (1987) present the 'process specification' given in Figure 2.5 for their version of the 'maintain speed' process.

Overall, there are a number of problems with this design solution. For example, little consideration is given to the system architecture to be employed, and the consequences this will have for the rest of the design. In addition, Hatley and Pirbhai (1987) present their version of the control algorithm at the heart of the system (see Figure 2.5) largely

Set: $V_{Th} = \begin{cases} 0; & (S_D - S_A) > 2 \\ 2(S_D - S_A + 2); & -2 \leq (S_D - S_A) \leq 2 \\ 8; & -2 > (S_D - S_A) \end{cases}$	Subject to: $\frac{dV_{Th}}{dt} \leq 0.8V / \text{sec}$
--	--

FIGURE 2.5 A possible process specification ('PSpec') for the 'Maintain Speed' process (adapted from Hatley and Pirbhai, 1987, p.291)

[Note: V_{Th} = Throttle setting; S_D = Desired speed; S_A = Actual speed. The throttle setting is assumed (here) to be proportional to an output voltage: a 0V output closes the throttle, and an 8V output sets it fully open. The intention is to vary the throttle setting when the actual speed varies by more than 2mph above or below the desired speed. The rate of throttle movement is restricted to 0.8V / second.]

without comment. They do not consider ‘standard’ control techniques, such as ‘proportional-integral-differential’ (PID) control, as solutions to this problem,² and give no indication how the effectiveness of the chosen approach should be assessed (or even that it should be assessed). This is not particularly unusual in software texts: for example, some ten years later, Awad *et al.* (1996) describe, in considerably more detail, an object-oriented design for the same cruise control system and, again, largely ignore these issues.

To design many embedded systems successfully, including this cruise control system, requires not just knowledge of software engineering and microcontroller hardware, but also contributions from related technical and engineering fields, often including instrumentation, digital signal processing, artificial intelligence and control theory. In our experience, such contributions are often essential to the success of real-time software projects, yet they are not part of the formal training of the majority of software engineers and are frequently ignored in books and papers on real-time software design.

Alarm clock

Of course, a CCS is a specialized product and it might reasonably be argued that most developers would not expect to tackle the production of such an application without having previous experience in the automotive area. However, similar problems arise if in the simplest of embedded applications.

Assume, for example, that you have been asked to develop an alarm clock application that operates as follows:

- Time is displayed on an LED display.
- The time may be adjusted by the user.
- An (optional) alarm will sound at a time determined by the user.

A sketch of the required user interface is given in Figure 2.6.

To create the design for such an application in a ‘traditional’ manner, we might begin by drawing a context diagram (Figure 2.7).

As with the CCS, the notation can only assist us in recording design decisions: it cannot assist in making those decisions. For example, in this type of application, a basic requirement is to display information on the LED display. In most cases, to reduce costs, a multiplexed display will be used. As we discuss in Chapter 21, in a multiplexed, 4-digit, LED display, we need to refresh the display approximately every 5 ms. The need to update this display at this frequency will have implications for the software architecture of the whole application and must be taken into account very early in the design process. If the designer of the system is unaware of this basic requirement, then many of the assumptions that underlie the design will be wrong, and a large amount of expensive and time-consuming redesign will be required when the project reaches the implementation phase.

2. We discuss PID control algorithms in Chapter 35.

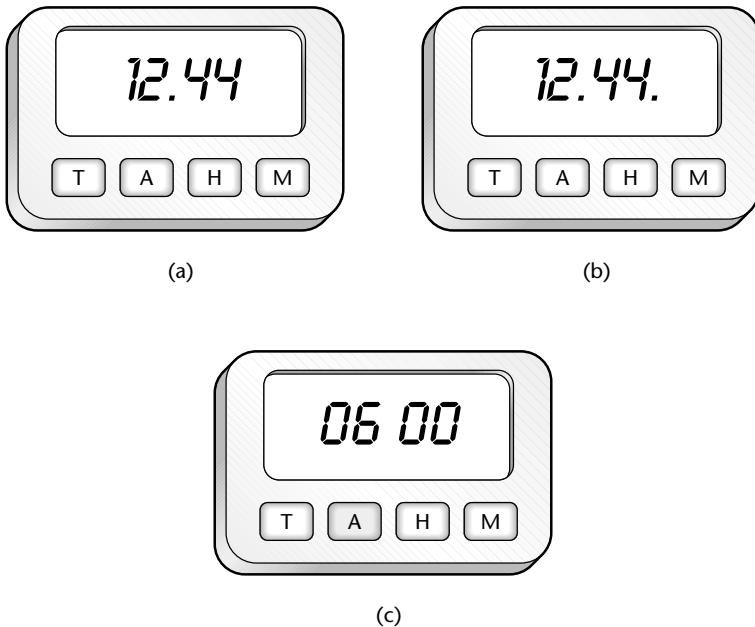


FIGURE 2.6 The required user interface. [a] The normal (current time) display when the alarm is not set. [b] The normal (current time) display when the alarm is set to ring. [c] The alarm time display when the A button is pressed

[Note: Pressing The A + H buttons will let the user change the alarm time (hours); the A + M buttons will similarly change the alarm time (minutes). The same procedure, using the T button will be used to change the displayed time. Pressing the A button when the alarm sounds will stop the alarm.]

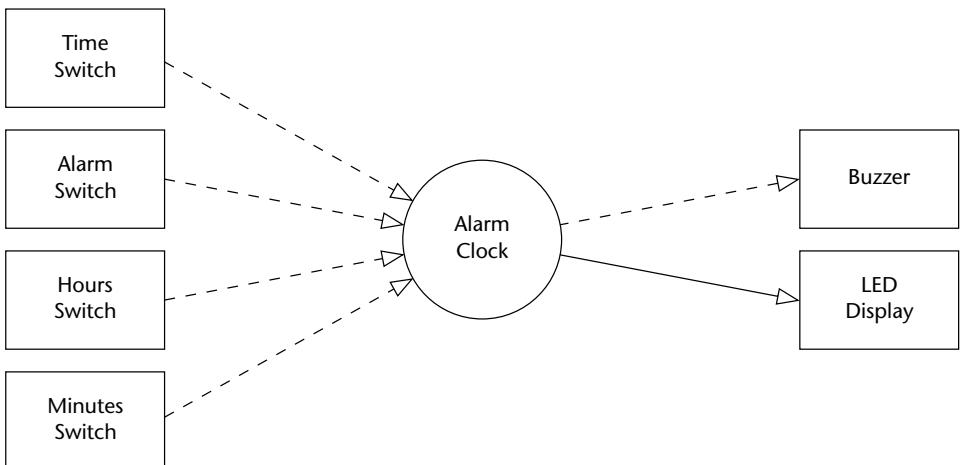


FIGURE 2.7 A possible context diagram for an alarm clock application

2.3 Patterns

We can sum up the conclusions from these two examples by saying that – for those developers with experience of control system design, or the use of LED displays – the tasks are straightforward: however, for those without such experience, even the smallest of decisions can have unexpected repercussions. Unfortunately, the standard design notations that have been developed do not provide any means of substituting for the lack of experience on the part of a particular designer. The consequence is not difficult to predict, and is summarized succinctly in this quotation from an experienced developer of embedded applications: ‘It’s ludicrous the way we software people reinvent the wheel with every project’ (Ganssle, 1992).

To address these problems use of ‘objects’, ‘agents’ or any new form of software building block or design notation will not greatly help. Instead, what is needed is a means of what we might call ‘recycling design experience’: specifically, we would like to find a way of incorporating techniques for reusing successful design solutions into the design process.

Recently, many developers have found that **software patterns** offer a way of achieving this. Current work on software patterns has been inspired by the work of Christopher Alexander and his colleagues (for example Alexander *et al.*, 1977; Alexander, 1979). Alexander is an architect who first described what he called ‘a pattern language’ relating various architectural problems (in buildings) to good design solutions. He defines patterns as ‘a three-part rule, which expresses a relation between a certain context, a problem, and a solution’ (Alexander, 1979, p.247).

For example, consider Alexander’s **WINDOW PLACE** pattern, summarized briefly in Figure 2.8. This takes the form of a recognizable problem, linked to a corresponding solution. More specifically, like all good patterns, **WINDOW PLACE** does the following:

- It describes, clearly and concisely, a successful solution to a significant and well-defined problem.
- It describes the circumstances in which it is appropriate to apply this solution.
- It provides a rationale for this solution.
- It describes the consequences of applying the solution.
- It gives the solution a name.

This basic concept of descriptive problem–solution mappings was adopted by Ward Cunningham and Kent Beck who used some of Alexander’s techniques as the basis for a small ‘pattern language’ intended to provide guidance to novice Smalltalk programmers (Cunningham and Beck, 1987). This work was subsequently built upon by Erich Gamma and colleagues who, in 1995, published an influential book on general-purpose object-oriented software patterns (Gamma *et al.*, 1995).

For example, consider the **OBSERVER** pattern (Gamma *et al.*, 1995), illustrated in Figure 2.9. This describes how to link the components in a multi-component application, so that when the state of one part of the system is altered, all other related parts are notified and, if necessary, updated. This pattern successfully solves the problem, while leaving the various system components loosely coupled, so that they may be more easily altered or reused in subsequent projects.

WINDOW PLACE**Context**

WINDOW PLACE is an architectural pattern. It is most frequently applied in the design of hotels, offices or substantial houses.

Problem

You need to design a ‘living room’ in which people will congregate to sit and talk, drink tea, read newspapers, and so forth.

Solution

In developing a solution to this problem, Alexander *et al.* made the following observations:

- During the day, people generally dislike spending time in rooms without windows.
- When a room has windows, then people will be drawn to them. As a result, if the seating area is adjacent to the windows (preferably so that people can see out from their seats), then most people will tend to feel comfortable.
- By contrast, if the windows are on one side of the room and the seats on the other, most people will tend to feel uncomfortable.

Based on these observations (presented in greater detail in the original than is possible here), Alexander *et al.* proposed that, in solving this problem, architects should aim to create a well-lit ‘window place’, where people can sit comfortably adjacent to the window.

FIGURE 2.8 A summary of the **WINDOW PLACE** architectural pattern (adapted from Alexander *et al.*, 1977)

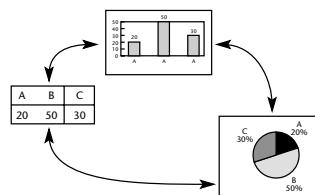
OBSERVER**Context**

OBSERVER is a software pattern, intended for use in applications with two or more communicating components. The description given by Gamma *et al.* focuses on the development of desktop applications, but the pattern may also be applied in certain (comparatively complex) embedded systems.

Problem

You need to implement a one-to-many dependency between components in an application so that when the state of one component is altered, all other related components are notified and, if necessary, updated.

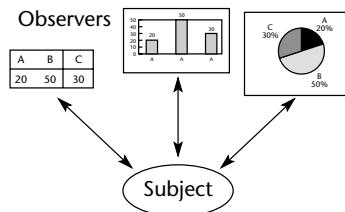
For example, suppose you are developing a spreadsheet application. In such a program, the same information may be displayed (for example) in a table, as a bar chart and as a pie chart. Changes to any of these displays need to be reflected in the others, illustrated as:



Solution

OBSERVER describes how to break down such applications into ‘subject’ and ‘observer’ components.

As described by Gamma *et al.*, a subject may have any number of observers, all of which are notified when the state of the subject changes: in response, observers will (usually) synchronize their state with the subject’s state, illustrated schematically as:

**Applicability and consequences**

One of the situations in which **OBSERVER** may be applied is when a change to one component requires changing others, and you do not know how many other components need to be altered.

One important consequence of using this pattern is that the various communicating components are loosely coupled together: this means, for example, that new components can be added, or existing components can be removed, with little impact on the rest of the program.

Related patterns and alternative solutions

The type of interaction described in **OBSERVER** is also referred to as a publish-subscribe relationship.

Gamma *et al.* describe two related patterns: **MEDIATOR** and **SINGLETON**. These are not considered further here.

Examples

Gamma *et al.* describe applications in which **OBSERVER** has been employed.

FIGURE 2.9 An overview of the pattern **OBSERVER** (adapted from Gamma *et al.*, 1995)

2.4 Patterns for time-triggered embedded systems

We found the software patterns described by Gamma *et al.* (1995) to be useful. However, they were insufficiently specialized for use with time-triggered embedded systems. We therefore began to assemble a collection of patterns based on our experience with the development of applications for the 8051 and other families of microcontrollers.

The first version of these patterns were used ‘in house’, primarily for teaching and training purposes. We then began to publish and discuss the next versions of the patterns more widely, not just at pattern workshops (see, for example, Pont *et al.*, 1999a; Pont, in press) but also at more general technical conferences (see, for example, Pont, 1998; Pont *et al.*, 1999b). Through this process we obtained a great deal of useful feed-

back on the project, and refined the collection again. The end result was the set of patterns described in detail throughout this book.

The structure of the final version of the patterns is illustrated in Figure 2.10. The patterns are grouped by chapter, and the book is further divided into sections. This arrangement is intended to make it easy for you to find the information you require.

2.5 Conclusions

In this second brief introductory chapter, we have argued that developers of embedded applications can benefit from pattern-driven design techniques not least because many embedded projects require the developer to have both knowledge of software design and from a range of related technical and engineering fields.

Four points should be made as we conclude this chapter:

- Software patterns should not be seen as an attempt to produce a panacea or what Brooks (1986) calls a ‘silver bullet’ for the problems of embedded software design or implementation. Software development is a multifaceted activity, requiring intelligence and creativity: the solutions to such problems come from intelligent and creative individuals, and there are no ‘miracle cures’ or ‘silver bullets’ waiting to be uncovered.
- For similar reasons, use of the patterns in this book does not guarantee that your system will be reliable. Thus, for example, the first time you get on a plane and the pilot announces, reassuringly, that the flight control software was designed by engineers who used the latest set of ‘time-triggered software patterns’, this may mean that the software is more reliable than it would have been if the designers and programmers had not been exposed to a set of patterns like those presented in this book. However, it can never mean that the flight software is fault free.
- At best, ‘the pattern solution’ will be a partial one. For example, it is not feasible to provide all software engineers or their managers, irrespective of background or training, with sufficient knowledge of relevant fields to ensure that they can, for example, create appropriate designs for aircraft flight control systems or fault diagnosis systems based on sliding-mode observers. However, what we may be able to achieve is to make software managers, and the teams they manage, better able to recognize projects in which it would be advisable to appoint (say) an artificial intelligence, signal processing or control expert from within the company on the project team or to employ an outside consultant to fulfil such a rôle.
- No useful pattern collection is ever ‘finished’: further patterns will gradually be added and existing patterns can always be improved. This collection is certainly no exception. As discussed in the preface, your comments and feedback on this collection would be very much appreciated.

<PATTERN NAME>

Context

'Context' summarizes situations in which you may find the pattern useful. In most of the patterns in this book, the overall context will be similar:

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

However, in most cases, the pattern will be more focused. For example:

- The application is to be powered by batteries.
- You are creating the user interface for your application.

Problem

'Problem' provides a brief summary of the problem which is addressed by the pattern. For example:

- When and how should you use a quartz crystal to create an oscillator with members of the 8051 family of microcontrollers?
- How do you create and use a co-operative scheduler?

Background

'Background' provides information that will help less experienced developers make full use of the pattern.

Solution

The solution section describes one or more solutions to the problem addressed by the pattern. This solution may include software designs, code listings and / or hardware schematics.

Hardware resource implications

Every pattern provides and / or consumes hardware resources. For example, the microcontroller patterns (Chapter 3) provide CPU and limited memory resources and the external memory patterns (Chapter 6) provide substantial additional memory resources. By contrast, most of the remaining patterns require both CPU and memory resources. Part of the design process involves balancing the need for, and provision of, hardware resources: 'Hardware resource implications' helps to achieve this.

Reliability and safety implications

Many patterns have potential reliability and safety implications: such issues are discussed in this section.

Portability

This section considers issues involved in porting the pattern to a different microcontroller.

Overall strengths and weaknesses

- 😊 This section summarises both the strengths
- 😢 ... and the weaknesses of the pattern.

Related patterns and alternative solutions

This pattern may not be precisely what you require. 'Related patterns and alternative solutions' discusses alternative solutions, and gives references to other, related patterns that may also be of interest.

Example

At least one example of the application of each pattern is given.

Further reading

'Further reading' gives suggestions for sources of additional information relevant to those using this pattern.

FIGURE 2.10 The structure of the patterns in this book

Hardware foundations

The patterns in Part A are concerned with the creation of the basic hardware foundation that is required in all microcontroller-based embedded applications.

We begin by considering the microcontroller itself. Often the initial choice of processor must be made early in the project life-cycle; this choice will have a substantial impact on many later software and hardware design decisions and development costs can be substantially reduced if subsequent changes can be avoided. In Chapter 3, a small set of patterns are presented to help you decide whether a member of the 8051 family is appropriate for your application and, if so, which one you should use.

We then turn our attention to oscillator circuits. All digital computer systems are driven by some form of oscillator. This circuit is the ‘heartbeat’ of the system and is crucial to correct operation. For example, if the oscillator fails, the system will not function at all; if the oscillator runs irregularly, any timing calculations performed by the system will be inaccurate. In Chapter 4, two key forms of oscillator circuit are discussed and compared.

We next consider the non-trivial process required to start the microcontroller when power is applied. Because of the system complexity, a small, manufacturer-defined ‘reset routine’ must be run to place this hardware into an appropriate state before it can begin executing the user program. Running this reset routine takes time and requires that the microcontroller’s oscillator is operating. In Chapter 5, we consider different ways of creating a suitable reset circuit.

Memory is the next important issue we need to consider. Specifically, in Chapter 6, we explore techniques for making effective use of the internal memory in the 8051 family and, where necessary, of adding external (code and / or data) memory to the system.

Finally, we consider how you can create the hardware needed to drive DC loads (in Chapter 7) and AC loads (in Chapter 8).

3

chapter

The 8051 microcontroller family

Introduction

Early in the life-cycle of most embedded projects, an initial choice of microcontroller must be made. While it may become necessary to change the microcontroller as the project develops, the particular hardware platform that is used will have a substantial impact on many later software and hardware design decisions and development costs can be substantially reduced if subsequent changes can be avoided.

In this chapter, three patterns are presented to support this selection process:

- **STANDARD 8051** [page 30]
- **SMALL 8051** [page 41]
- **EXTENDED 8051** [page 46]

Note that using any processor family requires a considerable investment in hardware, software and staff training: few firms can afford to select devices on a per-project basis: instead, they tend to specialize in a single family or a small number of families. As we will see, the huge – and growing – range of 8051 devices available can make it an excellent choice for a wide range of projects and, as a result, can frequently justify this investment.

STANDARD 8051

Context

You are developing a microcontroller-based embedded application and have some flexibility in the choice of hardware platform to be used.

Problem

Should you base your application on a standard 8051-family microcontroller?

Background

Taken as a whole, the 8051 family has what is, in the semiconductor world, a very long history. The underlying architecture is derived from that of the Intel 8048 microcontroller (introduced in 1976): the first 8051 was introduced in 1980 (Intel, 1985).

The original 8051 architecture had the following features:

- Up to 12 MHz operating frequency.
- Thirty-two digital input / output pins (arranged as four 8-bit ports).
- Internal data (RAM) memory – 128 bytes.
- Three versions with different program memory options:
 - No program memory: all programs needed to be stored in external memory (8031)
 - 4K × 8 bits internal mask-programmed ROM (8051)
 - 4K × 8 bits UV-erasable EPROM (8751)
- Two 16-bit timer / counters (Timer 0 and Timer 1).
- Five interrupt sources were provided (two external) with two priority levels.
- One programmable, full-duplex, serial port.

The external interface to the 8051 is illustrated in Figure 3.1.

Shortly after the launch of the 8051, the 8052 was launched (again by Intel). The 8052 differed in several important respects from the earlier device. The 8052 had the following features:

- Internal data (RAM) memory was increased to 256 bytes.
- Two 8052 versions were available with different program memory options:
 - No program memory: all programs needed to be stored in external memory (8032)
 - 8K × 8 bits internal mask-programmed ROM (8052)
- Three 16-bit timer / counters (Timer 0, Timer 1 and Timer 2).
- Six interrupt sources were provided (two external) with two priority levels.

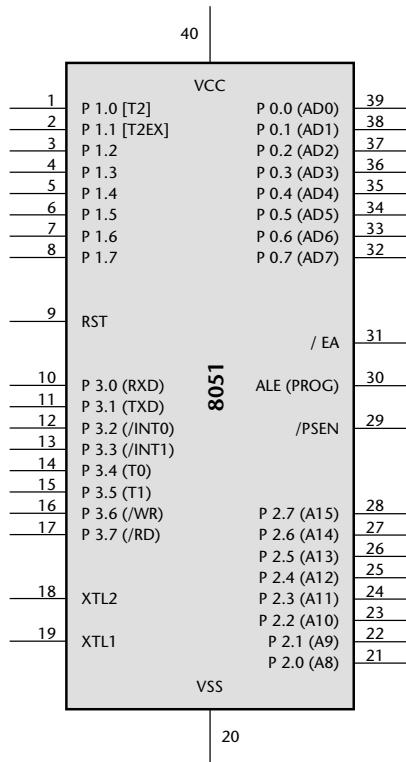


FIGURE 3.1 The external interface of the 8051 microcontroller (40-pin package)

[Note: that many of the digital I/O pins have alternative functions: for example, in applications involving a UART-based serial interface, Pins 3.0 and 3.1 are used (see Chapter 18). Note also that the alternative functions on pins 1.0 and 1.1 are only provided on 8052-based derivatives.]

The 8052 added useful features to the basic architecture, particularly the additional RAM and ‘Timer 2’. It was also ‘upwardly compatible’ with the 8051: that is, it was pin, and code compatible with the 8051. Because of this, in almost all cases, modern ‘standard’ 8051 devices are based on the 8052 family. Within this text, we will consider ‘Standard 8051’ devices to be those which are pin and code compatible with either the 8051 or (more commonly) the 8052 device.

The popular Atmel 89S53 is a representative example of a modern Standard 8051. Listed here is a summary of the main features of the AT89S53:

- Fully static operation: 0–24 MHz operating frequency.
- Thirty-two input / output lines (arranged as four 8-bit ports).
- Internal data (RAM) memory – 256 bytes.
- 12 Kbytes of ‘in circuit programmable’ ROM.

- Three 16-bit timers / counters (Timer 2 with up/down counter feature).
- Nine interrupts (two external) with two priority levels.
- Programmable watchdog timer (see Chapter 12).
- SPI interface (see Chapter 24).
- Low-power idle and power-down modes.
- 4V to 6V operating range.

Modern Standard 8051 devices like the AT89S53 are now generally packaged in 40-pin DIP, 44-pin PLCC or 44-pin MQFP cases. Examples of each type of package are shown in Figure 3.2.

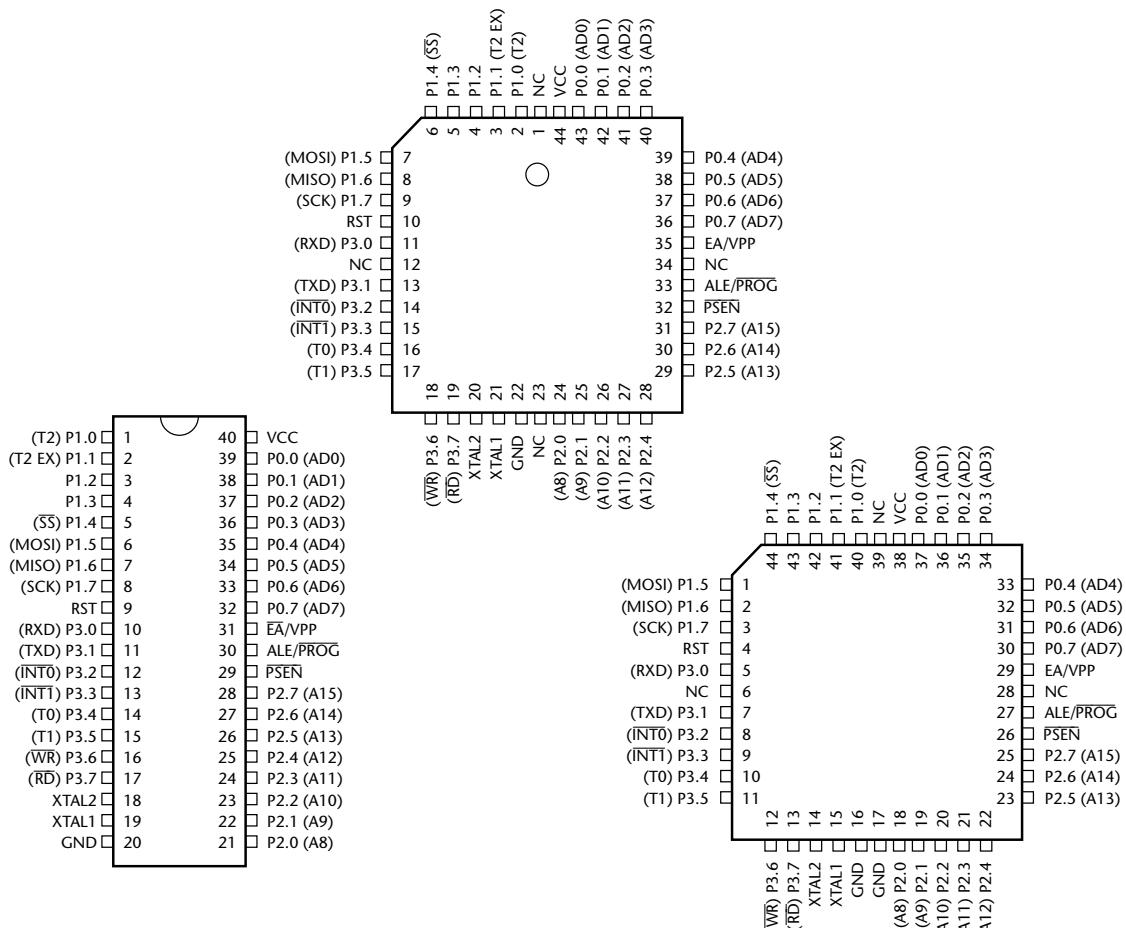


FIGURE 3.2 Examples of common packages used for the Standard 8051 (in this case, the Atmel AT89S53). [Left] The DIP package. [Middle] The PLCC package. [Right] The MQFP package (reproduced courtesy of Atmel Corporation)

What's in a name?

It should be noted that the naming of various members of the '8051' family is a source of considerable confusion to new developers. For example, the 8031, 8751, 8052, 8032, C505c, C515, C509, 80C517, 83C452, AD μ C812, and the 80C390 are all members of the 8051 family. The names of the devices provide little or no indication of the family connections.

Originally, there were some basic conventions used to identify the features of different variants of a particular 8051 device. For example, the standard 8051 had mask ROM, the 8031 had no ROM and the 8751 had UV-erasable ROM memory. This basic convention is now rarely observed. In the case of the Infineon C501, for example, different suffixes are used to distinguish different C501 versions: C501-1R, C501-1E, and so on.

Solution

The aim of this pattern is to help you decide whether you should use a Standard 8051 in your application. When making such a decision, some or all of the following questions need to be addressed:

- 1 Is the microcontroller powerful enough to perform the required tasks?
- 2 Does the microcontroller have sufficient memory 'on chip' to store the required code and data? If not, then does the microcontroller allow the use of appropriate external memory?
- 3 Does the microcontroller have appropriate on-chip hardware components (for example, CAN interface, PWM interface) to support the required tasks?
- 4 Does the microcontroller have sufficient port pins (or a suitable serial interface) to allow any required external components (such as switches, keypads, LCD displays) to be connected?
- 5 Is the power consumption of the chosen microcontroller appropriate (a particular concern with battery powered applications)?

We will consider each of these points in turn here.

Performance issues

One of the first question to be asked when considering a microcontroller for a project is whether it has the required level of performance. There are various ways in which such performance may be described: one measure is the number of machine instructions that may be executed in one second, usually expressed in MIPS (million instructions per second).

For example, in an original Intel 8051 microcontroller (and most current members of the 8051 family), a minimum of 12 oscillator cycles are required to execute a machine instruction. As a result, at best, a 12 MHz 8051 has a performance of approximately 1 MIPS.

A simple way of improving this performance is to increase the clock frequency. More modern (Standard) 8051 devices allow the use of clock speeds well beyond the 12 MHz limit of the original devices. For example, the Infineon C501 allows clock speeds up to 40 MHz: this raises the performance to around 3 MIPS.

Another way of improving the performance is to make internal changes to the microcontroller so that fewer oscillator cycles are required to execute each machine instruction. The Dallas ‘high speed microcontroller’ devices (87C520 and similar) use this approach, so that only four oscillator cycles are required to execute a machine instruction. These Dallas devices also allow faster clock rates: typically up to 33 MHz. Combined, these changes give a total performance of around 6 MIPS. Similar changes are made in members of the Winbond family of Standard 8051 devices (see the Winbond W77E58, for example) resulting in performance figures of up to 10 MIPS.

Clearly, for maximum performance, we would like to execute instructions at a rate of one machine instruction per oscillator cycle. The Dallas ‘ultra high speed’ 89C420 is the first 8051 device to achieve this: as a result, it runs at 12 times the speed of the original 8051. In addition, the 89C420 can operate at up to 50 MHz, increasing overall performance to around 40–50 MIPS.

To put these figures in context, the popular Infineon C167 family of (16-bit) microcontrollers has a modern architecture and performance level of around 10 MIPS. Clearly, therefore, in microcontroller terms, the performance of many 8051 devices is respectable.

Memory issues

The second question you need to ask is whether the microcontroller you are considering supports the memory that your application requires.

The memory architecture of the standard 8051 is shown in Figure 3.3.

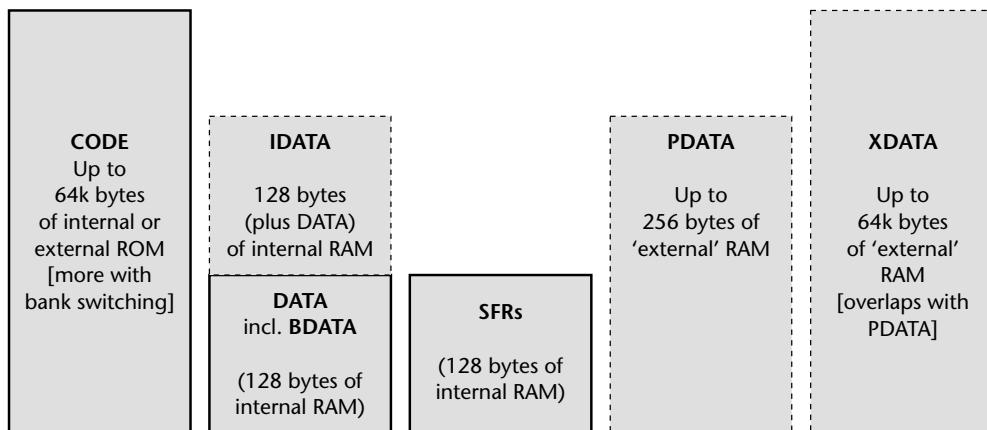


FIGURE 3.3 A schematic representation of the key memory areas on the 8051 family

[Note: that areas shown with dotted boundaries need not be present on all systems and that, as the family of 8051 devices grows, additional memory areas are available in some devices. Note also that the effective use of the various on-chip memory areas is discussed in **ON-CHIP MEMORY** [page 82].]

The first thing to note is that, for the Standard 8051, a maximum program size of 64 kbytes is directly supported. While it is possible to use larger programs by ‘bank switching’ the memory (a technique considered in **OFF-CHIP CODE MEMORY** [page 100]) this technique is complex, less efficient than a linear address space, and can be error prone. Similarly, the memory available for data is restricted to 64 kbytes. In situations where code or data memory in excess of 64 kbytes is required, use of an **EXTENDED 8051** [page 46] is often a better alternative.

The second thing to note is that Figure 3.3 shows the total memory (both internal and external) available on all Standard 8051 devices. Where possible, it is better to use a device with all the required memory on the chip, since this can improve reliability, reduce costs, reduce the application size and reduce power consumption. As discussed in ‘Background’, the original 8051 family had up to 128 bytes of RAM and 4 kbytes of ROM available, from data and code (respectively). More modern Standard 8051s rarely provide more than around 1 kbyte of RAM (usually 256 bytes), but may provide up to 64 kbytes of flash, OTP or mask ROM (see Chapter 6 for further details).

Availability of on-chip hardware components

One of the main reasons for choosing to use a microcontroller is that it integrates most or all of the hardware features you require in your application on a single chip. The 8051 family is particularly impressive in this area. There are numerous variants available which between them meet the needs of a huge number of projects, without having to resort to using large numbers of external components.

Some of the on-chip hardware components available on Standard 8051s are as follows:

- All Standard 8051s have at least one serial port available, supporting RS-232 serial protocols. This makes it easy, for example, to download data to a desktop PC. We discuss the linking of embedded and desktop systems in Chapter 18.
- All Standard 8051s have two or three timers.
- Many Standard 8051s have on-chip ‘watchdog timers’. Use of such components is discussed in Chapter 12.
- Some Standard 8051s have on-chip support for the SPI bus. We discuss this important serial bus protocol in Chapter 23.
- Some Standard 8051s have on-chip support for the I²C bus. We discuss this important serial bus protocol in Chapter 24.

Note that many more features are available in the **EXTENDED 8051** [page 46] devices.

Despite this variation, the core architecture remains the same and software for one variant can generally be used without major alteration on another.

Pin count

All Standard 8051s have four 8-bit ports available, allowing a number of external devices to be added.

Please note:

- Port 0, Port 2 and part of Port 3 are required to support external memory (if used). Use of external memory therefore has a dramatic impact on the number of spare port pins (see Chapter 6 for details).
- The availability of new ‘serial bus’ standards, like I²C and SPI (see Chapter 23 and Chapter 24), means that many peripherals may be connected to a device, by sharing a bus requiring a small number of port pins.

Power consumption

All modern implementations of Standard 8051s have at least three operating modes:

- Normal mode
- Idle mode
- Power-down mode

The ‘idle’ and ‘power-down’ modes are intended to be used to save power at times when no processing is required. Typical current requirements for the various modes are shown in Table 3.1.

TABLE 3.1 Typical current consumption figures for a selection of Standard 8051 devices

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

[Note: that figures vary (approximately linearly) with oscillator frequency: in this case, the clock frequency is assumed to 12 MHz for each device.]

The Infineon C501 is an example of a Standard 8051 device, which offers power-down modes identical to those available in the 8052 and many other modern devices. The following description of the C501 idle modes, adapted from the user manual, describes these modes in detail. Please note that this description applies equally well to most Standard 8051s.

Idle mode

In the idle mode the oscillator of the C501 continues to run, but the CPU is gated off from the clock signal. However, the interrupt system, the serial port and all timers are connected to the clock. The CPU status is preserved in its entirety.

The reduction of power consumption which can be achieved by this feature depends on the number of peripherals running. If all timers are stopped and the serial interfaces are not running, the maximum power reduction can be achieved: the developer has to determine which peripheral must continue to run and which may be stopped.

The idle mode is entered by setting the flag bit IDLE (PCON.0). Because PCON is not a bit-addressable register, the easiest way to set the IDLE bit is with the following 'C' statement:

```
PCON |= 0x01; // Enter idle mode
```

The instruction that sets bit IDLE is the last instruction executed before going into idle mode.

There are two ways to terminate idle mode:

- Activate any enabled interrupt. This interrupt will be serviced and the program will continue by executing the instruction following the instruction that sets the IDLE bit.
- Perform a hardware reset.

Power-down mode

In the power-down mode, the on-chip oscillator is stopped. Therefore all functions are stopped; only the contents of the on-chip RAM are maintained.

The power-down mode is entered by setting the flag bit PDE (PCON.1). This is most easily done in 'C' as follows:

```
PCON |= 0x02; // Enter power down mode
```

The instruction that sets bit PDE is the last instruction executed before going into power down mode. The only exit from power-down mode is a hardware reset.

Hardware resource implications

To summarize, the Standard 8051 provides the following hardware resources:

- A CPU performance of between 1 MIPS and 50 MIPS (approximately).
- Available on-chip memory of up to 64 kbytes for code and (typically) at least 256 bytes for data.
- One 'RS-232' serial port.
- Two or three hardware timers.
- Current consumption of around 20 mA in normal operating mode, 5 mA in idle mode, and 50 µA in power-down mode.

Reliability and safety implications

There are no available figures to suggest that the Standard 8051 is any more (or less) reliable than any other microcontroller family. Nonetheless, there are differences

between the facilities provided by the various 8051 family members and the choice of an inappropriate microcontroller can have a detrimental impact on the safety and / or reliability of your application.

These differences arise simply due to the availability of on-chip resources: as we have mentioned, these include – for example – different amounts of memory (RAM and ROM), serial interfaces (SPI and I²C) and analog-to-digital converters. Where possible, the use of on-chip components generally increases application reliability, for the following reasons:

- With external components, each of the soldered joints has a risk of failure, particularly in the presence of vibration and / or high humidity: reducing the number of joints reduces this risk.
- External wires act as miniature aerials and increase vulnerability to electromagnetic interference (EMI): as a consequence reduction in external wiring tends to make the application more robust in the presence EMI.
- Without external components, the complexity of the hardware design is reduced, which means there are fewer opportunities for wiring and / or hardware design errors,

As (in almost all cases) the ‘on-chip’ solution will also be both cheaper to produce and physically smaller, the message is clear: you should generally use a microcontroller with all the on-chip resources you require if at all possible. Note, however, that use of less common on-chip resources will make your design less portable (see next section).

Portability

Because of the huge range of different 8051 devices available, design based on the Standard 8051 are inherently portable. However, if you assume the availability of non-standard components (extra RAM, extra serial interfaces etc.), your design will not be as portable.

Overall strengths and weaknesses

To summarize, the Standard 8051 has the following strengths and weaknesses:

- ☺ It is a flexible, general-purpose microcontroller suitable for use in many projects.
- ☺ It is a low-cost device.
- ☺ It has an architecture with which many developers are familiar.
- ☺ It is supported by many development tools.
- ☺ The family as a whole is available in more than 300 different forms.
- ☺ It is available from a very wide range of different manufacturers: if one company fails, there will be an alternative source.
- ☺ Its CPU performance (in some versions) is equal to or greater than many 16-bit devices.

- (:(It has limited memory available (compared with 16-bit or 32-bit microcontrollers).
- (:(Its memory architecture is comparatively complex.

Related patterns and alternative solutions

In this section, we consider some alternatives to the Standard 8051 microcontroller.

Smaller alternatives

If your application does not require external memory and you do not require more than (approximately) 15 port pins for input and output, you may find that the Small 8051s are a good option: see **SMALL 8051** [page 41].

Extended 8051 alternatives

The Extended 8051s can generally do everything that the Standard 8051 can do. In addition, they usually have a larger number of available port pins and a wider range of on-chip hardware components such as digital-to-analog converters, CAN interfaces, SPI interfaces, I²C interfaces and so on. The Extended 8051s also, in some cases, provide support for large amounts of external memory: up to 16 Mbytes in some cases.

We discuss such devices in **EXTENDED 8051** [page 46].

The Intel 80251 family

The Intel MCS-251 is both software and hardware compatible with the Standard 8051 family. This means that, in most cases, you can load your existing code into a 251 and place this (40-pin or 44-pin) device into your existing 8051-based circuit board.

It is claimed that performance can be improved by a factor of 5–15 times by this approach (compared with the original 1-MIP 8051) and that further improvements are possible by recompiling and / or rewriting code to take advantage of the new architecture.

To summarize, the key features of the 251 are:

- Software and hardware compatibility with the Standard 8051.
- 5–15 × the performance of the original 8051.
- Large (up to 16 Mbyte) linear address space.
- Additional on-chip RAM compared with the Standard 8051 (up to 1 kbyte).
- Other additional components, such as a watchdog timer and a PWM unit.
- Two serial ports in some versions.

Overall, the main advantage of the 251 family is that it can provide a more powerful, drop-in replacement for the original 8051 and that it may be used without purchasing additional tools (such as compilers). However, the 251 family has not proved nearly as popular as the 8051 and it offers little that Standard and Extended 8051 devices cannot now provide.

If you wish to find out more about the 251 family, then Ayala (2000) may be of interest.

Example: Using the Standard 8051

We give many examples of the use of Standard 8051 devices throughout this book.

Further reading

A collection of data books for a range of Standard 8051 devices is included on the CD-ROM.

SMALL 8051

Context

You are developing a microcontroller-based embedded application and have some flexibility in the choice of hardware platform to be used.

Problem

Should you base your application on a Small 8051-family microcontroller?

Background

The desktop microprocessor market is characterized by constant demands for increased power. As a result, processors become obsolete after two or three years in production. By contrast, the 8051 architecture is more than 20 years old, yet the family is growing in both size and popularity.

This only makes sense because, as we saw in Chapter 1, the driving forces behind the embedded market are rather different from those of the desktop. In the embedded market, the trend is to exploit the flexibility of low-cost microcontrollers in an ever wider range of applications: indeed, as prices fall, these devices are finding their way into applications that would have involved a small number of discrete components (transistors, diodes, resistors, capacitors) a few years ago, but which are now implemented with microcontrollers.

To emphasize the very different nature of the embedded market, some of the more recent 8051 devices – far from being more powerful and having more features than the original – generally have *fewer* features.

Most immediately obvious is the fact that these Small 8051 devices typically have 20 or 24 pins and only some 15 I/O pins. In addition to their small physical size, the other common feature linking the Small 8051s is that they do not support external memory. For example, in the case of the popular AT89C1051, AT89C2051 and AT89C4051, Port 0 and Port 2¹ from the original device are omitted entirely (along with the ALE and PSEN pins, and some pins on Port 3), allowing the number of external pins to be reduced, in these cases to 20 pins. Similar changes are made in the Philips 87LPC764 and Philips 80c751 devices (see Figure 3.4).

Solution

Should you use a Small 8051 in your application?

1. As was noted in **STANDARD 8051** [page 32], Port 0 and Port 2 are used to support the address and data buses when external memory is used. See Chapter 6 for further details.

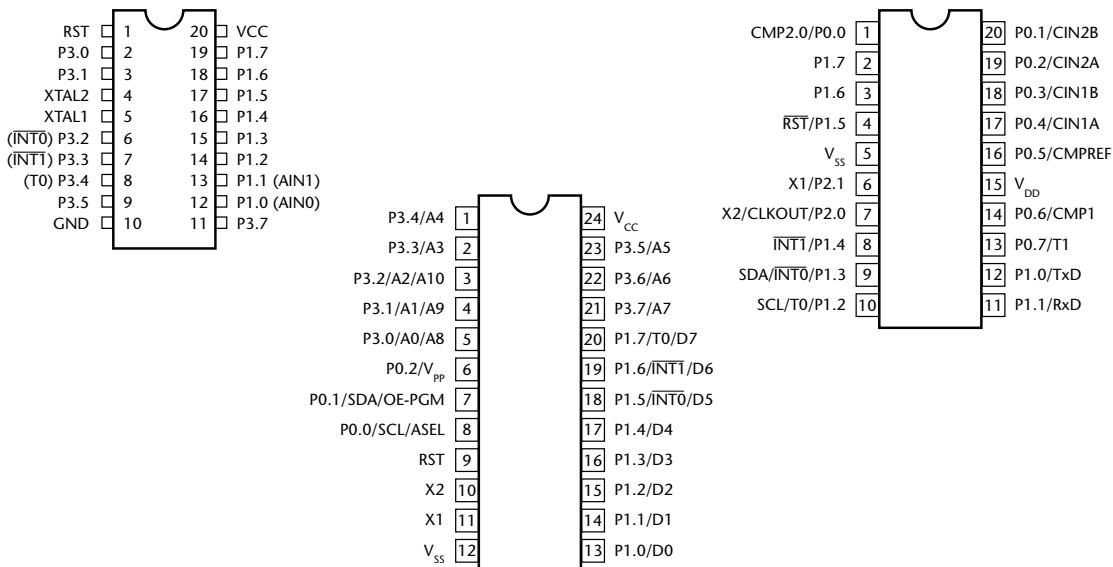


FIGURE 3.4 Examples of three ‘Small 8051s’, devices intended for applications where external memory will not be required. [Left] Atmel AT89C1051. [Middle] Philips 80c751. [Right] Philips 87LPC764 (reproduced courtesy of Atmel Corporation and Philips Semiconductors)

Performance issues

Most Small 8051s provide a CPU performance of 1–2 MIPS (see **STANDARD 8051** [page 30]).

Memory issues

A key feature of the Small 8051s is that they do not support a standard external data and address bus. As a result, the memory map of a typical Small 8051 looks like that shown in Figure 3.5.

Availability of on-chip hardware components

The on-chip hardware components on Small 8051s vary greatly between devices. The popular Atmel range has few on-chip hardware components, while the Philips devices tend to have a very wide range of features, including ADCs and pulse width modulator (PWM)² units. For example, the Philips 87LPC768 is a low-cost, 20-pin, 8051-based microcontroller with 4 kB OTP ROM memory, an 8-bit ADC and a PWM unit.

Pin count

Inevitably, Small 8051s have a very low pin count.

2. We discuss the use of PWM units in Chapter 33.

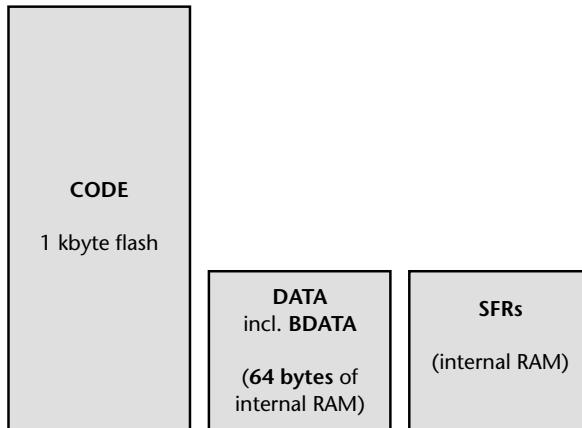


FIGURE 3.5 A schematic representation of the key memory areas on the Atmel 89C1051 device, a popular example of a Small 8051

[Note: that the 89C2051 and 89C4051 are similar, but have more RAM and 2 kbytes or 4 kbytes, respectively, of flash memory.]

Where external components must be added, either consider using a Standard 8051 as a replacement microcontroller. Alternatively, consider using a serial bus (e.g. I²C: see Chapter 23) to connect the external devices.

Power consumption

A Standard 8051 (typically) has a narrow operating voltage range: around 4.5V to 5.5V. One important feature of the Small 8051 devices is that they have a very wide operating voltage range: typically around 3V to around 7V. This wide operating voltage range makes it very easy to create low-cost, battery-powered applications.

In addition, like most Standard 8051s, the Small 8051s have three operating modes: normal, idle and power down. Typical current requirements for the various modes are shown in Table 3.2. Note also that the power supply is assumed to be 5V.

TABLE 3.2 Typical current requirements for two Small 8051 devices

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

[Note: that the figures are approximate and vary with oscillator frequency: this is assumed to be 12 MHz (in each case) here.]

Hardware resource implications

The Small 8051 provides the following hardware resources:

- A CPU performance of between 1 MIPS and 3 MIPS (approximately).
- Available internal memory (typical) of up to 4 kbytes for code and 256 bytes for data. No support for external memory.
- Usually one, full duplex ('RS-232') serial port.
- Two or three hardware timers.
- Current consumption of around 10 mA in normal operating mode, 5 mA in idle mode, and 10 µA in power-down mode.

Additional features are also available on some devices.

Reliability and safety implications

There are no available figures to suggest that the Small 8051 is any more (or less) reliable than any other microcontroller family.

Portability

Please note that the Small 8051s have a core architecture based on the 8051 family. However, as is apparent from Figure 3.4, the various Small 8051s are in no sense pin compatible and vary greatly in features and functionality. As a result, code written for a particular Small 8051 is less portable than code written for a Standard 8051.

Overall strengths and weaknesses

To summarize, the Standard 8051 has the following strengths and weaknesses:

- ☺ It is based on the core 8051 architecture and thus has many of the strengths of the Standard 8051.
- ☺ It has a small physical size.
- ☺ It is a low-cost device.
- ☹ It has limited on-chip RAM and ROM memory and no support for external memory.
- ☹ Designs based on Small 8051s are less easy to port than designs based on Standard 8051s, due to the diverse nature of this branch of the microcontroller family

Related patterns and alternative solutions

The main alternative to a Small 8051 (considered in this text) is the **STANDARD 8051** [page 30].

Alternatively, consider one of the microchip family of PIC devices, such as the (8-pin) PIC12CE673; these have similar capabilities to the Small 8051 devices, albeit with a completely different architecture.

Example: Using the Small 8051

We give many examples of the use of Small 8051 devices throughout this book.

Further reading

A collection of data books for a range of Small 8051 devices is included on the CD-ROM.

EXTENDED 8051

Context

You are developing a microcontroller-based embedded application and have some flexibility in the choice of hardware platform to be used.

Problem

Should you base your application on an Extended 8051-family microcontroller?

Background

As we have seen in connection with **STANDARD 8051** [page 30] and **SMALL 8051** [page 41], this long-lived microcontroller family continues to thrive partly because it offers a huge variety of ‘standard devices’ (covering the territory of traditional 8-bit microcontrollers) and a range of ‘small devices’ (overlapping with the range of 4-bit controllers, and challenging devices such as the small PIC range).

Both the Standard and Small 8051s are aimed, largely, at low-cost, low-performance application areas where limited memory is required and the three most important considerations are ‘cost, cost and cost’. Of course, not all projects take this form.

To develop applications requiring specialized hardware or larger amounts of memory, we can opt to switch to a 16-bit (or 32-bit) microcontroller environment. However, such a move can require a major investment in staff, staff training and development tools. An alternative is to use one of the Extended 8051 devices introduced in recent years by a range of manufacturers. Such devices preserve the investment in the 8051 range and, at the same time, open up new application areas to this microcontroller family.

In general, the extended 8051s offer the widest range of features available in 8051 devices. For example, the Infineon C505C and C515C include a useful range of on-chip hardware components (including in this case support for the CAN³ bus) that have led to these devices being used in the vast automotive market.

The C505C and C515C both retain the memory limitations of the Standard 8051. By contrast, other Extended 8051s, such as the Dallas 80C390 (Figure 3.6) and the Analog Devices ADμC812 (Figure 3.7) can access much larger amounts of memory, in a linear address space.

Compared to many 16-bit microcontrollers, the Extended 8051s are, usually, comparatively inexpensive: however, they are inevitably more expensive than either the Standard 8051 or Small 8051 alternatives.

3. We discuss the use of the CAN bus in Chapter 28.

<ul style="list-style-type: none"> ● 8051-COMPATIBLE CORE <ul style="list-style-type: none"> – 8051 instruction-set compatible – Five 8-bit I/O ports – Three 16-bit timer/counters – 256 bytes scratchpad RAM – 4 clocks/machine cycle (8051=12) – Runs DC to 40 MHz clock rates – Frequency multiplier reduces EMI – Single-cycle instruction in 100 ns – 16 total interrupt sources with 6 external – Two full-duplex hardware serial ports – SIESTA low power mode ● MEMORY <ul style="list-style-type: none"> – 4 kB internal SRAM usable as program/data/stack memory – Addresses up to 4 MB external – Defaults to true 8051 memory compatibility – User-enabled 22-bit program/data counter – 16-bit/22-bit paged/22-bit contiguous modes – User-selectable multiplexed / non-multiplexed memory interface – Optional 10-bit stack pointer – Available in 64-pin QFP, 68-pin PLCC and 64-PIN QFP 	<ul style="list-style-type: none"> ● HARDWARE MATHS SUPPORT <ul style="list-style-type: none"> – 16/32-bit math co-processor ● TWO FULL-FUNCTION CAN 2.0B CONTROLLERS <ul style="list-style-type: none"> – 15 message centres per controller – Standard 11-bit or extended 29-bit identification modes – Supports DeviceNet, SDS and higher layer CAN protocols – Disables transmitter during autobaud ● PROGRAMMABLE IRDA CLOCK ● OTHER FEATURES <ul style="list-style-type: none"> – Power-fail reset – Early-warning power-fail interrupt – Programmable watchdog timer – Oscillator-fail detection ● PACKAGING
---	--

FIGURE 3.6 Features of the Dallas 80C390 microcontroller

<ul style="list-style-type: none"> ● 8051-COMPATIBLE CORE <ul style="list-style-type: none"> – 12 MHz Nominal Operation (16 MHz Max) – Three 16-bit timer/counters – 32 programmable I/O lines – High current drive capability – port 3 – Nine interrupt sources, two priority levels ● POWER <ul style="list-style-type: none"> – Specified for 3 V and 5 V Operation – Normal, idle and power-down modes ● MEMORY <ul style="list-style-type: none"> – 8K bytes on-chip flash/EE program memory – 640 bytes on-chip flash/EE data memory – On-chip charge pump (no ext. VPP requirements) – 256 bytes on-chip data RAM – 16M bytes external data address space – 64K bytes external program address space 	<ul style="list-style-type: none"> ● ANALOG I/O <ul style="list-style-type: none"> – 8-channel, high-accuracy 12-bit ADC – On-chip, 40 ppm/oC voltage reference – High speed 200 kSPS – DMA controller for high-speed ADC-to-RAM capture – Two 12-bit voltage output DACs – On-chip temperature sensor function ● ON-CHIP ‘PERIPHERALS’ <ul style="list-style-type: none"> – UART serial I/O – I²C and SPI serial I/O – Watchdog timer – Power supply monitor ● PACKAGING <ul style="list-style-type: none"> – 52-lead plastic quad flatpack
---	---

FIGURE 3.7 Features of the Analog Devices ADμC812 microcontroller

Solution

Should you use an Extended 8051 microcontroller in your application?

Performance issues

The Extended 8051s have good levels of performance. For example, as we have previously noted, the performance of the Dallas 80C390 is up to 10x higher than the original 8051. In addition, the presence of hardware maths units in several Extended 8051s can significantly improve the speed of maths-intensive programs. Nonetheless, the Dallas 89C420 (a Standard 8051) is more powerful than any of the current Extended 8051 devices.

Memory issues

Some of the Extended 8051s support the use of large amounts of external memory. Of particular note here is the Dallas 80C390, the Analog Devices 80 μ C812 and the Philips 80C51MX. (See Chapter 6 for details.)

Availability of on-chip hardware components

Some of the on-chip hardware components available on Extended 8051s are as follows:

- Several Extended 8051s have on-chip analog-to-digital converters (typically up to eight channels, 10-bit resolution). We discuss the use of such converters in Chapter 32.
- Several Extended 8051s have hardware support for mathematical operations, ensuring that (for example) floating-point maths operations are carried out comparatively rapidly. See, for example, the data sheets for the Infineon C517, C537 and C509 and the Dallas 80C390 (included on the CD).
- Some Extended 8051s have support for the Inter-Integrated Circuit (I²C) bus. We discuss this important serial protocol in Chapter 23.
- Some Extended 8051s have on-chip support for the SPI bus. We discuss this important serial bus protocol in Chapter 24.
- Some Extended 8051s have support for the Controller Area Network (CAN) bus. We discuss the CAN bus in Chapter 28.
- Some Extended 8051s have on-chip digital-to-analog (D-A) converters. We discuss such converters in Chapter 34.

Pin count

Many Extended 8051s have a very large number of available port pins. For example, the C509 has nine 8-bit ports. Even where external memory is used, six complete 8-bit ports are available.

Power consumption

See **STANDARD 8051** [page 30] for details of the three main operating modes of the 8051 family. Inevitably, given the large number of on-chip hardware components, the basic current requirements of the Extended 8051s is larger than that of the Standard 8051. In addition, if external memory is used, the current requirements are best determined for the prototype circuit.

As a basic guide, typical current requirements for the various modes of some representative Extended 8051s are shown in Table 3.3.

TABLE 3.3 Typical current consumption figures for a range of different Extended 8051 devices

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

[Note: the figures are approximate and vary with oscillator frequency: this is assumed to be 12 MHz in each case.]

Hardware resource implications

The Extended 8051s provide a range of different hardware resources depending on the device chosen:

- In all cases, the core is 8051 compatible.
- In most cases, many additional peripheral devices are included on chip. In most cases, high CPU performance is available.
- In some cases large amounts of external memory may be directly accessed.

Reliability and safety implications

There are no available figures to suggest that the Extended 8051 is any more (or less) reliable than any other microcontroller family. However, it should be noted that many Extended 8051s require the use of external memory: this may reduce the overall system reliability compared to an otherwise identical system constructed using only internal memory, for reasons discussed in **STANDARD 8051** [page 30].

Portability

Because of the huge range of different 8051 devices available, designs based on the Extended 8051 are inherently portable. However, as already discussed, the various

'extended' 8051s share only the common core and are not pin compatible (by any means). This means that core routines (like schedulers, for example) may be easily ported, but other components will need to be adapted to suit a particular device.

Overall strengths and weaknesses

To summarize, the Extended 8051 has the following strengths and weaknesses:

- ☺ It is based on the 8051 architecture and thus has many of the strengths of the Standard 8051.
- ☺ It has – in most cases – a large number of external port pins available.
- ☺ It has – in most cases – a number of additional on-chip hardware components available.
- ☺ It has – in many cases – an ability to access large amounts of ROM and RAM memory.
- ☹ Still, essentially, an 8-bit device: for higher levels of performance, a 32-bit device may be a better option.

Related patterns and alternative solutions

We consider three alternative solutions in this section.

Use two (or more) Standard 8051s

Suppose we require a microcontroller with the following specification:

- 60+ port pins
- Six timers
- Two USARTS
- 128 kbytes of ROM
- 512 bytes of RAM
- A cost of around \$2.00 (US)

We can meet many of these requirements with an **EXTENDED 8051**: however, this will typically cost five to ten times the \$2.00 price we require. By contrast, the 'microcontroller' in Figure 3.8 matches these requirements very closely.

Figure 3.8 shows two standard 8051 microcontrollers linked together by means of a single port pin: as we demonstrate in **SCI SCHEDULER (TICK)** [page 554], linking the two processors can be done with a minimal software and hardware load. The result is a flexible environment with 62 free port pins, five free timers, two USARTs and so on. Note that further microcontrollers may be added without difficulty and the communication over a single wire (plus ground) will ensure that the tasks on all processors are perfectly synchronized.

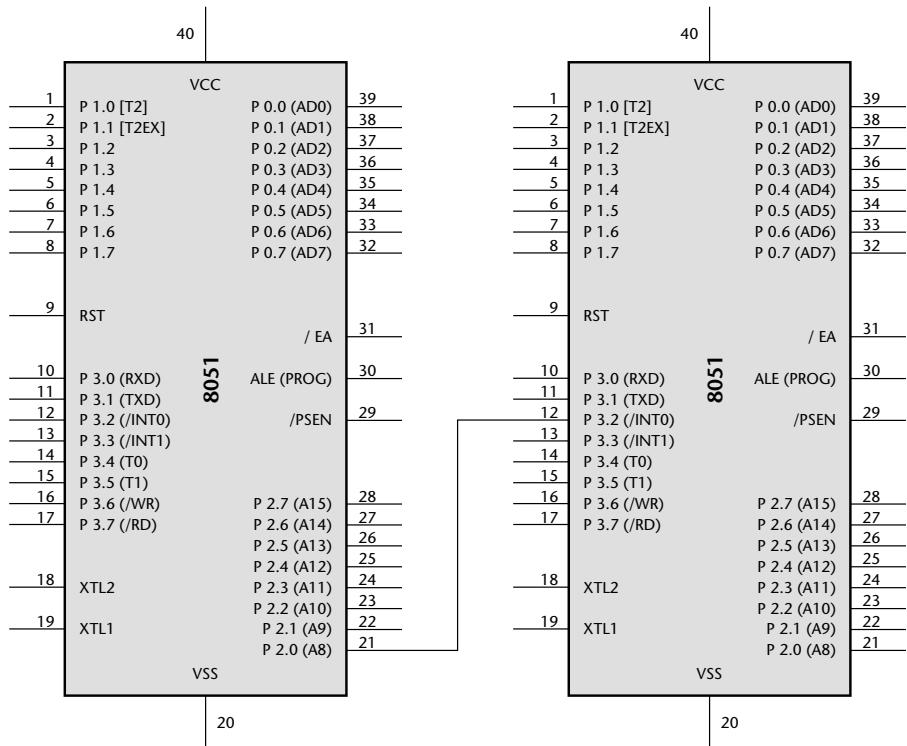


FIGURE 3.8 Creating an 'Extended 8051' from two Standard 8051s

Build your own 8051 device

If none of the available Extended 8051 devices matches your requirements, it is now possible to create your own. Specifically, Xilinx Foundation⁴ provides a comprehensive set of tools for the programming of field-programmable gate arrays (FPGAs) or application-specific ICs (ASICs). Compatible with these tools are a small range of 8051 'cores' which can be purchased from Dolphin Integration.⁵ These cores are not cheap (around \$16,000), but they are efficient (one oscillation per instruction) and the use of such techniques allows you to add hardware components to your specialized microcontroller, to meet your particular requirements.

The creation and use of such 8051 devices is beyond the scope of the present edition of this book, but the WWW sites for the companies concerned will provide further information. To make use of these techniques, you will need some familiarity with VHDL.⁶ Yalamanchili (2001) provides a good starting point.

4. www.xilinx.com

5. www.dolphin.fr

6. VHDL stands for **VHSIC** Hardware Description Language. The acronym VHSIC, in turn, stands for Very High-Speed Integrated Circuit (programme). These terms originated in a (US) Department of Defense programme which had the goal of developing a new generation of high-speed ICs. The first version of VHDL was released in 1985 and the most recent version is an IEEE standard (1076–1993).

It is worth noting that the availability of the 8051 core in this form is another, very useful consequence of the fact that this microcontroller architecture is very mature.

Use an XA-family device

The final alternative to the Extended 8051 which we will consider here is the so-called '8051XA' family, from Philips.

When developing the 251 family (discussed in **STANDARD 8051** [page 30]), Intel chose to produce a range of devices that was, to a large extent, both (executable) code and hardware compatible with the original 8051. When developing the XA family, Philips opted to follow a different route. The aim was to develop a new, 16-bit '8051' device which preserved *source* code compatibility with the 8051, but little else.

The XA family has features including dual 16 Mbyte address spaces (code and data) and fast (hardware) multiply and divide facilities. It also includes dual USARTs, an on-chip ADC and hardware support for the I²C bus.

It should be noted that very similar facilities are provided by recent Extended 8051 devices and that – unlike the Extended 8051 – the XA family requires that the developers purchase different software tools (compilers etc). In addition, the XA family has not proved particularly popular, with the result that tools, and development boards, are not very widely available.

Please refer to the Philips WWW site⁷ for further details of the XA family.

Example: Using the Extended 8051

We give many examples of the use of Extended 8051 devices throughout this book.

Further reading

A collection of data books for a range of Extended 8051 devices is included on the CD-ROM.

7. www.philips.com

4

chapter

Oscillator hardware

Introduction

All digital computer systems are driven by some form of oscillator circuit. This circuit is the ‘heartbeat’ of the system and is crucial to correct operation. For example, if the oscillator fails, the system will not function at all; if the oscillator runs irregularly, any timing calculations performed by the system will be inaccurate.

As a consequence, choice of an appropriate oscillator circuit is an important part of any hardware design. For most microcontroller-based systems, there are two main oscillator options, each of which is represented by a pattern in this chapter:

- CRYSTAL OSCILLATOR [page 54]
- CERAMIC RESONATOR [page 64]

CRYSTAL OSCILLATOR

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

When and how should you use a quartz crystal to create an oscillator for use with members of the 8051 family of microcontrollers?

Background

Quartz is a common mineral and is the main component of most sand grains. It has the useful quality that it is piezoelectric in nature, which means that if we apply pressure to a piece of quartz, it will generate an electric current at a particular frequency. In some materials, the converse is also true: application of an electric field will cause a mechanical deflection in the material.

We can use this behaviour as the basis of a useful oscillator by using an electric field (generated by plating some contacts on the surface of the mineral and applying a current) to set up mechanical oscillations in the crystal which are, in turn, converted into measurable voltage fluctuations at the surface of the crystal. We can precisely control the frequency of these fluctuations by cutting the quartz to a particular size and shape: a particular form of cut, known as the 'AT' cut, is reasonably inexpensive to produce and can create high-frequency crystals with good temperature stability at reasonable cost.

To create a complete oscillator, some further components are required. Figure 4.1 shows how crystals may be used to generate a popular form of oscillator circuit known as a Pierce oscillator.

A variant of the Pierce oscillator is common in the 8051 family. To create such an oscillator, most of the components are included on the microcontroller itself: these components are, together, sometimes referred to as the oscillator inverter. The user of this device must generally only supply the crystal and two small capacitors to complete the oscillator implementation. We discuss this further in the solution section of this pattern.

Note that, in some circumstances, it may be preferable to use a complete, self-contained external crystal oscillator module (based on a circuit like that illustrated in Figure 4.1) and use this to drive the microcontroller. We discuss this possibility in 'Reliability and safety implications'.

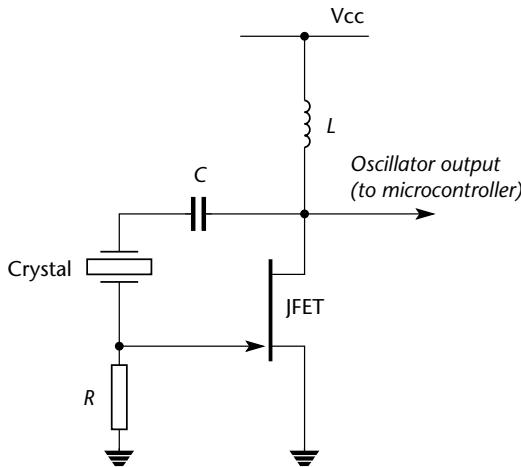


FIGURE 4.1 A Pierce oscillator circuit, driven by a quartz crystal (adapted from Horowitz and Hill, 1989)

The link between oscillator frequency and machine cycle period

When selecting an appropriate oscillator for an 8051-family device, the choice of oscillator frequency is really incidental to our real concern: the machine cycle period. That is, we are concerned with the speed at which instructions will execute.

As we discussed in Chapter 3, the various members of the 8051 family have different relationships between the oscillator cycle period and the machine cycle period. For example, in the original members of the 8051 family, the machine cycle takes 12 oscillator periods. In later family members, such as the Infineon C515C, a machine cycle takes six oscillator periods; in more recent devices such as the Dallas 89C420, only one oscillator period is required per machine cycle. As a result, the later members of the family operating at the same clock frequency execute instructions much more rapidly.

In general, the improved performance of modern implementations of the 8051 is ‘A Good Thing’: however, in situations where timing is critical, care must be taken to ensure that any timer-related calculations are implemented correctly on a particular device: see **HARDWARE DELAY** [page 194], and **CO-OPERATIVE SCHEDULER** [page 255] for further details.

Why you should keep the clock frequency as low as possible

As a general rule, the speed at which your application runs is directly determined by the oscillator frequency: in most cases, if you double the oscillator frequency, the application will run twice as fast.

In our experience, many developers select an oscillator / resonator frequency that is at or near the maximum value supported by a particular device. For example, the Infineon C505/505C will operate with crystal frequency of 2–20 MHz and many people automatically choose values at or near the top of this range, in order to gain maximum performance.

This can be a mistake, for the following reasons:

- Many applications do not require the levels of performance that a modern 8051 device can provide.
- In most modern (CMOS-based) 8051s, there is an almost linear relationship between the oscillator frequency and the power supply current. As a result, by using the lowest frequency necessary it is possible to reduce the power requirement: this can be useful in many applications.
- When accessing low-speed peripherals (such as slow memory or LCD displays), programming and hardware design can be greatly simplified – and the cost of peripheral components, such as memory latches, can be reduced – if the chip is operating more slowly.
- The electromagnetic interference (EMI) generated by a circuit increases with clock frequency.

In general, you should operate at the *lowest* possible oscillator frequency compatible with the performance needs of your application.

0 MHz operating frequencies?

Several modern 8051 family members can be operated at speeds down to 0 Hz: for example, the Atmel 89C52 device has an operating range from 0 to 24 MHz. This facility can allow significant power savings, through operating the system at very low frequencies (in kiloHertz or even Hertz, rather than in megaHertz). We make use of these features in **ONE-YEAR SCHEDULER** [page 919].

In some applications, even 0 Hz can be useful. At first glance, this may not make sense: at 0 Hz, the device is not operating and no code will execute. However, in devices designed for low-frequency operation, the system state will be maintained even if the clock frequency is reduced. This means that the clock frequency can be reduced to 0 to save power. In addition it means that, if the clock temporarily fails (for whatever reason) and then recovers, your system has a better chance of recovering, too.

Solution

The aim of this pattern is to help you decide if you should use a quartz crystal with your 8051 microcontroller and, if so, how to connect such a device. This section directly addresses these issues.

Stability issues

A key factor in selecting an oscillator for your system is the issue of oscillator stability. In most cases, oscillator stability is expressed in figures such as ‘ ± 20 ppm’: ‘20 parts per million’.

To see what this means in practice, consider that there are approximately 32 million seconds in a year.⁸ In every million seconds, your crystal may gain (or lose) 20 seconds. Over the year, a clock based on a 20 ppm crystal may therefore gain (or lose) about 32×20 seconds, or around ten minutes.

Standard quartz crystals are typically rated from ± 10 to ± 100 ppm and so may gain (or lose) from around 5 to 50 minutes per year. Note that this figure also applies to external oscillator modules. If you require greater accuracy than this, refer to 'Related patterns'.

Cost issues

Crystals cost around twice the price of a ceramic resonator, with prices linked to the crystal stability.

How to connect a crystal to a microcontroller

Basic connections for a crystal oscillator are given in Figure 4.2.

The values of the capacitors will vary, depending on the microcontroller and the crystal frequency. We will provide examples of recommended capacitor values for a range of different 8051 devices in the examples that follow; please refer to the data sheet describing your chosen microcontroller for further information. In the absence of specific information, a capacitor value of 30 pF will perform well in most circumstances.

Hardware resource implications

Use of a crystal oscillator has no direct implications for the CPU or memory requirements in your application in most cases. However, if you choose to make temperature measurements in order to increase the stability of your oscillator (see 'Reliability and safety issues'), this will have a CPU and memory overhead.

Note also that, as discussed in the background section, the performance of your application is directly related to the crystal frequency. If your application cannot perform sufficiently rapidly, consider increasing the oscillator frequency. Alternatively, consider using a more modern 8051 design, from Dallas or Infineon (for example) that uses fewer clock cycles to carry out each instruction.

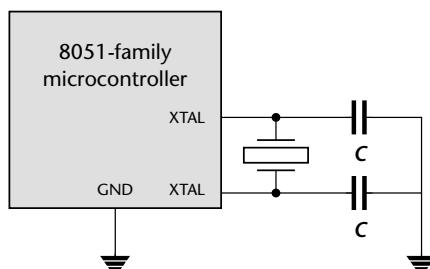


FIGURE 4.2 A simple crystal oscillator circuit

8. $(365 \text{ days}) \times (24 \text{ hours}) \times (60 \text{ minutes}) \times (60 \text{ seconds}) = 31,536,000 \text{ seconds.}$

Reliability and safety implications

We consider some reliability and safety issues related to the use of crystal oscillators in this section.

System heartbeat

The oscillator forms the ‘heartbeat’ of any digital computer. If this heartbeat stops, your system will stop. If this heartbeat varies, timing loops, delays, generated waveforms etc. will vary too. Correct operation of your embedded system relies therefore on the provision of a robust and regular clock input.

Heart of glass

Quartz is similar to glass in some physical characteristics: in particular, it is fragile.

If you require an oscillator that will operate in an environment where there is significant vibration, then quartz may not be the ideal choice. If you use a quartz crystal in these circumstances, you will need to package your application to avoid vibration influencing the operation of your system.

Time taken for oscillator to start

If the start of the (crystal) oscillator in your circuit is delayed, then the reset cycle may be completed before the oscillation begins. If this happens, the chip will not be reset.⁹

The time taken for a crystal oscillator to start operating depends on its being mounted correctly and having appropriate capacitors. Typical start-up times are 0.1 to 10 ms (Mariutti, 1999).

Using an external crystal oscillator module

As we noted in ‘Background’, it is possible to use a self-contained external crystal oscillator module (based on a circuit like that illustrated in Figure 4.1) to drive the microcontroller. This technique has the considerable advantage that the oscillator is guaranteed to start. This can make it a good solution if your system must operate very reliably.

Connecting an oscillator module is very straightforward. Figure 4.3 shows a circuit that will work with all members of the 8051 family. Note that, as shown in the figure, pin XTAL1 should be driven, while XTAL2 is left unconnected.

Particularly where higher clock frequencies (> 12 MHz) are being used, then modules may improve your system reliability. However, oscillator modules do have several drawbacks:

- Oscillator modules cost around twice the price of a crystal oscillator and four times as much as a ceramic resonator.
- Oscillator modules typically draw currents comparable to that of an 8051 microcontroller: 15–35 mA. This may represent a very significant power drain in battery-powered applications.

9. See **RC RESET** [page 68] for further details.

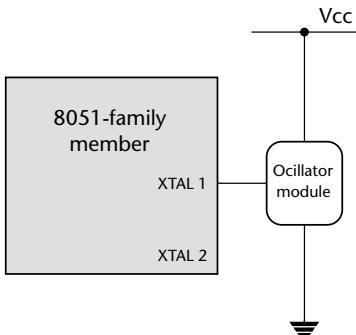


FIGURE 4.3 Using an external oscillator module

- Oscillator modules are not always easy to obtain in ‘odd’ frequencies, such as 11.059 MHz. This frequency is very useful in 8051-based designs involving a serial interface, as discussed in ‘Related patterns and alternative solutions’.

Improving the stability of a crystal oscillator

As we have discussed, typical crystal oscillators have a stability of around $\pm 20\text{--}100$ ppm. If we use this device to control a real-time clock we may gain or lose up to 50 mins per year: that is, up to ~ 1 minute / week. This result is not specific to the 8051 family: the result of this behaviour is evident even in expensive servers for desktop computer networks. By contrast, most ‘quartz’ wristwatches use crystal oscillators, cost very little and keep very good time. This is because they have a sophisticated temperature control system attached, which keeps them operating at a temperature of 35°C for about 16 hours every day. The temperature control system is your wrist (and attached biological mechanisms).

If you want a general crystal-controlled embedded system to keep accurate time, you can choose to keep the device in an oven (or fridge) at a fixed temperature and fine-tune the software to keep accurate time. This is, however, rarely practical. Instead, ‘temperature compensated crystal oscillators’ (TCXOs) are available that provide – in an easy-to-use package – a crystal oscillator and circuitry that compensates for changes in temperature. Such devices provide stability levels of up to ± 0.1 ppm (or more): in a clock circuit, this should gain or lose no more than around 1 minute every 20 years. Such levels of accuracy are adequate for all but the most demanding of applications. However, there is a catch. TCXOs can cost in excess of \$100.00 per unit and may even cost several times this amount. This price puts them well out of reach of most embedded projects.

One practical alternative is to determine the temperature-frequency characteristics for your chosen crystal and include this information in your application. For the cost of a small temperature sensor (around \$2.00), you can keep track of the temperature and adjust the timing as required. This is the basis of **STABLE SCHEDULER** [page 932].

Another alternative is to use an atomic clock. For example, a caesium beam clock uses atomic transitions as the reference for a crystal oscillator and can provide accuracy at a level of a few parts per million million (that is, around 1 in 10^{12}). This translates into an accuracy of around 1 minute every million years. Use of such a device may sound like an outlandishly expensive solution, but you can now access the atomic clocks in various satellites using global positioning system (GPS) receivers or GPS chip sets. This is the approach increasingly used by the mobile phone (cell phone) companies that include such technology on their base stations.

Portability

These techniques can be, and are, used with a wide range of microcontrollers and microprocessors.

Note that, as discussed in the ‘Solution’ section, the value of capacitors to be used depends on both the crystal frequency and the microcontroller used. The manufacturer’s data sheet for the microcontroller will provide recommended values.

Overall strengths and weaknesses

- 😊 Crystal oscillators are stable. Typically $\pm 20\text{--}100 \text{ ppm} = \pm 50 \text{ mins per year}$ (up to $\sim 1 \text{ minute / week}$).
- 😊 The great majority of 8051-based designs use a variant of the simple crystal-based oscillator circuit presented here: developers are therefore familiar with crystal-based designs.
- 😊 Quartz crystals are available at reasonable cost for most common frequencies. The only additional components required are usually two small capacitors. Overall, crystal oscillators are more expensive than ceramic resonators.
- 😢 Crystal oscillators are susceptible to vibration.
- 😢 The stability falls with age.

Related patterns and alternative solutions

An alternative solution

The main alternative to an external crystal oscillator is an external ceramic resonator: see CERAMIC RESONATOR [page 64].

Using an on-chip oscillator

As we saw in Chapter 3, Small 8051 devices, such as the popular Atmel 89C4051, are designed as flexible, cost-effective replacements to discrete circuits (assembled from transistors, resistors, capacitors etc.). However, these devices still require external oscillator (and reset) circuits. As the oscillator and reset components can, together, cost as much as these small microcontrollers and greatly increase the board size, it

would seem sensible to include oscillator (and reset) circuits within the microcontroller itself.

This is now possible, with 8051-family devices such as the Philips 87LPC764. This 20-pin device includes an on-board reset circuit (see Chapter 5). It also includes an on-chip resistor-capacitor (RC) oscillator. It can therefore be used without any external components.

Increasingly, RC oscillators (also known as ‘relaxation oscillators’) are becoming available as on-chip components. A simple implementation of such an oscillator is illustrated in Figure 4.4.

Other implementations of this simple oscillator are possible using, for example, small numbers of logic gates. However – whatever the implementation – the problem with the RC solution is that the oscillator can never be very stable, largely due to the variation of the resistor values with temperature. This is apparent in many practical implementations: for example, the RC oscillator on the 87LPC764 (and similar devices) has a stability of only $\pm 25\%$. This is not sufficient for many applications: for instance, if used to generate baud rates for a serial interface, this level of stability would mean that the communication was unlikely to be effective.

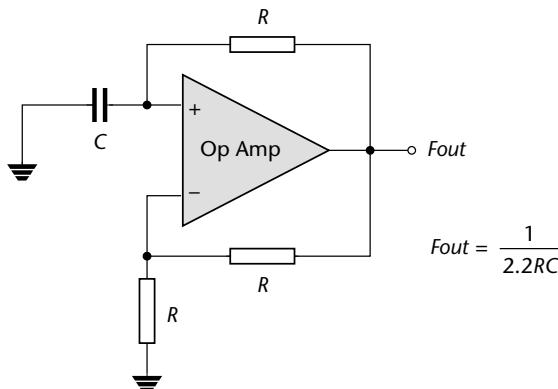


FIGURE 4.4 A simple op amp-based RC oscillator (adapted from Warnes, 1998)

[Note: that other implementations are possible, using, for example, small numbers of logic gates].

Timing issues

Do not assume that just because your microcontroller will operate over a wide range of frequencies that you are free to choose any frequency in this range. The choice of oscillator frequency will have a major impact on any time-related aspects of your application.

For example, you will see numerous designs for 8051-based systems which use crystal frequencies of 11.0592 MHz. The reason why this frequency is used is that, with standard 8051 devices, this crystal frequency may be easily used to generate standard baud rates (such as 9600 baud) from the built-in serial port: with other frequencies

(e.g. 10 MHz, 12 MHz) it is more difficult to produce these standard baud rate values. This issue is discussed in greater depth in Chapter 18.

Similarly, the oscillator frequency dictates the rate at which the hardware timers in your application will be incremented. If you need, for example, to schedule a task to run precisely every one minute, this can be difficult to achieve if you have selected an inappropriate oscillator frequency. (See Chapter 14 for further details.)

Note that such peculiar numbers are not restricted to the 8051 family. ‘Quartz’ digital wristwatches use a frequency of 32.768 kHz, since, by dividing this frequency by 2^{15} , you obtain a 1 Hz ‘tick’ ($2^{15} = 32,768$).

Example: Attaching a crystal to an Atmel 89C2051

Recommended capacitor values for connecting most quartz crystals to an Atmel 89C2051 are shown in Figure 4.5.

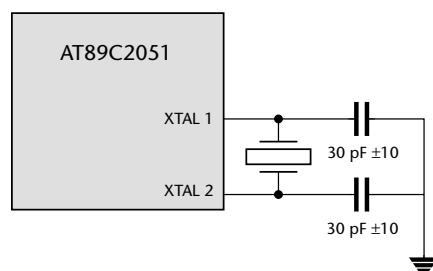


FIGURE 4.5 Connecting a crystal oscillator to an AT89C2051

Example: Attaching a crystal to dual-processor board

It should be noted that most crystals or oscillator modules will drive more than one (typically up to five) microcontrollers. The data sheet will specify this ‘fan out’ value. This can be useful where a multiprocessor design is planned, not least because this means that both microcontrollers (assuming they are both 8051s) will always be ‘in step’ (see Part F for further details).

Figure 4.6 illustrates how the two boards should be connected to the same crystal. Note that the same approach can be applied to any combination of microcontrollers: if the two boards require different capacitor values, then select a mid-range value.

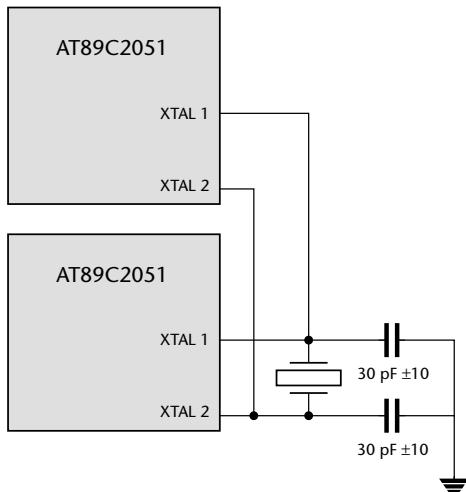


FIGURE 4.6 One crystal can typically drive several microcontrollers

Further reading

Refer to the manufacturer's data sheet for your chosen microcontroller to ensure you use the required capacitor values in your crystal oscillator circuit.

CERAMIC RESONATOR

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

When and how should you use a ceramic resonator with members of the 8051-family microcontrollers?

Background

A ceramic resonator is, like a quartz oscillator, based on a piezoelectric material. In this case, the material is (as the name suggests) a form of piezoelectric ceramic.

See **CRYSTAL OSCILLATOR** [page 54] for additional background material.

Solution

The aim of this pattern is to help you decide if you should use a ceramic resonator with your 8051 microcontroller and, if so, how to connect such a device. This section directly addresses these issues.

Stability issues

As discussed in **CRYSTAL OSCILLATOR** [page 54], a key factor in selecting an oscillator for your system is the issue of oscillator stability. Unlike crystal oscillators, which usually have stability measured expressed in parts per million, ceramic resonator stability is usually stated in percentage terms. A figure of 1% stability is common. There are 1,440 minutes in a day and a clock based on a 1% ceramic resonator could expect to gain (or lose) around 14 minutes every day. Clearly, such devices are not suitable for operations requiring accurate timing over a long period. Note, however, that if we require the resonator to form the basis of a 30-second delay, the likely gain or loss is 0.3 seconds: this may not be a problem.

Cost issues

Ceramic resonators cost half the price of a crystal oscillator.

External capacitors

Most ceramic resonators include internal capacitors. They may therefore be directly accessed to the microcontroller without the need for external capacitors. This makes them easy to use and can further reduce costs and the required board size.

Hardware resource implications

Use of a ceramic resonator has no direct implications for the memory requirements in your application.

Note also that the performance of your application is directly related to the resonator frequency. If your application cannot perform sufficiently rapidly, consider increasing this frequency. Alternatively, consider using a more modern 8051 design, from Dallas or Infineon (for example) that requires fewer clock cycles to carry out each instruction.

Reliability and safety implications

See **CRYSTAL OSCILLATOR** [page 54] for a general discussion of reliability and safety issues associated with oscillators.

Overall, the ceramic resonator is the most physically robust form of oscillator we consider.

Portability

These techniques can be, and are, used with a wide range of microcontrollers and microprocessors.

Please note that ceramic resonators should not, generally, be used as plug-in replacements for crystal oscillators: different capacitors (if any) are required for each solution.

Overall strengths and weaknesses

- ☺ **Cheaper than crystal oscillators.**
- ☺ **Physically robust: less easily damage by physical vibration (or dropped equipment etc.) than crystal oscillator.**
- ☺ **Many resonators contain in-built capacitors and can be used without any external components.**
- ☺ **Small size. About half the size of crystal oscillator.**
- ☹ **Comparatively low stability: not general appropriate for use where accurate timing (over an extended period) is required. Typically ± 5000 ppm = ± 2500 min per year (up to ~50 minutes / week).**

Related patterns and alternative solutions

CRYSTAL OSCILLATOR [page 54] describes the main alternative.

Example: Connecting a ceramic resonator to an 8051 microcontroller

Many simple consumer applications, where accurate timing is not required and cost is an issue, make use of ceramic resonators. In most cases, resonators with internal capacitors are used: the same resonator can be used with any member of the 8051 family (Figure 4.7).

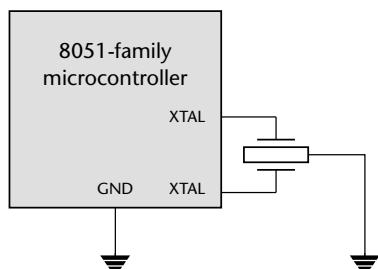


FIGURE 4.7 A simple ceramic resonator circuit, for use where the resonator has internal capacitors

[Where no such capacitors are included, the ‘quartz crystal’ circuit (Figure 4.2) should be used: refer to the data sheet for your microcontroller for recommended capacitor values. Note that it is easy to distinguish the different types of resonator: those without capacitors have two pins (like crystals); those with capacitors have a third pin. Where there are three pins, the middle pin should be grounded.]

Further reading

chapter 5

Reset hardware

Introduction

The process of starting any microcontroller is a non-trivial one. The underlying hardware is complex and a small, manufacturer-defined ‘reset routine’ must be run to place this hardware into an appropriate state before it can begin executing the user program. Running this reset routine takes time and requires that the microcontroller’s oscillator is operating.

Where your system is supplied by a robust power supply, which rapidly reaches its specified output voltage when switched on, rapidly decreases to 0V when switched off, and – while switched on – cannot ‘brown out’ (drop in voltage), then you can safely use low-cost reset hardware based on a capacitor and a resistor: this form of reset circuit is addressed in **RC RESET** [page 68].

Where your power supply is less than perfect, and / or your application is safety related, the simple RC solution will not be suitable. **ROBUST RESET** [page 77] discusses a more reliable alternative.

RC RESET

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you create a low-cost reset circuit for your 8051 microcontroller?

Background

As discussed in the introduction to this chapter, the reset process which must be completed prior to the execution of any other code requires that the microcontroller's oscillator is operating. To trigger the reset operation, the original members of the 8051 family have a 'RESET' pin. When this is held at Logic 0, the chip will run normally. If, while the oscillator is running, this pin is held at Logic 1 for two (or more) machine cycles, the microcontroller will be reset.

Note that, if the reset operation is not completed correctly, the microcontroller will usually not operate at all: in rare circumstances, it may operate, but incorrectly. In either event, there is usually nothing that you can do, in software, to recover control of the system. Clearly, therefore, ensuring correct reset operation is a crucial part of any application.

Solution

Various techniques may be used to ensure that – when power is applied to your 8051-based application – the reset process is automatically carried out. The most widely used techniques are based on the use of an external capacitor and resistor: these techniques are considered in detail here.

RC reset circuits

A typical RC reset circuit is as shown in Figure 5.1.

The circuit in Figure 5.1 operates as follows. We assume that V_{cc} is initially at 0V (that is, the power has not been applied to the system) and that the capacitor C is fully discharged. When power is applied, the capacitor will begin to charge. Initially, the voltage across the capacitor will be 0V and – therefore – the voltage across the resistor (and the voltage at the RESET pin) will be V_{cc}: this is a Logic 1 value. Gradually, the capacitor will charge and its voltage will rise, eventually to V_{cc}: at this time, the voltage at the reset pin will be 0V.

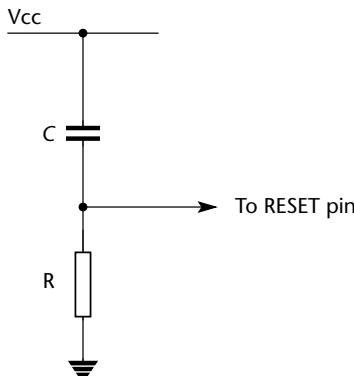


FIGURE 5.1 An (active high) RC reset circuit

In the real system, the microcontroller's input voltage threshold is around 1.1 – 1.3V¹⁰: input voltages below this level are interpreted as Logic 0 and voltages above this level are interpreted as Logic 1. Thus, the reset operation will continue until the voltage at the RESET pin falls to a level of around 1.2V.

We can use this information to calculate the required values of R and C. To make this calculation, we use the fact that the capacitor in Figure 5.1 will have a voltage (V_{cap}) at time (t) seconds after it begins charging, given by Equation 5.1.

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

EQUATION 5.1 The voltage across the capacitor in Figure 5.1 as a function of time

Note that Equation 5.1 assumes that the capacitor begins charging at a voltage of 0 and that the power supply voltage increases from 0V to Vcc in an instantaneous 'step' (rather than a slow ramp): these assumptions, although often made, are frequently invalid: see 'Safety and reliability issues' for a discussion of these issues.

The Intel 8051 data sheet recommends values of 8.2K for R and 10uf for C when this form of reset circuit is used. Figure 5.2 substitutes these values into Equation 5.1 and plots the result over a period of 500 ms.

When looking at Figure 5.2, remember that all 8051s complete their reset operation in 24 oscillator periods or less: if we use a 12 MHz oscillator, this is a maximum period of 0.002 ms: by contrast, the recommended reset circuit takes around 100 ms to complete the reset operation. This may seem like an excessive reset period

10. The data sheet for your chosen microcontroller will provide a precise value, if you require it.

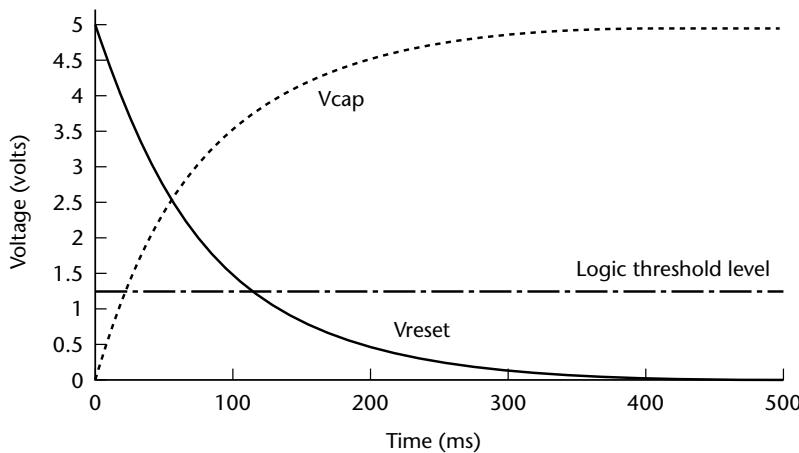


FIGURE 5.2 An example of the behaviour of an RC reset circuit using standard component values and an ideal power supply

but, for reasons discussed under ‘Safety and reliability issues’, allowing approximately 100 ms for the reset is generally good practice.

Choosing values of R and C

If, having reviewed all aspects of this pattern, you have decided to use an RC-based reset circuit, what values of R and C should you use?

Rather than trying to determine values of R and C directly from Equation 5.1, we can simplify matters by noting that the product of R (in Ohms) multiplied by C (in Farads) is known as the ‘time constant’ (in seconds) of this form of RC circuit. This time constant is the time taken for the capacitor to be charged to 63% of its final voltage. Thus, with a 5V supply and the circuit in Figure 5.1, this is the time taken for the capacitor voltage to reach 3V and, therefore, the voltage at the reset pin to reach 2V (that is, $V_{cc} - 3V$): this is still high enough (because it is greater than 1.2V, as already discussed) to ensure that the device is in reset mode. As long as the device is still in this mode until approximately 1 ms after the power supply reaches V_{cc} (typically around 100 ms after starting: see ‘Safety and reliability issues’), the device will be reset correctly.

A basic rule of thumb, therefore, is that the RC time constant should be approximately 100 ms and values of R and C chosen to meet this requirement will usually ensure effective reset operation (Equation 5.2):

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

EQUATION 5.2 A ‘rule of thumb’ for calculating appropriate RC values

A suitable RC reset circuit satisfying these conditions is shown in Figure 5.3.

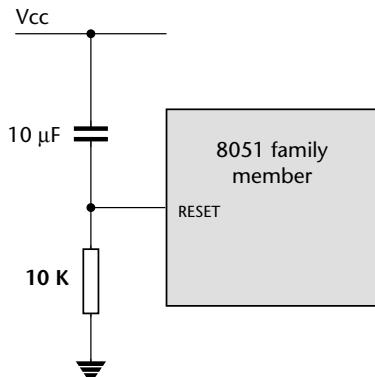


FIGURE 5.3 A suitable (active high) RC reset circuit

We can summarize the key material in this section as follows:

- A combination of a 10K resistor and a 10 μ F capacitor in a RC reset circuit gives a 100 ms time constant. Bearing in mind the general limitations of RC reset circuits (see ‘Safety and reliability issues’), this value is suitable for the majority of 8051-based systems.
- The standard 8K2, 10 μ F RC reset combination gives a time constant of 82 ms: this is generally adequate.
- Values of 1K and 10 μ F (which appear in some books) provide a time constant of only 10 ms: these values will not provide a reliable reset operation with all power supplies.

Adding a RESET button

In some systems, it is helpful to have a reset button, to force a hardware reset. This is easy to achieve. Figure 5.4 shows a suitable circuit.

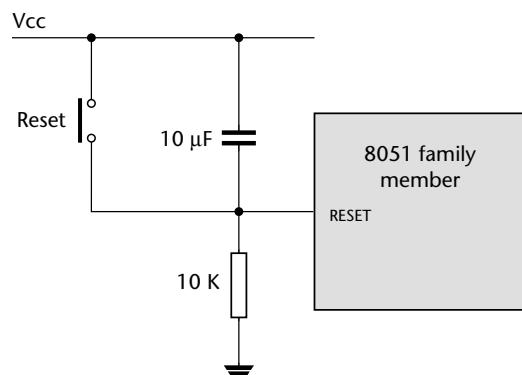


FIGURE 5.4 A reset circuit (active high) with reset switch

Note that the reset button pulls the RESET pin (assumed to be active high: see ‘Portability’) to Vcc. Note also that this button also discharges the capacitor, ensuring that – when the switch is released – the proper reset process will be carried out.

Hardware resource implications

This pattern has no implications for CPU or memory usage.

Reliability and safety issues

There are a number of reliability and safety issues related to the use of RC reset circuits. The key issues are considered in this section.

Overall, however, we make the recommendation as seen in the box.

Many of the reliability problems with embedded systems can be traced back to defects in the reset circuit. If cost is the **only** concern, consider using an RC reset: if reliability is a consideration, use a **ROBUST RESET** [page 77].

Time taken for power supply to reach steady state

Suppose you are developing an embedded industrial control system and you want to ensure that the system begins operating as soon as possible after power is applied. You note (from ‘Solution’) that the reset process on an 8051 microcontroller (with a 12 MHz oscillator) will take 0.002 ms. You conclude that, allowing a reset period of 1 ms (rather than the 100 ms figure recommended earlier) will provide sufficient margin for error.

Suppose you adjust the values to reduce the reset period to around 1 ms. For example, Figure 5.5 shows the result of using a 0.1 μ F capacitor and a 6K7 resistor.

This combination of values may, **sometimes**, work: but in most systems it will fail. The reason is that real power supplies do not switch instantly from 0V to their specified output voltage: in reality, many supplies take 50 ms or 100 ms to reach this voltage when first switched on. You need to allow for this ‘ramped’ voltage input in your design.

If the supply voltage increases slowly, then the capacitor in your RC reset circuit will comparatively quickly charge up and will simply ‘follow’ the increasing power supply voltage. As a result, Vreset will be held at Logic 0 many milliseconds before the chip reaches its operating voltage (~5V). Therefore, the chip will only be ready to run its reset routine after the RESET signal is complete and no reset will be performed. Your application will therefore not start correctly.

If you really must have a rapid reset and you have control over the design of the power supply there are various ways of dealing with this problem. You may, for example, be able to increase the transformer capacity or reduce the values of these filter capacitors. Note, however, that tying the operation of a device to a particular power

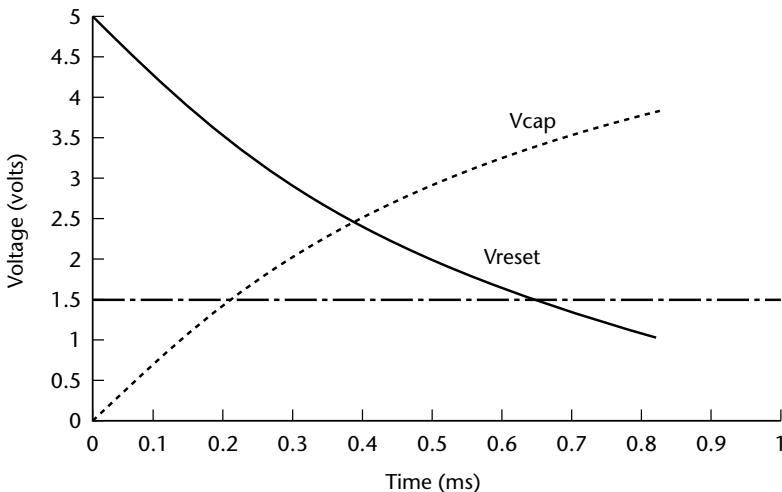


FIGURE 5.5 Using a rapid RC reset circuit

supply will make your system design much less portable. If, to give a common example, your company subsequently decides to ‘outsource’ the power supplies or to use a single power supply across a range of different boards, you can quickly run into difficulties.

Time taken for oscillator to start

If the start of the (crystal) oscillator in your circuit is delayed the RC reset cycle may be completed before the oscillation begins. If this happens, the chip will not be reset.

Typical start-up times for crystal oscillators are 0.1 to 10 ms: however, the time taken for a crystal oscillator to start operating depends on its being mounted correctly and having appropriate capacitors. These issues are discussed in detail in the pattern **CRYSTAL OSCILLATOR** [page 54].

Handling brownouts and other power disruptions

Potential problems with reset circuits do not, unfortunately, only arise when embedded devices are first powered up. Consider, for example, Figure 5.6. This shows changes in the system supply voltage (nominally 5V) in the presence of two problems. The first of these (at time = 4 seconds) is a simple power ‘glitch’, where the supply voltage drops briefly to 0V. The second problem (beginning at time = 14 seconds) is a ‘brownout’ condition: this means that the (mains) supply voltage is reduced significantly for a period of time, but the supply does not fail completely. These types of fault are comparatively common in mains-powered systems.

To ensure our system operates in a predictable manner, we need to be able to deal with each supply problem. In most systems, the power glitch will not pose a significant hazard. When the power fails, the voltage drops rapidly (to 0V) and the system will stop operating. When the power returns, the system will be reset in the usual way.

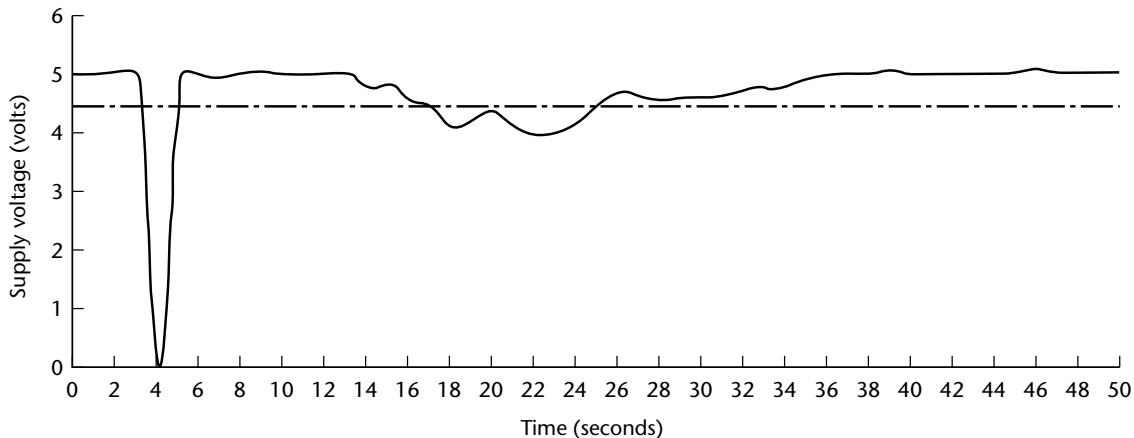


FIGURE 5.6 Examples of voltage fluctuations caused by ‘glitches’ and ‘brownouts’

The brownout is potentially more problematic. If the supply voltage drops below the minimum operating voltage (typically 4.5V for most members of the family, although this varies), the microcontroller will stop operating. If the voltage then rises again, the microcontroller will begin to operate again; however, if using a simple RC reset, the device will not be reset. The results are difficult to predict and the RC reset circuit is therefore neither reliable nor safe if brownouts are a possibility: see ‘Related patterns and alternative solutions’ for some alternative techniques.

Portability

Some portability issues, related to the different performance of various power supplies, have been considered elsewhere in this pattern. These will not be discussed further here.

Note also that, as discussed in a following example, not all 8051 family members have ‘active high’ resets: some are ‘active low’. While the underlying principles are the same, the wiring of ‘active high’ and ‘active low’ resets are fundamentally incompatible (see ‘Example: Working with active low resets’).

Overall strengths and weaknesses

- ☺ RC reset circuits are cheap to implement.
- ☺ RC resets are well understood and widely used in other microprocessor and microcontroller systems.
- ☺ If your system is mains powered and safety and reliability are not issues (and cost is) this technique may be a good solution.

-  If the system power supply characteristics are unknown or vary or are subject to brownout, the reset operation may not always be effective: RC resets are generally not suitable for main-powered applications which must be reliable or safe.

Related patterns and alternative solutions

The pattern **ROBUST RESET** [page 77] describes a more expensive but generally much more reliable reset solution.

Example: Minimal Atmel 89C2051 circuit with crystal and RC reset

A minimal Atmel 89C2051 circuit, using an RC reset, is shown in Figure 5.7. Please see the pattern **CRYSTAL OSCILLATOR** [page 54] for details of the oscillator circuit.

Note that the Atmel device does not support external memory so that the /EA pin is not present: see the ‘memory patterns’ (Chapter 6) for details.

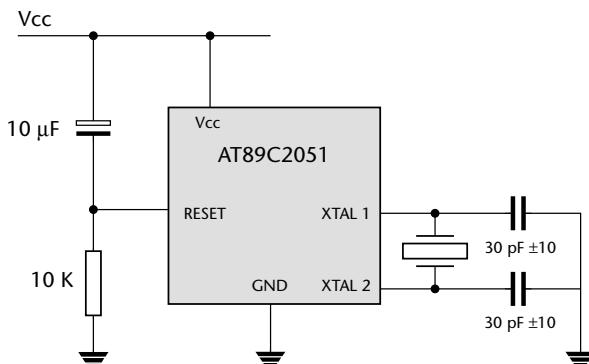


FIGURE 5.7 A minimal Atmel AT89C2051 circuit, with RC reset

Example: Working with active low resets

The reset circuits considered so far have been ‘active high’ in nature. This means that normally the RESET pin will be held at a low level (~0V): to effect a reset, the RESET pin needs to be pulled high (~Vcc), while the oscillator is running.

However, some 8051 devices have ‘active low’ inputs: these can be identified by the presence of a RESET pin. As the name suggests, these pins are held at a high level during normal operation and must be pulled low (again usually for 24 clock cycles) to effect a reset. Examples of 8051 devices with active low inputs include the Infineon C509, C515C and C517A. All of these are popular and widely used devices.

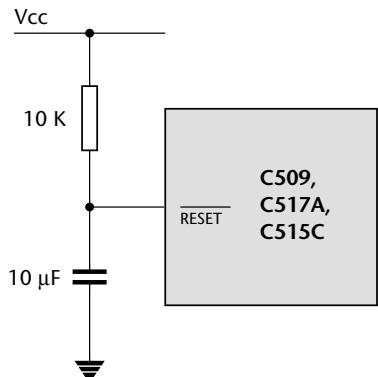


FIGURE 5.8 Creating an 'active low' RC reset circuit

Wiring an active low reset circuit is straightforward. Figure 5.8 shows a possible circuit for these various active low devices.

Further reading

ROBUST RESET

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you create a very reliable reset circuit for your 8051 microcontroller?

Background

See **RC RESET** [page 68] for some background material on reset circuits.

Solution

As discussed in **RC RESET** [page 68], a key problem for the designers of reliable embedded systems is that power supplies are seldom ideal. For example, mains power supplies can take around 100 ms to reach their normal operating voltage when first switched on and may subsequently suffer from voltage variations, including brownouts, in part due to variations in the network load caused by other users of the mains supply. On the other hand, '1.5V' primary cells ('batteries') often begin life with a voltage of up to 1.7V and decline to 0.9V or less as they are used.

Although careful design of your power supply can reduce these problems, it is often desirable to reduce the risks of system failure by maximizing the chances that your system will be reset correctly in the event of power supply problems. The robust reset circuits, notably from Dallas and Maxim, directly address this problem.

These useful devices perform two basic operations:

- When power is applied to a 'cold' system, they apply an appropriate (active high or active low) reset signal to the microcontroller for at least 100 ms to allow the oscillator (if used) and power supply to reach their operational state.
- If the supply voltage falls below a preset value during normal operation, the reset cycle will begin and will only end ~100 ms after the supply is restored to the normal value. This behaviour deals with both total power failures (long or short) and brownouts.

Overall, these devices are very effective and are cheap. Except where cost is the **only** concern, they are highly recommended.

Hardware resource implications

This pattern has no implications for CPU or memory usage.

Reliability and safety issues

This form of reset circuit is, as a general rule, much safer than any RC-based equivalent.

Portability

The various robust reset chips are generally insensitive to power supply variations over a wide range: use of such devices therefore tends to make your design more portable and less dependent on a particular power supply. Most devices are available with both ‘active high’ and ‘active low’ versions and are therefore compatible with a wide range of 8051 chips.

Overall strengths and weaknesses

- ☺ Robust reset provides reliable performance even with ‘slow’ power supplies and in the event of brownout.
- ☹ More expensive than RC reset alternatives.

Related patterns and alternative solutions

RC RESET [page 68] offers a cheaper alternative where safety and reliability are much less important than product cost.

In addition, two further ways of obtaining reliable reset behaviour are now discussed.

On-chip reset circuits

Reset circuits are always a challenge with microcontroller-based systems and – now that small reset ICs are available – it seems surprising that this circuitry is not simply included in the microcontroller itself.

In fact, in more recent 8051s, such circuitry is included: see, for example, the Dallas DS87C520 / DS83C520 and the Philips 87LPC764. An example of a Dallas 87C520 circuit using internal reset components is given in the following section.

Microcontroller / microprocessor supervisor ICs

The robust reset circuits discussed here are simple, economical reset options. However, in some applications, we require other external facilities too. Maxim, in particular, have developed a range of different ‘microcontroller supervisor’ ICs that include not only the equivalent of ‘robust reset’ circuits, but also have various combinations of additional facilities, such as watchdog timers and even RS-232 transceivers. Three examples of such devices are summarized here: consult the Maxim WWW site¹¹ for further details.

11. www.maxim-ic.com

- **Max6330/1:** Reset control, plus 3V / 3.3V / 5V shunt regulator. This allows a single, inexpensive IC chip to be used for both the power supply regulation and reset circuit.
- **Max819:** Reset control plus watchdog, plus battery switchover.
- **Max3320:** Reset control, plus RS-232 transceiver.

In addition, the '1232' power monitor (available in various versions from both Dallas and Maxim) combines ROBUST RESET behaviour with a watchdog timer: as we see in Chapter 12, this is a very useful combination in many applications.

Example: Using the Dallas DS1812 'Econoreset' with the 8051 family

Please refer to Chapter 26, Figure 26.10, for one of many examples in this book that use the DS1812 robust reset with the Standard 8051 device.

Example: Using Dallas 'Econoresets' with the Infineon C515C-8E

A minimal Infineon C515C-8E circuit, created using a Dallas DS1811 robust reset, is shown in Figure 5.9.

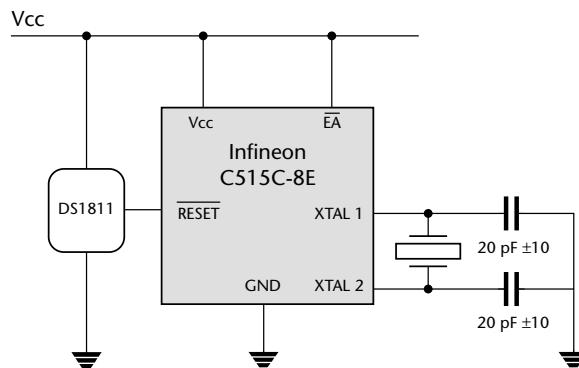


FIGURE 5.9 A minimal Infineon C515C-8E circuit, created using a Dallas DS1811 Robust Reset

Example: Using the Max810M with the Infineon C501-1E

A minimal Infineon C501-1E circuit, created using a Maxim 810M robust reset, is shown in Figure 5.10.

Example: Minimal Dallas circuit using on-chip reset circuit

As noted earlier, some more recent 8051 devices have on-chip reset circuitry.

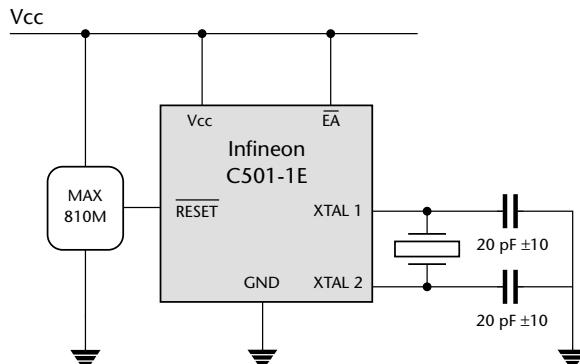


FIGURE 5.10 A minimal Infineon C501-1E circuit, created using a Maxim 810M Robust Reset

As an example, we will consider the DS87C520 (see Figure 5.11), whose circuitry operates as follows. While powering up, the internal monitor circuit maintains a reset state until V_{CC} rises above the ‘reset’ level. Once above this level, the monitor enables the oscillator input and counts 65,536 clock cycles, before leaving the reset state. This power-on reset (POR) interval allows time for the both the power supply and oscillator to stabilize. In addition, if the supply voltage drops during normal operation, power monitor will generate and hold a reset automatically.

Note that this solution works with many of the Dallas ‘high speed’ and ‘ultra high speed’ devices, including the 80C320, 80C323, 83C520, 87C520, 87C530, 87C550 and 89C420.

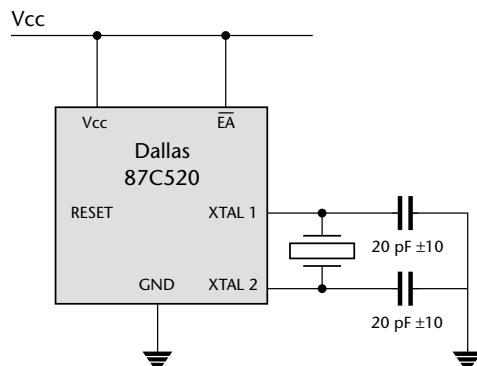


FIGURE 5.11 A minimal 87C520 circuit

[Note that no external reset circuitry is required.]

Further reading

chapter **6**

Memory issues

Introduction

All practical microcontroller-based systems require some form of non-volatile code memory (to store the program code) and some form of volatile memory (to store data and the stack).

In many cases, it is possible to create useful applications without adding external memory devices. The first pattern in this chapter (**ON-CHIP MEMORY** [page 82]) discusses how to do this by making effective use of the various memory areas available in members of the 8051 family.

In some applications, it is necessary to add external memory: the remaining patterns in this chapter (**OFF-CHIP DATA MEMORY** [page 94] and **OFF-CHIP CODE MEMORY** [page 100]) consider how best to add additional memory to your 8051-based application.

Please note that the material in this chapter is concerned primarily with devices using the Standard 8051 memory architecture. Some of the more recent 8051 devices, such as the Dallas 80C390, Analog Devices AD μ C812 and Philips 80C51MX, provide support for much larger amounts of external memory than was possible in the original 8051 device: we briefly consider such extended memory devices in this chapter, but – as each manufacturer has an individual solution – we do not attempt to cover them in detail.

ON-CHIP MEMORY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you create an 8051-based circuit that uses only internal memory?

Background

Some general background material on memory is presented in this section.

Direct vs. indirect addressing

You will often see the terms ‘indirect addressing’ and ‘direct addressing’ used in discussions about microcontroller memory. Although these terms often cause confusion, they are not difficult to understand.

You will recall that whatever language you write in (for example, C or assembly), your code must ultimately be translated into machine code instructions that can be executed by your chosen microcontroller. This set of possible machine instructions is defined by the hardware manufacturer. Through this process, even complex statements in a high-level language are eventually broken down into basic operations, such as ‘Copy this piece of data from one memory location to another’. These, in turn, are implemented by machine instructions that take the form ‘Move the contents of memory address X to register Y’.

There are essentially two ways in which such fundamental ‘Move’ instructions may be implemented in microcontrollers and microprocessors:

- Using **direct addressing**, the address of the memory location (that is, memory address X in the last example) is specifically given as part of the instruction.
- Using **indirect addressing**, the address of the memory location is not explicitly included as part of the instruction: instead the address (of another memory location or another register) that contains memory address X is included in the instruction.

Since the use of indirect addressing means that two steps are required to find the address of the required memory location it may appear to be slower than direct addressing. However, universal use of direct addressing in an 8-bit architecture (with a 16-bit address space) would mean that all ‘Move’ instructions would need to include two bytes of address information, and would therefore take more time to fetch from

memory. A compromise is thus often made in devices (including the 8051) where a small area of memory can be directly addressed and most other memory areas must be indirectly addressed.

Note that the distinction between direct and indirect addressing also has other uses. For example, within members of the 8051 family, there is an area of 'special function register' memory and another area of general purpose memory. Both blocks of memory are the same size (128 bytes) and both blocks share the same address range. On the surface, having two areas of memory with the same address makes no sense; however, in this case, there is no problem. One block of memory can only be accessed indirectly, the other block can only be accessed directly. As a result, when our compiler translates a particular 'C' statement, appropriate machine-level instructions are selected to ensure that the correct memory area is accessed: in most circumstances, this process is completely hidden from the programmer.

Types of memory

On the desktop, most designers and programmers can safely ignore the type of memory they are using. This is seldom the case in embedded environments and we therefore briefly review some of the different types of memory.

First, a short history lesson, to explain the roots of an important acronym. On early mainframe and desktop computer systems, long-term data storage was carried out using computer tapes. Reading or writing to the tape took varying amounts of time, depending whether it involved, for example, rewinding the entire tape or simply rewinding a couple of centimetres. In this context, new memory devices appeared that could be used to store data while the computer was running, but which lost these data when the power was removed. These read-write memory devices were referred to as 'random access memory' (RAM) devices, because – unlike tape-based systems – accessing any element of memory 'chosen at random' took the same amount of time.

Tapes have now largely disappeared, but the acronym RAM has not and is still used to refer to memory devices that can be both read from and written to. However, since RAM was first introduced, new forms of memory devices have appeared, including various forms of ROM (read-only memory). Since these ROM devices are also 'random access' in nature, the acronym RAM is now best translated as 'read-write memory'.

Dynamic RAM (DRAM)

Dynamic RAM is a read-write memory technology that uses a small capacitor to store information. As the capacitor will discharge quite rapidly, it must be frequently refreshed to maintain the required information: circuitry on the chip takes care of this refresh activity. Like most current forms of RAM, the information is lost when power is removed from the chip.

In general, dynamic RAM is simple and comparatively cheap.

Static RAM (SRAM)

Static RAM is a read-write memory technology that uses a form of electronic flip-flop to store the information. No refreshing is required, but the circuitry is more complex and costs can be several times that of the corresponding size of DRAM. However, access times may be a third those of DRAM.

Mask read-only memory (ROM)

A true read-only memory (ROM) would be useless. The most basic kind of practical ROM is, from the designer's perspective, read only: however, the manufacturer is able to write to the memory, at the time the chip is manufactured, according to a 'mask' provided by the company for which the chips are being produced. Such devices are therefore sometimes referred to as 'factory-programmed ROM' or 'mask ROM'. Factory or mask programming is not cheap and is not a low-volume option: as a result, mistakes can be very expensive and providing code for your first mask can be a character-building process. Access times are often slower than RAM: roughly 1.5 times that of DRAM.

It should be noted that mask ROMs retain their contents even in environments with high levels of electromagnetic activity. This behaviour is in contrast to some of the erasable devices in which there is a risk that data corruption may occur due, for example, to high UV levels (UV-EPROMs: see later in this section) or strong electrical fields (EEPROMs: see later in this section).

Many members of the 8051 family are available with on-board mask-programmable ROM.

Programmable read-only memory (PROM)

The name PROM sounds like a contradiction and it is. This is, in fact, a form of write-once, read-many (WORM) or 'one-time programmable' (OTP) memory. Basically, we use a PROM programmer to blow tiny 'fuses' in the device. Once blown, these fuses cannot be repaired; however, the devices themselves are cheap.

Many modern members of the 8051 family are available with OTP ROM.

UV-erasable programmable read-only memory (UV-EPROM)

Like PROMs, UV-EPROMs are programmed electrically. Unlike PROMs, they also have a quartz window which allows the memory to be erased by exposing the internals of the device to UV light. The erasure process can take several minutes and, after erasure, the quartz window will be covered with a UV-opaque label. This form of EPROM can withstand thousands of program / erase cycles.

More flexible than PROMs and once very common, UV-EPROMs now seem rather primitive compared with EEPROMs. They can be useful for prototyping but are prohibitively expensive for use in production.

Many older members of the 8051 family are available with on-board UV-EPROM.

Electrically erasable programmable read-only memory (EEPROM, E²PROM)

EEPROMs are a more user-friendly form of EPROM that can be both programmed and erased electrically. This does not mean they can simply be used in place of RAM for all purposes, not least because writing to the EEPROM is a very slow process and there is a limit to the number of write operations that may be performed.

Many members of the 8051 family are available with on-board EEPROM.

Flash ROM

Flash ROM is not only a relief from increasingly long and irrelevant acronyms, it is also the most civilized form of ROM currently available. As the name suggests, it can usually be programmed much more rapidly than an EEPROM. In addition, while many EEPROMs often require high (12V) programming voltages, Flash ROM devices can usually be programmed at standard (3V/5V) levels.

Members of the 8051 family are now available with on-board flash ROM.

Solution

The memory map of the 8051 family is illustrated in Figure 6.1.

In order to make best use of the internal memory, or to select an appropriate device for your application, you need to understand the meaning of the different memory areas. These are now discussed.

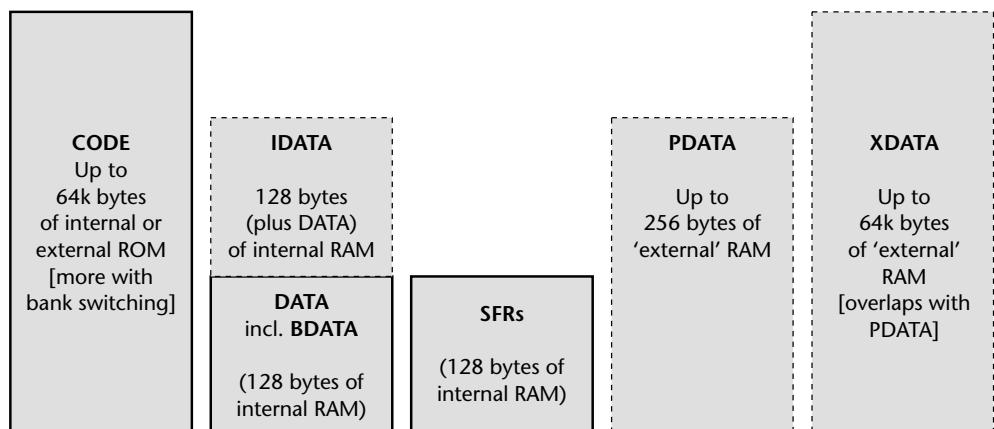


FIGURE 6.1 A schematic representation of the key memory areas on the 8051 family

[Note: that areas shown with dotted boundaries need not be present on all systems. Note also that, as the family of 8051 grows, some Extended 8051 devices have added additional memory areas.]

The CODE area

As the name suggests, the main rôle that CODE memory plays in your 8051 application is to store the program (executable) code. This happens automatically with all C compilers. Note that the code is executed ‘in place’ in ROM: it is **not** copied to RAM for execution.

In addition to code, it can be useful to store (read-only) data, such as lookup tables, in the CODE area. This is often an excellent idea: many 8051 devices have comparatively large amounts of ROM available (64 kbytes is no longer uncommon), but only small amounts of available RAM (usually no more than 4 kbytes; frequently no more than 256 bytes). Placing read-only data in ROM when possible usually makes good sense.

Using ROM for data tables can also make sense on performance grounds. For example, if your code requires calculation of sines or cosines, this can require a large number of CPU operations on most systems (typically more than 3000 CPU operations). If you are able to include the relevant results in an array (lookup table), this can reduce the CPU load by a factor of around 1,000.

Placing data in ROM is easy to do. For example, using the Keil C51 compiler and the `code` keyword, we can store a large array in ROM as follows:

```
int code CRC16_table[256] =  
{0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7, 0x8108,  
 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef, 0x1231, 0x210,  
 // etc...
```

The DATA, BDATA and IDATA areas

Up to 256 bytes of internal data memory are available depending on the particular 8051 device that is used. Access to internal data memory is generally very fast because it can be carried out using an 8-bit address.

The internal data area is split into three overlapping areas: DATA, IDATA and BDATA. We now discuss each of these areas.

Using the DATA area

The DATA area refers to the first 128 bytes of internal data memory. Variables stored here are accessed very quickly using direct addressing.

Using the DATA area is the default with the C51 compiler, if – as recommended by Keil – the small memory model is used. Alternatively, the `data` keyword may be used to explicitly specify that a variable should be stored in the DATA area. For example:

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

Using the IDATA area

The IDATA area refers to all 256 bytes of internal data memory (including the 128 bytes of the data area, with which it overlaps). Variables stored here are accessed less rapidly than DATA variables, using indirect addressing. The **idata** keyword may be used to explicitly state that variables are to be stored in the IDATA area, as follows:

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

Note, however, that the IDATA area is usually most useful as a stack area: in general, it is better to leave this area free for use by the compiler.

Using the BDATA area

The BDATA area overlaps with the DATA area. Specifically, it refers to 16 bytes of bit-addressable memory in the internal DATA area (addresses 0x0020 to 0x002F).

Use of the **bit**, **bdata** and **sbit** keywords allows you to make use of this area and to declare data types that can be accessed at the bit level. Consider the following examples:

```
// The definition of the unsigned character variable Bit_addressable
// - this can take values 0 to 255
unsigned char bdata Bit_addressable;

// The definition of the bit variable Flag
// - this can take values of 0 or 1
bit Flag;

// The declaration of the bit areas in the variable bit_addressable
// - Note use of the sbit keyword (*NOT* bit)
// - Note that this declaration does not use any memory
sbit Bit0 = Bit_addressable^0;
```

Special function register (SFR) memory

As shown in Figure 6.1, all 8051s provide up to 128 bytes of memory for special function registers (SFRs). SFRs are bit, byte or word-size (2-byte) registers that are used to control timers, counters, serial I/O, port I/O and various peripherals. The port SFRs are discussed in the pattern **PORT I/O** [page 162].

Note that – as briefly mentioned in ‘Background’ – the IDATA locations 128 to 255 and the SFR area share the same address space. However, memory locations in these two areas are accessed using different addressing modes. IDATA locations 128 to 255 are only indirectly addressable and the special function registers are only directly addressable.

External data memory

There may be up to 64 kbytes of external data memory. Compared to the internal memory areas, access to external memory is slow.

The external data memory can be divided into two areas. The XDATA area refers to any location within the (64-kbyte) data address space. The PDATA area represents the first 256 bytes of this address space. If programmed appropriately, access to PDATA variables is faster than access to those in the rest of the external data space.

Internal ‘external’ memory

Although it is mapped into the XDATA area, it should be noted that not all XDATA memory need be physically located outside the microcontroller. In fact, many modern devices include on-chip XDATA RAM. For example, the Dallas 83C520 includes 1 kbyte of such ‘XRAM’, while the Infineon C509 includes 3 kbytes of XRAM. These areas of RAM are used in the same way as external memory, as we discuss in the section ‘Controlling access to internal and external memory’ below.

Avoiding confusion between the various CODE and DATA areas

Note that many 8-bit microcontrollers have a single (64K) memory area, shared by code and data. In the case of the 8051, we have up to 64 kbytes of code and 64 kbytes of data available. Because both of these areas share the same address space, the chip (and compiler) need a means of accessing the correct area.

The main way of distinguishing between code and data access is using the /RD, /WR and /PSEN pins. The /RD and /WR pins are used only when accessing (external) data memory, while the /PSEN pin is used only when accessing (external) code memory.

As with direct and indirect addressing, this process is generally hidden from the programmer working in a high-level language.

Most members of the 8051 family operate in one of two modes determined at reset by the state of the ‘external access’ (/EA) pin. If /EA is held low, on-chip instruction (but not data) memory is disabled and the entire 64KB of instruction space is accessed externally. If /EA is held high, on-chip instruction memory is enabled. In these circumstances, external access to (code) memory will only occur if the program attempts to access an address beyond the range of the on-chip memory.

Forgetting to pull the /EA pin high is a very common error in single-chip designs created by developers new to the 8051 family.

Controlling access to internal and external memory

Note that, unlike CODE memory, the state of the /EA pin at reset does not affect on-chip data RAM which is always enabled and accessible. Another difference is related to the way the presence of on-chip RAM affects the external data memory space. For CPUs with up to 256 bytes of on-chip (IDATA) RAM, the full 64KB external data space

is also available. Where devices have on-chip XDATA memory, this will overlap with any external memory. Note that the address at which any on-chip XDATA memory is mapped into the address space varies between devices. Most chips map this memory at address 0x00, but some use different values.

Note that these comments do not apply to the Small 8051 devices, which do not support external memory and, therefore, do not require an /EA pin. In addition, some members of the 8051 family (notably the 8031 and derivatives, and some Extended 8051s) have no on-chip ROM, so always require the use of external (CODE) memory.

Different internal memory available on different 8051 devices

Examples of the various memory components on a range of 8051 devices is given in Table 6.1.

TABLE 6.1 Memory options available on a range of different 8051 family members

Device	Basic RAM (DATA and IDATA)	Extended RAM (XDATA)	ROM (CODE)	Comments
Analog Devices AD μ C812	256 × 8-bit	640 × 8-bit data Flash EEPROM	8K × 8-bit flash EEPROM	16M × 8-bit external data address space; 64K × 8-bit external program address space
Atmel 89C1051	128 × 8-bit	0	1K × 8-bit flash EEPROM	
Atmel 89C2051	128 × 8-bit	0	2K × 8-bit flash EEPROM	
Atmel 89C4051	128 × 8-bit	0	4K × 8-bit flash EEPROM	
Atmel 89S53	256 × 8-bit	0	12K × 8-bit flash EEPROM	
Dallas 83C520	256 × 8-bit	1K × 8-bit	16K × 8-bit OTP EPROM	
Dallas 87C520	256 × 8-bit	1K × 8-bit	16K × 8-bit mask ROM	
Dallas 80C390	256 × 8-bit	4K × 8-bit	0	4M × 8-bit external data address space; 4M × 8-bit external program address space
Infineon C501-1E	256 × 8-bit	0	8K × 8-bit OTP EPROM	

TABLE 6.1 Continued

Device	Basic RAM (DATA and IDATA)	Extended RAM (XDATA)	ROM (CODE)	Comments
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 EPROM	
Infineon C509-L	256 × 8-bit	3K × 8-bit	0	
Infineon C515C-8E	256 × 8-bit	2K × 8-bit	64K × 8-bit OTP EPROM	
Intel 8031	128 × 8-bit	0	0	
Intel 8032	256 × 8-bit	0	0	
Intel 87C51FA	256 × 8-bit	0	8K × 8-bit UV-EPROM	
Philips 80C51MX	256 × 8-bit	?	?	8M × 8-bit external data address space; 8M × 8-bit external program address space
Philips 83C751	64 × 8-bit	0	2K × 8-bit mask ROM	
Philips 87C751	64 × 8-bit	0	2K × 8-bit UV-EPROM	
Philips 87LPC764	128 × 8-bit	0	4K × 8-bit EEPROM	

[Note that the recently introduced 80C390 supports external code and data areas of up to 4 Mbyte (each area). Note also that the 80C51MX data are based on preliminary data released by Philips prior to the first chip release.]

Hardware resource implications

On-chip memory is a net provider of hardware (memory) resources.

Reliability and safety implications

Suppose that we have two identical applications, using (almost) identical software, but one using internal memory (only), and one using external memory (for code and/or data). Everything else being equal, it is likely that the ‘internal’ application will prove more reliable. This is partly because the use of internal memory reduces the opportunities for wiring or design errors, partly because of the reduction in external wires (‘aerials’) makes the system less vulnerable to EMI, and partly because (in the external version) each of the soldered joints has a risk of failure in the presence of vibration and/or high humidity.

In addition, the ALE pin can be a source of EMI. This pin is required for external memory access but its operation can (in some devices) be disabled, if internal memory is being used. This can reduce the likelihood that your application will induce an error in some other part of the application.

As (in almost all cases) the ‘internal’ solution will also be both cheaper to produce and physically smaller, the message is clear: use internal memory if at all possible.

Portability

In general, the most portable 8051 code assumes only the presence of a small amount of CODE memory (some kind of ROM, say 4 kbytes) and 128 bytes of RAM. This combination is available even in most of the smallest 8051s.

Further memory (typically 256 bytes of RAM) will be available in many modern 8051s. Some modern devices also have on-chip XRAM. However, the more of these facilities you use, the less easy it will be to port your code to another microcontroller in the 8051 family.

Overall strengths and weaknesses

Use of internal memory (in place of external memory) can have the following implications:

- ☺ Lower application cost.
- ☺ Increased hardware reliability.
- ☺ Reduced EM emissions (where you are able to disable ALE activity).
- ☹ In most cases, the available data memory will be restricted.

Related patterns and alternative solutions

See OFF-CHIP DATA MEMORY [page 94] and OFF-CHIP CODE MEMORY [page 100].

Example: Internal memory on the Philips 8XC552

As an example of the memory options available, we will consider the Philips 8XC552.

The 8XC552 contains 8 kbytes of on-chip program memory which can be extended to 64 kbytes through the use of external memory. When the EA pin is held high, the 8XC552 fetches instructions from internal ROM unless the address exceeds 1FFFH. Locations 2000H to FFFFH are fetched from external program memory. When the EA pin is held low, all instruction fetches are from external memory. ROM locations 0003H to 0073H are used by interrupt service routines.

The internal data memory is divided into three sections: the lower 128 bytes of RAM, the upper 128 bytes of RAM and the 128-byte special function register areas. The lower 128 bytes of RAM are directly and indirectly addressable. While RAM locations 128 to 255 and the special function register area share the same address space, they are accessed through different addressing modes. RAM locations 128 to 255 are only indirectly addressable and the special function registers are only directly addressable. All other aspects of the internal RAM are identical to the 8051. The stack may be located anywhere in the internal RAM by loading the 8-bit stack pointer. Stack depth is 256 bytes maximum.

The special function registers (directly addressable only) contain all the 8XC552 registers except the program counter and the four register banks. Most of the 56 special function registers are used to control the on-chip peripheral hardware. Other registers include arithmetic registers (ACC, B, PSW), stack pointer (SP) and data pointer registers (DHP, DPL). Sixteen of the SFRs contain 128 directly addressable bit locations.

Example: Comparing speed of access to different memory areas

Typical access times for data stored in the various memory areas (measured in instruction cycles) are as follows:

- Access to **DATA** area takes one cycle
[direct access]
- Access to **IDATA** area takes two cycles
[8-bit copy of address to register (1 cycle), then 1-cycle move]
- Access to **PDATA** area takes three cycles
[8-bit copy of address to register (1 cycle), then 2-cycle move instruction]
- Access to **XDATA** area takes four cycles
[16-bit copy of address to register (2 cycles), 2-cycle move instruction]
- Access to **CODE** area takes four cycles
[16-bit copy of address to register (2 cycles), 2-cycle move instruction]

Although these figures are typical, it is difficult to predict precisely how access times to variables in different memory areas will compare, partly because the results vary depending on different compiler options used.

Example: Making use of internal XRAM memory on the Infineon C515C

The Infineon C515C is a powerful (extended) 8051 device that we have used in a range of different applications, particularly because of its good performance and on-chip CAN component.

In addition to the standard (256 bytes) of IDATA RAM, the C515C has 2 kbytes of on-chip XRAM.

To use this memory, you need to be aware that (unlike the Dallas 520, for example), this memory is not mapped from address 0x0000. Instead, the starting address of this memory is 0xF800.

If you are using the Keil compiler, you need to provide this information to the linker, via the 'Size / Location' menu item. You should enter the address (start of XDATA) as '0F800': omitting the initial '0' causes problems with current versions of the compiler.

Further reading

OFF-CHIP DATA MEMORY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you add up to 64 kbytes of external RAM to your Standard 8051 microcontroller?

Background

To add external data memory to an 8051 we first need to understand the memory interface. An overview of this interface is shown in Figure 6.2 and will be referred to in the discussions that follow.

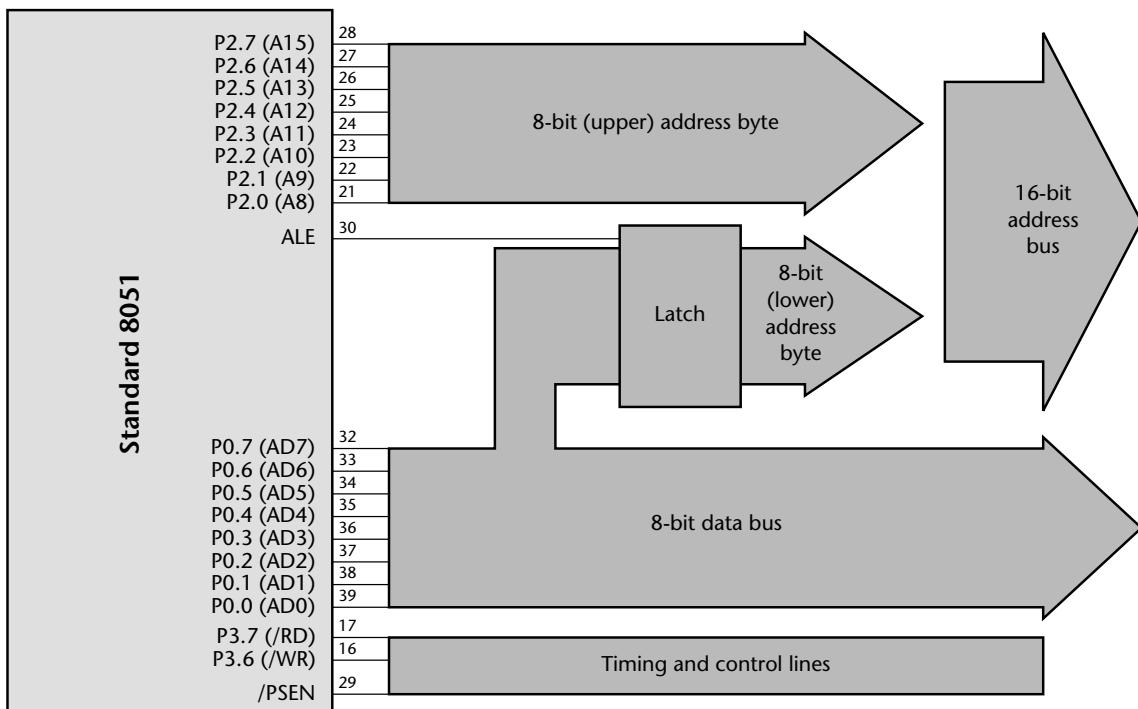


FIGURE 6.2 The 8051 memory interface

The address bus (P0.0–P0.7)

For both program and data access, Port 0 is used as a multiplexed address/data bus that outputs the low-order address bits (A0–A7) and inputs/outputs the 8-bit data (D0–D7).

The data bus (P2.0–P2.7)

For all external program accesses, P2 outputs the high-order address bits (A8–A15). The same is true for external data accesses with 16-bit addresses.

Note one special case: Systems that use on-chip code memory and only 256 bytes of external data memory are free to use P2 for general purpose I/O.

ALE (address latch enable)

ALE is used to demultiplex the AD0–7 bus. At the beginning of the external cycle ALE is high and the CPU emits A0–A7 which should be externally latched when ALE goes low.

Note that on most 8051-based systems, ALE is always active, even during internal program and data accesses: however, on some more modern 8051 designs, it is possible to disable ALE activity if external memory access is not required: this can help to reduce EM emissions.

Note also that, where external memory access is not required, ALE may be treated as a continuous clock that runs at one-sixth the oscillator frequency. This output can be used, for example, to control timing in external circuits.

PSEN (program store enable)

PSEN is the read strobe for external instruction (code memory) access. Unlike ALE, PSEN is not asserted during internal accesses. We consider the use of PSEN in the pattern **OFF-CHIP CODE MEMORY** [page 100].

RD (data read)

RD is the read strobe for external data access and (like PSEN) is not asserted during internal accesses.

WR (data write)

WR is the write strobe for external data access and (like PSEN and RD) is not asserted during internal accesses.

Solution

With a basic understanding of the memory interface (see ‘Background’), adding external data memory to an 8051 microcontroller is easy to do, provided some care is taken in the choice of components (see Figure 6.3).

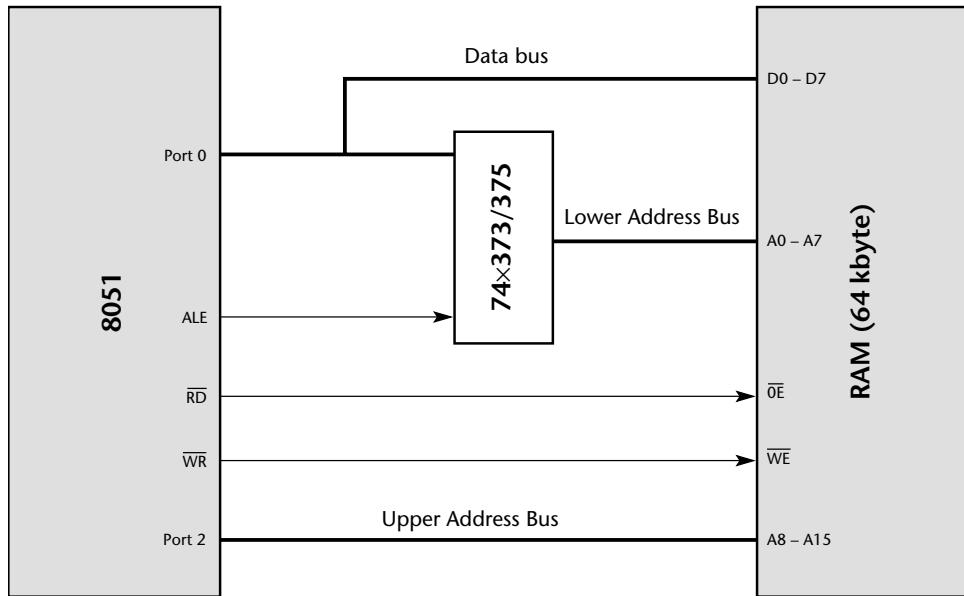


FIGURE 6.3 Adding external data (RAM) memory to an 8051 microcontroller

[Note that, if the microcontroller is not a CMOS device and the memory devices are CMOS devices, you will require pull-up resistors (not shown) on Port 0. This is most easily achieved using a DIL (or similar) 10K resistor pack.]

Note that the 74x373 and the 74x375 are functionally identical, but have different pin arrangements. The '375 arrangement is usually easier to wire up. Both latches are available in different speed ratings: inevitably, the cost increases with the speed.

Recommended latch and (RAM) memory combinations for a wide range of clock speeds are given in Table 6.2. These are taken from Dallas Application Note 89; however, latch and memory combinations which will operate with these Dallas devices will also work with most other (generally slower) 8051 devices.

Hardware resource implications

Use of external memory has **major** resource implications: it requires the use of two ports (P0 and P2), plus two pins (P3.6, P3.7) on Port 3. This reduces the number of available port pins from 32 to 14. See the following for additional comments on this.

Reliability and safety implications

As discussed in **ON-CHIP MEMORY** [page 82], the addition of external memory can reduce the reliability of your application: use an on-chip solution where possible.

However, as many 8051s simply do not have enough RAM to support larger applications and you will be forced to use off-chip RAM in some applications. When you

TABLE 6.2 Recommended latch and (RAM) memory combinations for a wide range of clock speeds (adapted from Dallas Application Note 89)

Clock frequency (MHz)	Recommended latch	Recommended memory speed
33.0	74F373/375	55 ns
29.5	74F373/375	55 ns
25.0	74F373/375	80 ns
22.1	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

do use off-chip memory (code or data), please note that one of the most common errors made by inexperienced 8051 developers is to continue to use P0, P2 or P3 as normal I/O ports when using external memory. This cannot be done (for reasons discussed in ‘Hardware resource implications’).

In short: if you use external memory, you cannot safely use P0 and P2 for any other purpose, and you must also take care when writing to Port 3.

For example, any statement similar to this:

```
P3 = AD_data;
```

is potentially catastrophic.

Instead, make use of sbit variables to ensure you only write to ‘safe’ port pins: see **PORT I/O** [page 174] for further details.

Portability

Hardware designs for external memory are generally portable. However, if you upgrade from a ‘slow’ to a ‘fast’ processor (or simply increase the crystal frequency) you need to make sure that the external latch and memory components are sufficiently fast: refer to Table 6.2 for suggestions.

The only way to make your external access design as portable as possible is always to use the fastest external components available. This approach has cost implications, but, if you are able to produce 100,000 copies of one, generic (fast) board rather than smaller numbers of ‘slow’, ‘medium’ and ‘fast’ versions, you may find the ‘one size fits all’ solution to be both a reliable and cost-effective solution.

Overall strengths and weaknesses

- ☺ Many 8051s simply do not have enough RAM to support larger applications: this pattern solves that problem.
- ☹ As discussed in **ON-CHIP MEMORY** [page 82], the addition of external memory can reduce the reliability of your application: use an on-chip solution where possible.

Related patterns and alternative solutions

See **ON-CHIP MEMORY** [page 82] for a discussion on the use of on-chip memory.

Example: Using external RAM and internal XRAM on the C509

The Infineon C509 microcontrollers have 3 kbytes of internal XRAM. By default, XRAM is disabled when the device is reset and the full 64 kbytes of external data memory are accessible.

However, if XRAM is enabled, then the internal XRAM and external RAM memory areas overlap: specifically, in the case of the C509, the XRAM is mapped to the upper 3 kbytes of data memory, as illustrated in Figure 6.4.

The internal XRAM is enabled via the SYSCON1 SFR. Specifically, the PRGEN bit must be set to 0 and the SWAP bit must be set to 1.

Example: Using external RAM and internal SRAM on the Dallas 8XC520

The Dallas 8XC520 microcontrollers have 1 kbyte of internal ‘SRAM’. By default, this SRAM is disabled when the device is reset and the full 64 kbytes of external data memory are accessible. However, if SRAM is enabled, then the internal XRAM and external RAM memory areas overlap: specifically, in the case of the 8XC520, the SRAM is mapped to the lowest 1 kbyte of data memory, as illustrated in Figure 6.5.

Further reading

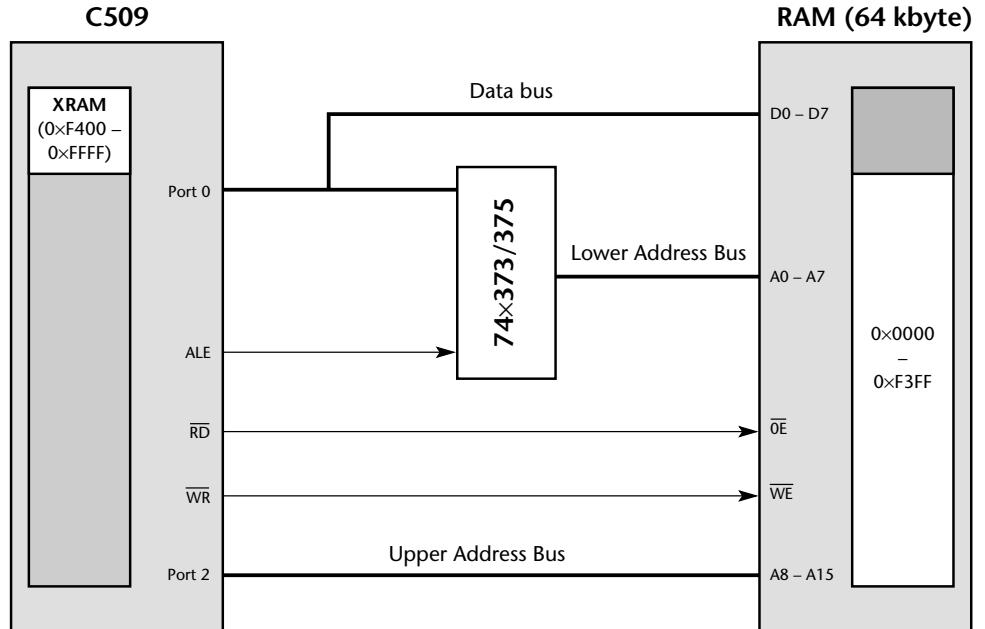


FIGURE 6.4 Adding external RAM to the Infineon C509 microcontroller

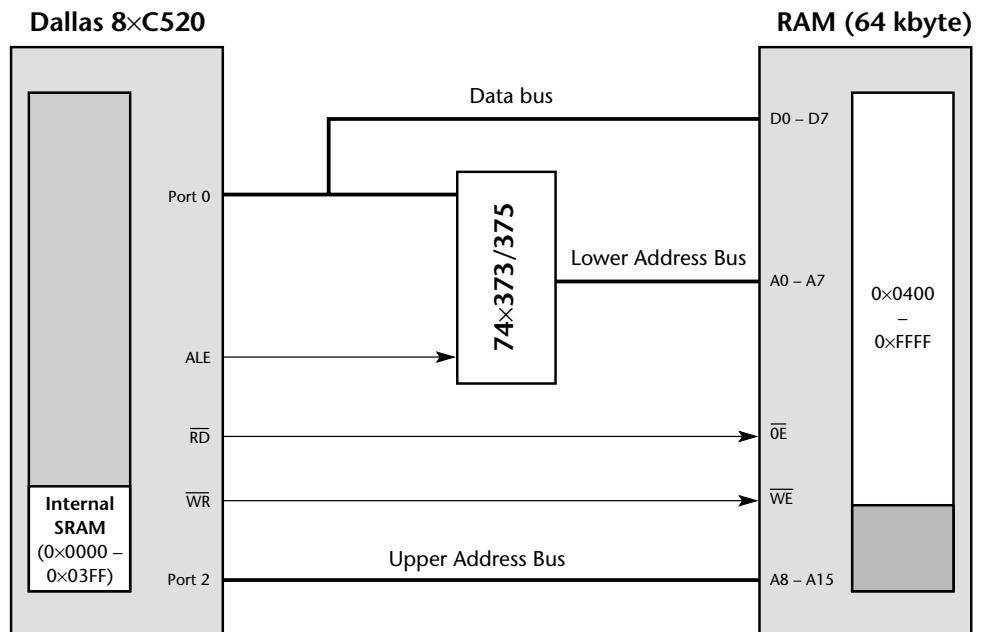


FIGURE 6.5 Adding external RAM to the Dallas 8XC520 microcontroller

OFF-CHIP CODE MEMORY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you add up to 64 kbytes of external code memory (usually ROM) to your 8051 microcontroller?

Background

See **OFF-CHIP DATA MEMORY** [page 94] for background information.

Solution

If you have decided that you need to use external code memory doing so is straightforward. Figure 6.6 illustrates the basic circuit.

The recommended latch and (ROM) memory combinations for a wide range of clock speeds are given in Table 6.3. These are taken from Dallas Application Note 89. As we noted in **OFF-CHIP DATA MEMORY** [page 94], latch and memory combinations which will operate with these Dallas devices will also work with most other (generally slower) 8051 devices.

TABLE 6.3 Recommended latch and (ROM) memory combinations for a range of clock speeds (adapted from Dallas Application Note 89)

Clock frequency (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

Clock frequency (MHz)	74F373/375	74AC373/375	74HC373/375
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

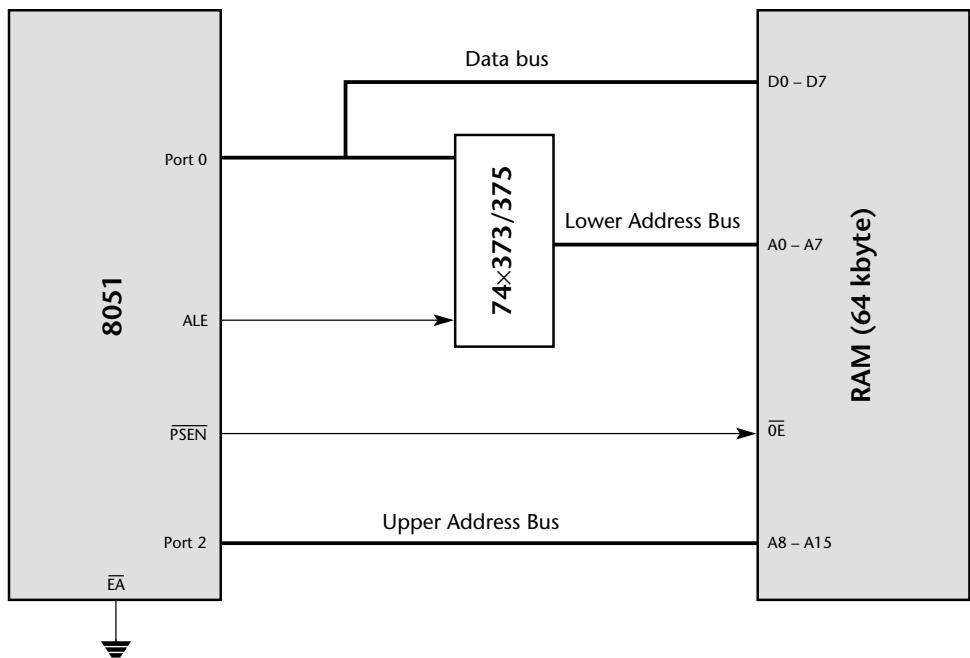


FIGURE 6.6 Adding external ROM memory

[Note that, if the microcontroller is not a CMOS device and the memory devices are CMOS devices, you will require pull-up resistors (not shown) on Port 0. This is most easily achieved using a DIL (or similar) 10K resistor pack. Note also the control of the EA pin, to force the microcontroller to retrieve instructions from external memory.]

Hardware resource implications

As discussed in **OFF-CHIP DATA MEMORY** [page 94], use of external memory has **major** resource implications: it requires the use of two ports (P0 and P2), plus two pins (P3.6, P3.7) on Port 3. This reduces the number of available port pins from 32 to 14. See the following for additional comments on this.

Reliability and safety implications

As discussed in **ON-CHIP MEMORY** [page 82], the addition of external memory can reduce the reliability of your application: use an on-chip solution where possible.

In many cases, adding external ROM can be avoided if you use an appropriate 8051-device with on-chip ROM: see ‘Related patterns and alternative solutions’ for further details.

Portability

Hardware designs for external memory are generally portable. However, if you upgrade from a ‘slow’ to a ‘fast’ processor (or simply increase the crystal frequency) you need to make sure that the external latch and memory components are sufficiently fast: refer to Table 6.3 for suggestions.

The only way to make your external access design as portable as possible is always to use the fastest external components available. This approach has cost implications, but, if you are able to produce 100,000 copies of one, generic (fast) board rather than smaller numbers of ‘slow’, ‘medium’ and ‘fast’ versions, you may find the ‘one size fits all’ solution to be both a reliable and cost-effective solution.

Overall strengths and weaknesses

(If at all possible, use of internal code memory is a better option!

Related patterns and alternative solutions

Before you add ROM to your 8051-based system, think carefully. There are many 8051 devices with large amounts of on-chip code memory available. As an example, Table 6.4 shows a selection of recent Philips 8051 devices.

As is clear from the table, 8051 devices are now available with large amounts of on-chip ROM (and good performance levels). Of course, similar devices are available from other manufacturers.

There are, perhaps, three main reasons for adding external code memory:

- You need particular on-chip hardware components (e.g. CAN bus, ADC, hardware maths) and cannot find an 8051 device that has both sufficient ROM and the required peripheral(s).

TABLE 6.4 An example of the available (internal) ROM on various recent Philips 8051 devices

Part#	Flash size	Comment
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 (12 clock optional)
89CRC2	32K	6-clock machine cycle (12 clock optional)
89CRD2	64K	6-clock machine cycle (12 clock optional)

[Note that similar devices are available from other manufacturers, but the Philips range is particularly comprehensive.]

- You require external RAM. Having therefore already created much of the required external memory interface, you have decided that it makes good economic sense to add external ROM too, rather than using a device with on-chip ROM.
- You need to be able to update the code that your system will execute. For example, if your system is running on a comparatively expensive (Extended) 8051 device, you may decide that it will be more cost effective to replace a small ROM chip when code updates are required, rather than to replace the microcontroller itself.

Example: Adding ROM and RAM memory

Figure 6.7 illustrates how you can add both code and data memory to your system.

Example: Reducing the component count

If space is very tight and / or you are seeking to improve the reliability of your 8051-based system by keeping the number of connections to a minimum, you have two basic choices:

- Use a microcontroller with on-chip code memory.
- Use a memory (ROM) chip that does not require a latch.

Option 1 is not always available. For example, the powerful C509 microcontroller is very fast and has several useful features (including a hardware maths unit compatible with that in the 80C517). It does not, however, have on-chip ROM of any kind.

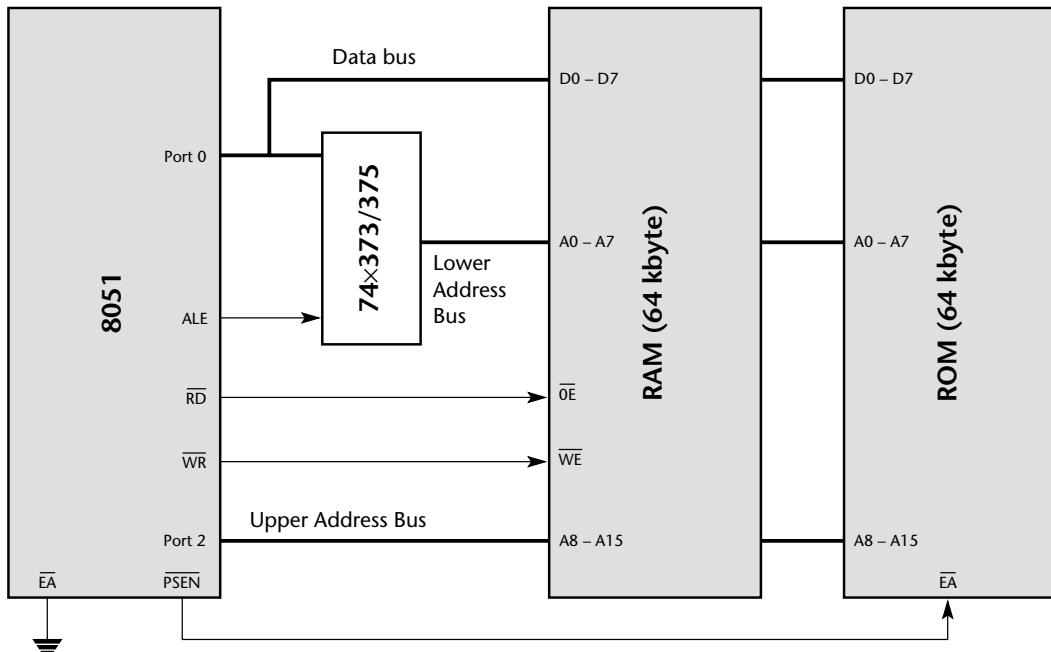


FIGURE 6.7 Adding external ROM and RAM memory to an 8051 design

To add external memory without requiring the use of a latch, consider using an Atmel AT27C520 (64 kbyte) EPROM. It has an internal latch, so no additional components are required.

Example: Adding more than 64 kbytes of code memory

With some recent exceptions (see Table 6.1), the various members of the 8051 family only directly support a maximum of 64 kbytes of external code memory. This may not be sufficient for larger programs or where large amounts of read-only variables need to be used. ‘Bank switching’ is one option.

The idea of a bank-switched memory arrangement is, **superficially**, very straightforward. In a simplified arrangement, we may have (for example) eight 64-kbyte banks of memory in the system. Access to memory locations within each block is controlled by the usual (16) address lines. To select between these eight memory blocks, we can use eight additional output pins on the microcontroller to select an appropriate memory bank, as required.

In practice, things become slightly more complicated.

First, consider interrupts. When an interrupt occurs, the microcontroller will jump to a particular address in the **current** bank. One way of ensuring that this address contains the relevant instructions is to duplicate your interrupt functions in all the banks (note that your compiler can usually do this automatically, as we will see later).

Similarly, library functions also need to be universally accessible and, therefore, (usually) in a common area.

The need for duplication can also arise with read-only variables stored in the code area. Unless you can ensure that these constants will only be called in a particular bank, they need to be placed in a common area.

Finally, the bank-switching code itself must be in a common area.

Having addressed these problems, you need to consider what happens when the chip is reset. At this time, all port outputs are set to 0xFFFF. If a port is being used for bank switching, the port outputs will control which bank of code your program begins to execute first. You need to ensure, mainly through the bank-switching hardware, that your program begins by executing code in the correct bank.

Health warning!

As these comments should begin to make clear, bank switching is complicated and prone to error. Even if you know what you are doing, people who subsequently have to maintain the system you create may not fully understand the consequences of any changes they make to the code.

If possible, avoid bank switching. If your application requires more than 64 kbytes of code memory, a much better solution is to use a Dallas 80C390, Analog Devices AD μ C812 or Philips 8051MX, all of which support more than 64 kbytes of code memory. Alternatively, consider one of the two ‘upgrade’ paths from the 8051 (viz, the 80251, the 8051XA) discussed in Chapter 3.

If you are determined to implement bank switching, you need, first, an appropriate address decoding scheme and, second, a means of implementing a common code area.

There are two basic ways of tackling the ‘common area’ problem. The simplest approach is simply to duplicate all the common code in every code bank. For example, Figure 6.8 shows the use of four 64-kbyte ROM chips, with a common area (around 32 kbyte) duplicated in each.

Alternatively, we can achieve a similar result by using five 32-kbyte memory devices, arranged with one common area and four upper code banks (Figure 6.9).

While superficially more efficient, the lower cost of large memory chips means that a solution based on one memory chip is most likely to prove cost effective. In addition, because most of the code examples presented in this book make comparatively little use of interrupts, or of library functions, the common area required is often small in size (typically a few kbytes) and the ‘duplicated common area’ solution is therefore less inefficient than it might initially appear.

For example, consider that we wish to add a 512-kbyte ROM and 64-kbyte RAM to a C509 microcontroller. Much of the required hardware is familiar: we have an external ROM device and an external RAM device, connected to the microcontroller through an 74HC573 latch. This is exactly the same arrangement seen previously in Figure 6.7.

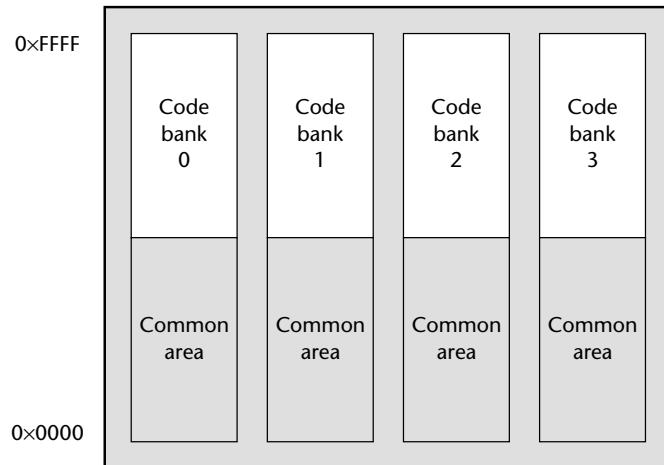


FIGURE 6.8 Bank switching with four 64-kbyte ROM chip

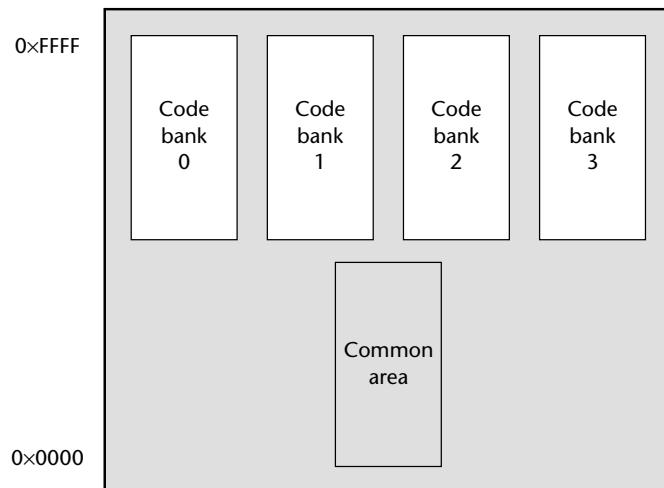


FIGURE 6.9 Bank switching with five 32-kbyte ROM chips

Where the hardware differs is first, of course, in the size of the code memory area and, second, the presence of the 74HC257. The 74HC257 is a (quad) 2-line to 1-line data multiplexer: it provides the necessary ‘glue logic’, to interface the port pins (lower nibble of Port 3 in this example: any available port may be used).

Details of the 74HC257 connection are given in Figures 6.10 and 6.11.

Note that the Keil compiler provides support for bank switching and will support the memory arrangement shown here: see the *Banker / Linker* manual for details.

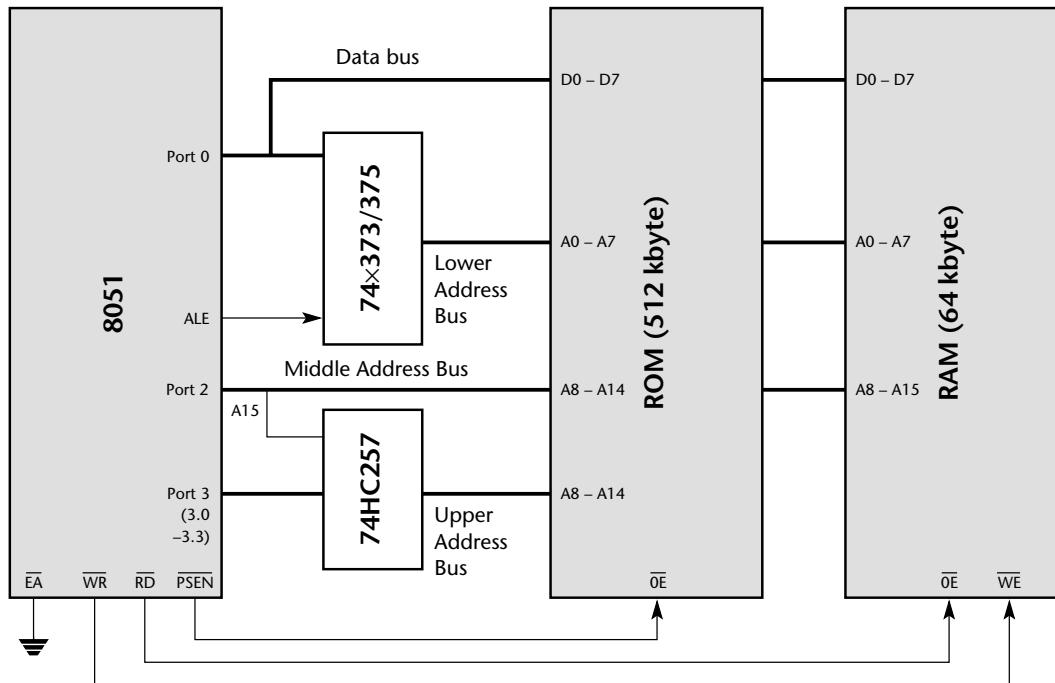


FIGURE 6.10 Adding a 512-kbyte ROM to an 8051 design

[The hardware in this example is adapted from Infineon Application Note AP0824.]

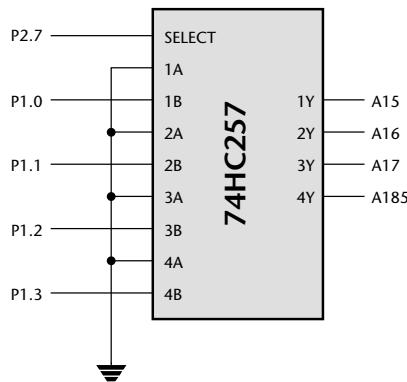


FIGURE 6.11 Details of the 74HC257 connection in Figure 6.10

Further reading

7

chapter

Driving DC Loads

Introduction

The port pins on a typical microcontroller can be set at values of either 0V or 5V (or, in a 3V system, 0V and 3V) under software control. Each pin can typically sink (or source) a current of around 10 mA. In this chapter, we are concerned with hardware designs that will allow us to control a range of low- and high-power, direct current (DC) loads via these pins.

With care, the port may be used directly to drive low-power DC loads, such as LEDs (see **NAKED LED** [page 110]) or, for example, small warning buzzers (see **NAKED LOAD** [page 115]). However, while a limited number of such loads may be connected directly to the port, connecting, say, eight LEDs will generally exceed the total port or microcontroller capacity. In these circumstances, use of an IC-based buffer circuit can be a cost-effective solution: see **IC BUFFER** [page 118]. Indeed, even with small loads, the reliability of the application may be improved through the use of such a buffer.

Of course, many output devices require a higher voltage and a far greater power output than the naked ports can provide. For example, to drive even a small DC motor may require up to 1A of current at 12V. To control such devices, we need to provide appropriate driver (or interface) circuit to convert the microcontroller output to an appropriate level. We will consider three ways of driving such loads in this chapter:

- Using bipolar-junction transistors (see **BJT DRIVER** [page 124])
- Using a driver IC (see **IC DRIVER** [page 134])
- Using ‘metal oxide silicon field effect transistors’ (see **MOSFET DRIVER** [page 139])

Please note that in this chapter our concern will be with hardware details of the interface only. In **PORT I/O** [page 174] we consider software suitable for controlling such hardware.

NAKED LED

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

What is the cheapest way of driving a small number of LEDs with a microcontroller?

Background

Even in the most basic application, the presence of a constant red or green output from a light-emitting diode (LED) is reassuring, implying that the application is at least powered. In other applications requiring numerical outputs, LEDs are grouped into seven- or eight-segment combinations¹² to display, for example, the current time, the voltage or the number of calls received by a telephone answering machine ('voice mail') system. Overall, LEDs are the most widely used components in user interfaces for embedded systems.

Given the name it is hardly surprising that, electrically, an individual LED operates as a diode. LEDs have a forward voltage drop of about 2V, and typically require a current of around 5–15 mA for a bright display (Figure 7.1). Note that the forward voltage required to 'switch on' a conventional (silicon) diode is around 0.7 V: the difference arises because LEDs are generally manufactured from gallium arsenide phosphide (see Horowitz and Hill, 1989, for further details).

Solution

The focus in this pattern is on low-cost techniques for driving LEDs.

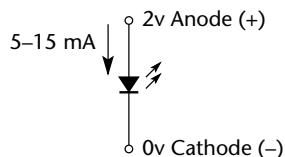


Figure 7.1 Lighting a single LED

12. Multi-segment LEDs are considered in **MX LED DISPLAY** [page 450].

Driving single LEDs

It is possible to drive small single LEDs directly from the port pins, as illustrated in Figure 7.2. Note that we usually need a resistor in series with the LED, between the supply and the port pin. This resistor is required to limit the current flow to the port when the LED is ‘switched on’.

To understand why this resistor is necessary, you need to remember that – as we discussed in ‘Background’ – the voltage drop across the LED will be around 2V. Assuming V_{cc} is 5V, then the voltage drop across the resistor and diode, when the port pin is at 0V, will be around 5V. Thus, the voltage drop across the resistor needs to be 3V. If there is no resistor, then we need to drop 3V across a stretch of wire: this can cause a very strong current to flow (limited only by the supply capacity), which will almost instantly destroy the port, the LED and, possibly, the whole microcontroller.

The equation in Figure 7.2 (essentially Ohm’s law) shows how to calculate the required resistor (R) value. Typical values of the required parameters are as follows:

- Supply voltage, V_{cc} = 5V
- LED forward voltage, V_{diode} = 2V
- Required diode current, I_{diode} = 10 mA (note that the data sheet for your chosen LED will provide this information)

This gives a required R value of 300Ω.

Use of pull-up resistors

Throughout this chapter, we will present examples of driver circuits like that shown in Figure 7.2.

This circuit will only work on port pins where the microcontroller has an internal pull-up resistor: this applies to most ports on the 8051 family, with the exception of Port 0. In addition, some other members of the 8051 family – notably the Atmel 89Cx051 devices – have small numbers of pins without internal pull-ups.

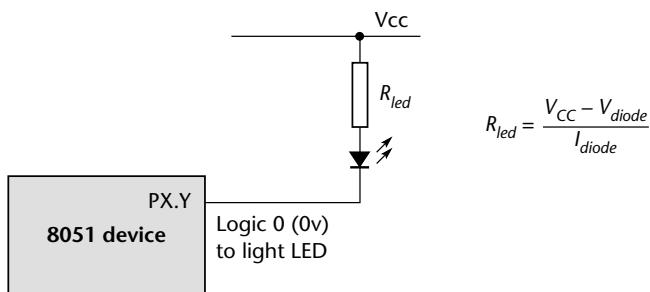


Figure 7.2 Connecting a single LED directly to a microcomputer port

[Note: When calculating the required value of R_{led}, the resistance values are in ohms, voltages in volts and current in amps.]

To adapt circuits such as that shown in Figure 7.2 for use on pins without internal pull-up resistors is straightforward: you simply need to add an external pull-up resistor, as illustrated in Figure 7.3. The value of the pull-up resistor should be between 1K and 10K. This requirement applies to all of the examples in this book.

In the unlikely event that you do not know whether a drive circuit will be used on a port with pull-up resistors, the best solution is to include 10K resistors in your circuit. If the port pin used already contains pull-ups, the extra resistor will have no discernible impact on the operation of the circuit.

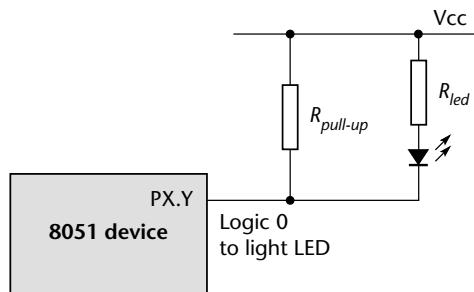


FIGURE 7.3 Using a pull-up resistor

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety implications

There are several reliability implications involved in the use of this pattern.

Connecting up LEDs

If you connect ordinary LEDs to a port, do not omit the resistor! The resulting high current flows may not cause the system to fail immediately, but will greatly reduce the life of the port pin and, potentially, the whole processor.

Note: There are '5V' LEDs available, at higher cost: these include series resistors. Provided they have suitable current requirements, they may be safely connected directly to the port pins.

Should you use a buffer?

Where more than two LEDs are connected to a single port, buffering is almost always essential, because – while the limit for an individual port pin may be (say) 10 mA – the port as a whole may have a limit of 20 mA or less. This issue is discussed in **IC BUFFER** [page 118].

Use of LEDs as warning devices

LEDs (particularly flashing LEDs) are frequently used as warning devices. Bear in mind that in bright sunlight, such warnings will be barely visible and that blind or partially sighted people will never be able to see them. Adding an additional or alternative audible output may be appropriate in some systems.

General use of LEDs

LEDs consume large amounts of power (compared, for example, to liquid crystal displays) and need to be used with care in many battery-powered designs.

Portability

The circuits presented here can be used with almost all microcontrollers and micro-processors. Please note, however, that most of the circuits we present for driving DC and AC loads involve some form of current 'sink', as in Figure 7.2. Here, the current flows 'in to' the processor port. This is despite the fact that most microcontroller ports, manufactured using some form of MOS technology, can also 'source' current, so that the load could be arranged as illustrated in Figure 7.4.

However, some of the other circuits used as buffers or drivers may use different manufacturing processes, including TTL technology. TTL devices can sink current in much the same way as MOS devices, but are poor current sources. As a result, hardware designs based on current sinking are generally more portable (across different logic families) than designs based on current sourcing and, for this reason, will be used throughout this pattern collection.

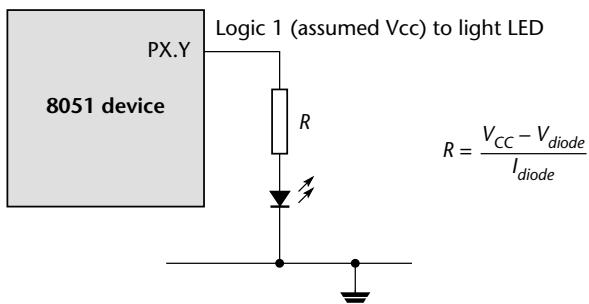


FIGURE 7.4 Connecting a single LED directly to a processor port

[Note: Here the port acts as a current source. See text for details and for a discussion of the drawbacks of this approach.]

Overall strengths and weaknesses

- ☺ This pattern allows small numbers of LEDs to be driven from a microcontroller port with a minimum of external hardware.
- ☺ This is a low-cost solution.
- ☺ Only applicable for small numbers of LEDs (typically two per port), otherwise a buffer will be required: see **IC BUFFER** [page 118].

Related patterns and alternative solutions

See **PORT I/O** [page 174] for software details.

See the remaining patterns in this chapter for techniques suitable for driving higher powered loads.

Example: Using low-current LEDs

To emphasize that all rules can be broken, we will consider in this example how you can connect eight ‘naked’ LEDs to the port of an 8051 microcontroller without using a buffer (see Figure 7.5).

Of course, for the reasons outlined in this pattern, it is not possible to connect up ‘normal’ (~10 mA) LEDs in this way without exceeding the current capacity of the port: however, if we use low-current (2 mA) LEDs, then – with most 8051 microcontrollers – this is possible.

Here, the required resistor values are calculated as follows:

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

Note that you get ‘nothing for nothing’ in this world: the 2 mA LEDs will not be as bright as 10 mA or 20 mA LEDs.

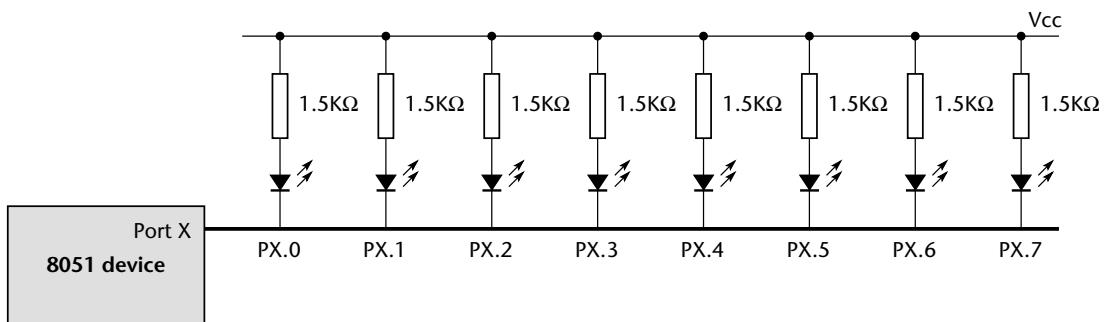


FIGURE 7.5 Using low-current ‘naked’ LEDs

Further reading

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

NAKED LOAD

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you connect a piece of low-voltage ($\leq 5V$), low-power ($\leq 100 \text{ mW}$) DC equipment to one of the port pins on your (8051-based) application?

Background

See **NAKED LED** [page 110] for general background material.

Solution

In **NAKED LED** [page 110] we considered how to connect an LED to a port pin of an 8051 microcontroller. In this pattern we consider a more general situation where we wish to control any small DC ‘load’ (Figure 7.6).

This shows a general (unspecified) load connected to the port pin.

The main difference between the general load and the LED is that the required voltage drop across the load will (typically) be 5V (in a 5V system: 3V in a 3V system),

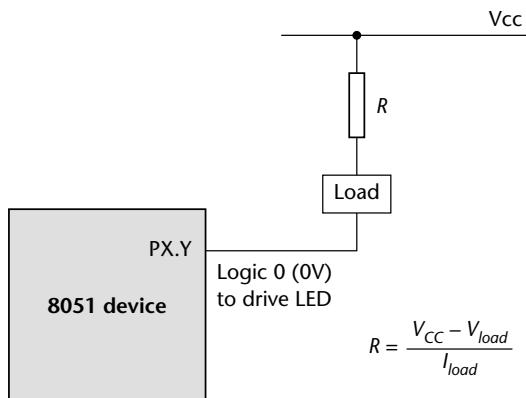


FIGURE 7.6 Driving a low-power load without using a buffer

rather than the 2V seen in the case of LEDs. In these circumstances, the resistor may be omitted, since the required resistance value will be:

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

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety implications

IC BUFFER [page 107] discusses techniques for buffering LEDs and other small loads. This can increase the reliability of some applications.

Portability

All microcontrollers can control small loads in this way.

Overall strengths and weaknesses

- ☺ The techniques discussed in this pattern allow small loads to be driven from a microcontroller port with a minimum of external hardware.
- ☺ This is a low-cost solution.
- ☹ Only applicable for small loads (and small numbers of very small loads), otherwise a buffer will be required: see **IC BUFFER** [page 118].

Related patterns and alternative solutions

See **PART I/O** [page 174] for software details.

See the pattern **IC BUFFER** [page 118] for techniques suitable for driving two or more small loads from a single microcontroller.

See the patterns **BJT DRIVER** [page 124], **IC DRIVER** [page 134] and **MOSFET DRIVER** [page 139] for techniques suitable for driving higher powered DC loads.

Example: Buzzer

A range of piezoelectric buzzers are available that generate very loud (~70 dB) tones at microcontroller port voltages (they will operate from 3V to 12V) and currents (they require around 3 mA). These make excellent warning devices, even in battery-powered applications (see Figure 7.7).

As we discussed in ‘Solution’, this 5V load requires no series resistor.

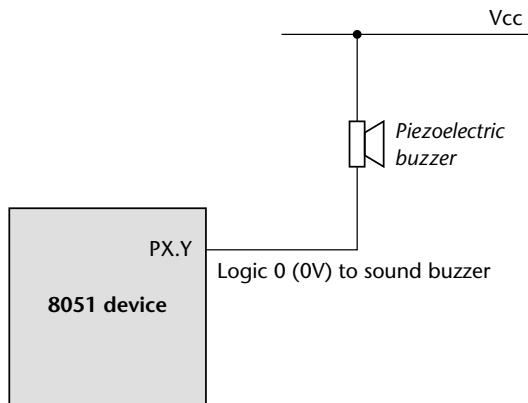


FIGURE 7.7 Connecting a buzzer directly to a microcontroller port

Further reading

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

IC BUFFER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you safely control one or more small, low-power DC loads from a single microcontroller port?

Background

Suppose we try to control eight 10 mA LEDs from a single port using the techniques discussed in **NAKED LED** [page 110]. Figure 7.8 illustrates one way in which we might try to achieve this.

In almost all circumstances, this approach will fail. As discussed in **NAKED LED**, the various members of the 8051 family can typically sink or source around 10 mA of current per port pin. This figure does not, however, give the whole story. Take one family member as an example. The Atmel 89C52 is a modern ‘standard’ 8051 device (40 pins, 4 ports), which is used in various examples throughout this book. It can sink up to 10 mA per port pin. However, **in total**, P0 can sink only 26 mA (over all port pins) and P1, P2 and P3 can only each sink a total of 15 mA. Overall, at most, the whole chip can sink only 71 mA.

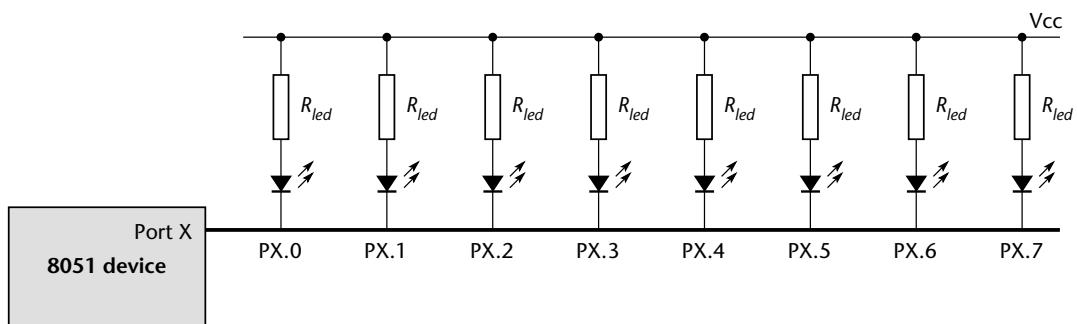


FIGURE 7.8 Attempting to drive eight LEDs directly from a microcontroller port

[Note: IN MOST CASES, UNLESS YOU USE LOW-CURRENT (~1 mA) LEDs, THIS WILL DESTROY THE PORT (and often the microcontroller too).]

Although the various 8051 family members vary, these figures are representative of those found in many devices. As a result, circuits like that shown in Figure 7.8 – which require a total current flow of around 80 mA for typical LEDs – cannot generally be used.

Solution

As discussed in ‘Background’, buffers can be needed if we need to drive multiple low-power loads from one microcontroller. Even where the ports can drive loads directly, we can both reduce the risk of damage to the ports and safeguard the application as a whole by using an IC buffer between the microcomputer and the load.

We consider, first, some of the many ICs that may be used as buffers and, second, the need for pull-up resistors at the buffer outputs.

Finding an IC

Various ICs can be used as buffers in this way. Suitable inverters are readily and very cheaply available: six are packaged together, for example, in the ubiquitous 74x04 (Figure 7.9).

An alternative to the 74x04 is the inverting (74x240) and non-inverting (74x241) buffers that come in packages of eight (see Figures 7.10 and 7.11). These are particularly useful when buffering a whole port.



FIGURE 7.9 Pinout of the 74x04 inverter and corresponding logic diagram (positive logic). (Reproduced courtesy of Texas Instruments)

[Note: that each chip contains six independent inverters.]

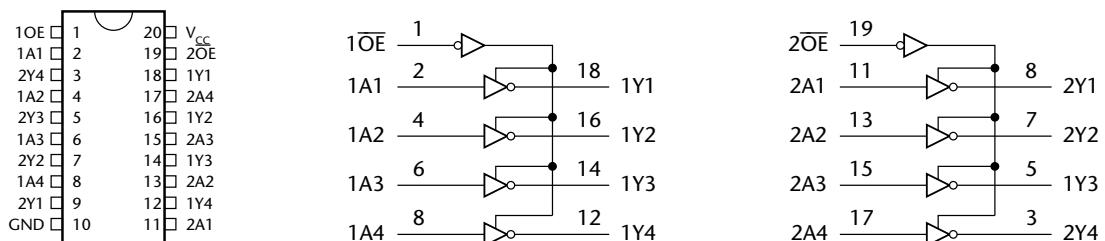


FIGURE 7.10 Pinout and logic diagram of the 74x240 inverting buffer (reproduced courtesy of Texas Instruments)

[Note: that each chip contains eight buffers, arranged in two groups of four (Group 1, Group 2). The buffers are all tri-state devices and, for use as a simple buffer, the gates must be enabled (in the case of Group 1) by applying a low (~0V) input to Gate 1 or (in the case of Group 2) Gate 2.]

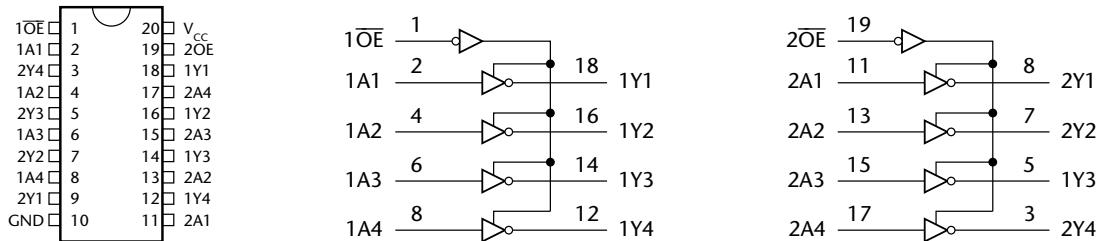


FIGURE 7.11 Pinouts of the 74x241 and 74x244 non-inverting buffers (reproduced courtesy of Texas Instruments)

[Note: that, like the 240, each chip contains eight buffers, arranged in two groups of 4 (Group 1, Group 2). The buffers are all tri-state devices and, for use as a simple buffer, the gates must be enabled (in the case of Group 1) by applying a low (~0V) input to Gate 1, or (in the case of Group 2), by applying a high (~5V) input to Gate 2.]

Note that these buffers are all members of the '74' series of ICs. All these 74x buffers can handle currents of 20 mA per pin. The total current per device is 70 mA in the case of the 240/241/244 devices and 50 mA for the 04. If your load satisfies the 20 mA per pin requirement but not the 'per device' requirement, you will need to use more than one buffer chip.

All these buffers operate reasonably rapidly, with a maximum delay of about 1 μ s.

Logic families

The various '74' series buffers just described are available in numerous versions, including 'LS' (the 74LS04 etc.), 'ALS', 'HC', HCT, VHCT, AHC, AHCT and so on. The different versions have different switching speeds, power consumption, operating voltages and prices. In general, you can choose any buffer that matches the needs of your project.

Despite the vast range of '74' buffers available there are only two logic families: CMOS and TTL. The CMOS devices generally have a 'C' somewhere in the name (e.g. 74HC04) while the TTL devices generally have an 'S' somewhere in the name (e.g. 74ALS04). You need to be aware of the differences between these two families.

The original 5V TTL family dates from around 1964. It has undergone various improvements and is still widely used. Its main advantage is that it is fast; its main disadvantage is that it has comparatively high power consumption figures.

The main competitor to TTL is the CMOS family. The 5V CMOS dates back to around 1983 and it too has undergone various improvements. Its main advantage is low power consumption and – in recent devices – speed has been greatly improved. It seems likely that CMOS logic families will come to replace TTL logic over the next few years.

Any 8051 microcontroller can drive a buffer made from TTL or CMOS logic without difficulty. However, there are differences in the buffer outputs depending on the technology used; these differences are important:

- With (5V) TTL, the Logic 0 output is in the range 0 to 1.5V; the Logic 1 output is 3.5 to 5V.
- With (5V) CMOS, the Logic 0 output is ~0V; the Logic 1 output is ~5V.

These differences have significant implications. For example, consider that we wish to use a CMOS logic gate to buffer an LED output (Figure 7.12). This approach works very effectively, because of the large, fixed voltage swing; however, consider the same buffer implemented using a TTL buffer (Figure 7.13).

In Figure 7.13 we show that the TTL buffer has two disadvantages. First, we need a pull-up resistor (the usual 5K–10K value) to pull the ‘high’ output to 5V. Second, the ‘low’ output varies, in a range from 0 to 1.5V. This makes it very difficult to choose an appropriate value of resistor to ensure that we have a bright display and – at the same time – do not exceed the buffer or LED current capacity.

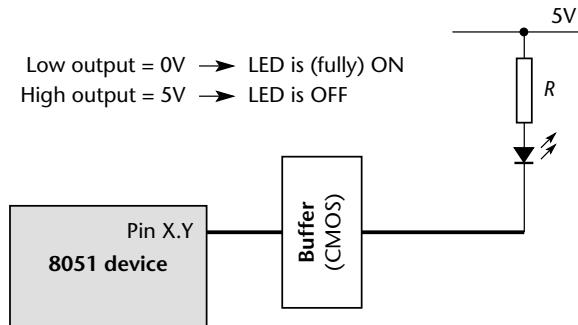


FIGURE 7.12 Using a CMOS buffer

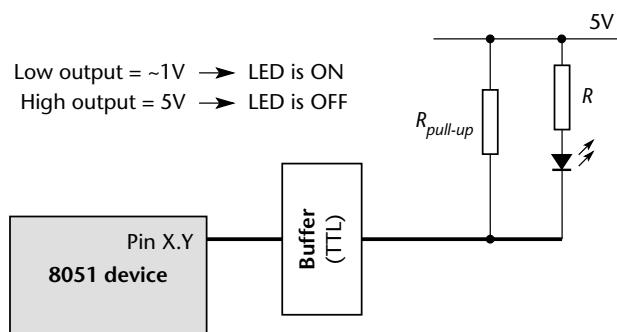


FIGURE 7.13 Using a TTL buffer

Use CMOS buffers

As these discussions suggest, it makes sense to use CMOS logic in your buffer designs wherever possible. You should also make it clear in the design documentation that CMOS logic is to be used.

If working with boards designed by other people, the use of pull-up resistors at the buffer outputs can suggest that TTL logic was assumed; however, CMOS logic can still be used. If there is no pull-up resistor, then CMOS should be used.

Using pull-up resistors at the buffer inputs

As usual, if working with port pins that do not have internal pull-up resistors, you need to include such resistors (10K will do) at the **input** to the IC buffer, whatever kind of logic you are using.

See **NAKED LED** [page 110] for further details.

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety implications

A key design decision to be made when driving a small output device is whether or not to use a buffer.

In some circumstances, omitting the buffer can make your system potentially less safe. For example, if you have an LED accessible at the front of an embedded application, someone wishing to interfere with the system may try to apply a high voltage across this LED or to damage it physically. If the LED is connected directly to a port pin, then it may be possible to damage the microcontroller itself through damage to the LED: if there is a suitable buffer between the LED and the microcontroller, it is much less likely that any damage to the microcontroller will be possible.

Use of a buffer will increase production costs. However, if there is any possibility of the output device suffering damage while the product is in use, a buffer may be a good option: it is almost always cheaper to replace a blown buffer (~\$0.10) than it is to replace a blown microcontroller (~\$1.00+). As a result, for low-volume and / or high-cost products where repairs may be required, then buffers are a good solution.

Overall, if reliability is an issue, use a buffer. Otherwise, in very low-cost (or high-volume) products, or in situations where repair is not a practical proposition, then use of a buffer will simply add to production costs.

Portability

These techniques work with all 8051s (and most other microcontroller and microprocessor families).

As usual, if working with port pins that do not have internal pull-up resistors, you need to include such resistors (10K will do) in your design: see **NAKED LED** [page 110] for further details.

Overall strengths and weaknesses

- ☺ IC buffers allow multiple (small) loads to be controlled from a single port.
- ☺ Buffers can improve reliability.
- ☺ Use of buffers increases the product cost.

Related patterns and alternative solutions

The other patterns in this chapter (particularly **IC DRIVER** [page 134]) provide alternative solutions.

Example: Buffering three LEDs with a 74HC04

Figure 7.14 shows a 74HC04 buffering three LEDs. As discussed in ‘Solution’, we do not require pull-up resistors with the HC (CMOS) buffers.

In this case, we assume that these LEDs are to be driven at 15 mA each, which is within the capabilities (50 mA total) of the buffer.

The required resistor values are:

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

Please refer to **PORT I/O** [page 174] for suitable software.

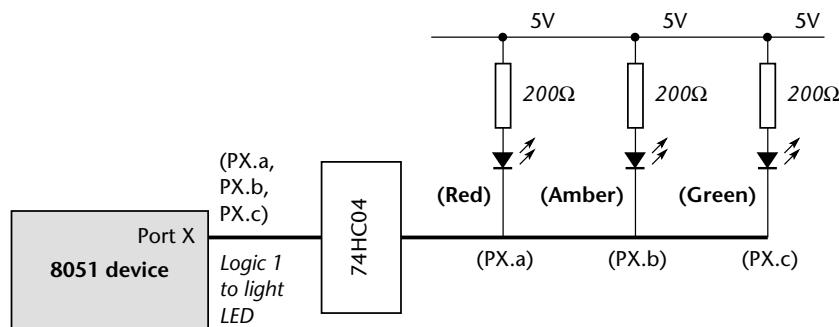


FIGURE 7.14 Driving three LEDs via a 74HC04 inverter

Further reading

BJT DRIVER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you drive DC loads requiring currents of up to around 2A, from the port of an 8051 microcontroller?

Background

A bipolar-junction transistor (BJT) is a three-terminal semiconductor device, developed in the early 1950s. The full name (or acronym) will be used here to distinguish this pattern from the FET (field-effect transistor) considered later in this chapter.

As far as we are concerned, a BJT is most useful as a current-controlled switch. We will not be concerned here with details of their underlying operation but will, instead, focus on the ways in which these transistors may be used to allow a microcontroller to drive a medium-power DC load.

Solution

BJT transistors come in two basic forms: PNP and NPN (Figure 7.15).

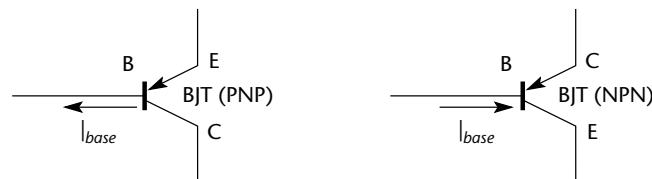


FIGURE 7.15 The two forms of BJT. [Left] PNP. [Right] NPN

[Note: For our purposes, the main value of the BJT is that it acts as a voltage controlled switch. In the case of the PNP device, current flowing from the base is used to control a much larger current through the load. In the case of the NPN device, current flowing into the base of the device performs the same function.]

To ‘turn on’ the transistor, we need to supply (in the case of NPN) or sink (in the case of PNP) a base current given by the formula:

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

where h_{fe} is the transistor ‘gain’. This varies from values of around 15 to around 800: values of 100 are typical for the types of devices we are considering. As previously noted, the pins of an 8051 device can sink or source around 20 mA of current (at most): thus, using a suitable BJT, we can control currents of up to around 2A (20 mA multiplied by h_{fe}).

We will give two specific examples here: a PNP transistor switch and an NPN transistor switch.

A PNP transistor switch

To illustrate the type of BJT application we will be concerned with in this pattern, consider Figure 7.16. This shows a small lamp (5V, 500 mA), connected to a PNP transistor. Note that, in this circuit, almost any PNP transistor will work, provided it has an h_{fe} value of around 100 (given on the data sheet), and can handle a collector current of around 500 mA. Note that in this case, a Zetex ZTX 751 transistor is used. This can control a (collector) current of up to 2A, which is a suitable level for a wide range of applications. Such devices are not expensive: expect to pay around \$0.30 per transistor (multiples of 1,000 devices).

Briefly, the circuit operates as follows. When the microcontroller pin is at Logic 0, current will flow out of the base of the transistor, driving the device into saturation: the required load current will then flow and the lamp will light. When the microcontroller pin is at Logic 1, however, no current will flow from the base of the transistor and the load current will be (essentially) 0.

Overall, the significant thing about this circuit is that a very small control current flowing out of the base of the transistor (I_{base} in the figure) controls the flow of a much larger current (I_{load}) flowing through the lamp. This is the basis of all BJT interface circuits.

Pull-up resistors

When using this pattern, you may need to incorporate pull-up resistors in your hardware design. (See **NAKED LED** [page 110].)

Buffering the output

We use BJTs to allow us to control comparatively high-power loads. The loads can change: for example, devices can short-circuit, motors can stall or the whole device may suffer physical damage. In such circumstances, the microcontroller port can be subject to high voltages and / or currents. This may result in the whole microcontroller being rendered inoperative.

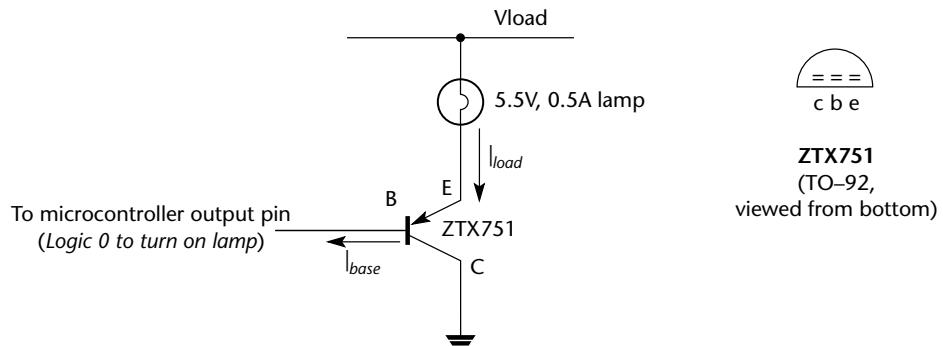


FIGURE 7.16 A PNP transistor driving a lamp. The lamp is controlled by an 8051 microcontroller

[Note: See **NAKED LED** [page 110] for details of pull-up resistors, one of which may be required here. Note the lamp supply voltage is, here, assumed to be 5V: however, the load supply voltage need not be the same as the controller supply voltage.]

In many circumstances, it is safer to use a buffer to protect the microcontroller in the event of a problem with the load or BJT driver. Figure 7.17 shows a suitable circuit.

Note that we have used two inverter gates to obtain the required logic. As multiple gates are available on a single 74x04, and this is a low-cost buffer, this can be a good solution. Alternatively, the 74x241-based circuit in Figure 7.18 may be used.

Note that, for reasons discussed in **IC BUFFER** [page 118], a pull-up resistor may be required at the base of the transistor in both of these buffer circuits, if you use TTL technology.

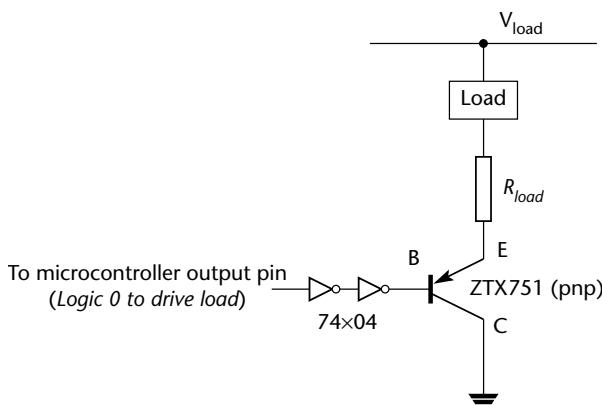


FIGURE 7.17 Adding a 74x04 buffer between the microcontroller and a BJT driver

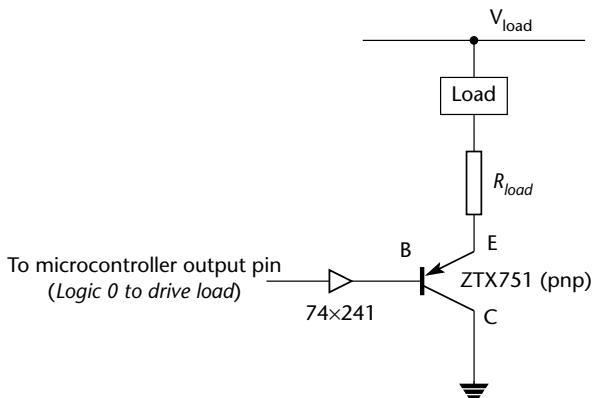


FIGURE 7.18 Adding a 74x241 buffer between the microcontroller and a BJT driver

An NPN transistor switch

We can develop a circuit similar to that shown in Figure 7.17, using an NPN transistor (Figure 7.19).

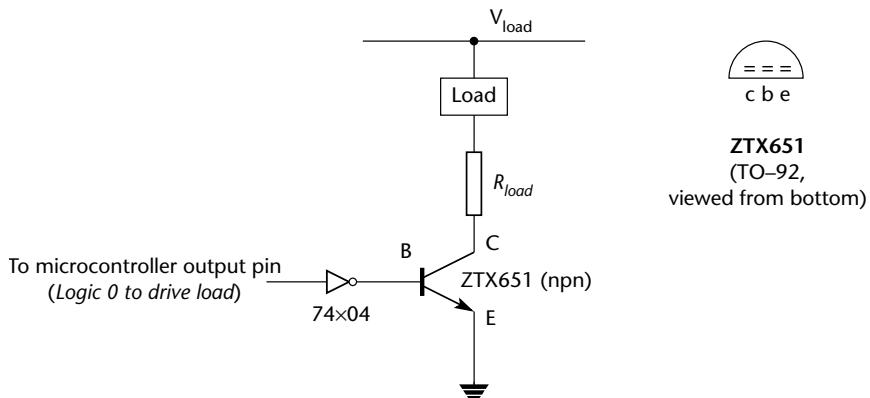


FIGURE 7.19 One way of driving an NPN transistor from an 8051 port pin

[Note: that use of a CMOS buffer is assumed: see IC BUFFER [page 118].]

In Figure 7.19 we have used an inverter to both invert the port output (so that a Logic 0 output is now required to drive the load) and to provide a degree of isolation between the transistor and the microcontroller.

Note that in Figure 7.19, we need to 'source' current: that is, the direction of conventional current flow is into the base of the transistor. As we discussed in NAKED LOAD [page 115], the need to source current may be problematic in some circumstances. For this reason, drivers based on PNP transistors can be slightly more portable.

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety implications

Pin reset values

After the system is reset, the contents of the various port special function registers (SFRs) are set to 0xFF. This fact has very important safety and reliability implications.

Consider, for example, that you have connected a motorized device to a port and that the device is activated by a ‘Logic 1’ output: that is, a ‘1’ output on a port pin. When the microcontroller is reset, the motorized device may be activated and begin moving. Even if you set the port outputs to 0 at the start of your program, the motor may be ‘pulsed’ briefly, before you are able to stop it. This can, in some systems, lead to the injury or even death of users of the system or those in the immediate vicinity of the system.

Because the output pins are ‘reset high’ it is important to ensure that any devices which have safety implications are connected to the microcontroller in such a way that they are ‘active low’: that is, that an output of ‘0’ on the relevant port pin will activate the device.

Switching on lamps and DC motors

We have presented various equations in this chapter that allow the value of series resistors to be calculated for use in drive circuits like that shown in Figure 7.19. In presenting such equations we have implicitly assumed that the load resistance itself is fixed.

In fact, not all devices present a fixed resistive load. For example, when controlling lamps or DC motors, the initial current required may be very high. This surge of current may last several hundreds of milliseconds, before it settles to the steady-state value. Your drive circuit needs to be capable of surviving the initial current surge.

One way of dealing with inrush currents is to ‘over rate’ your drive circuit. This means that if, for example, your load is a lamp or motor with a steady-state current requirement of (say) 1A, you should rate your drive circuit at (say) 10A or more, so that you can deal with the inrush current. Please note that it is seldom possible to guess the likely inrush currents. Check the data sheets – they will provide this information. In the absence of accurate data, assume a factor of at least 10.

Another way of solving this problem is to use what is known as a thermistor in series with the load (Figure 7.20). These have a resistance of around $1\text{--}2 \Omega$ when cold: when they warm up (which they do when carrying current) the resistance drops around 0Ω . Even a small amount of initial resistance will greatly reduce the impact of the inrush current.

Please also note that it is easy to overlook or ignore this issue. Not all drive circuits will fail immediately if subjected to excessive loads and your test circuit may operate reliably on the bench. However, stressing any drive circuit beyond its maximum ratings will dramatically shorten its useful life and it will fail in the field. If in any doubt: over rate by at least a factor of 10 and add a thermistor.

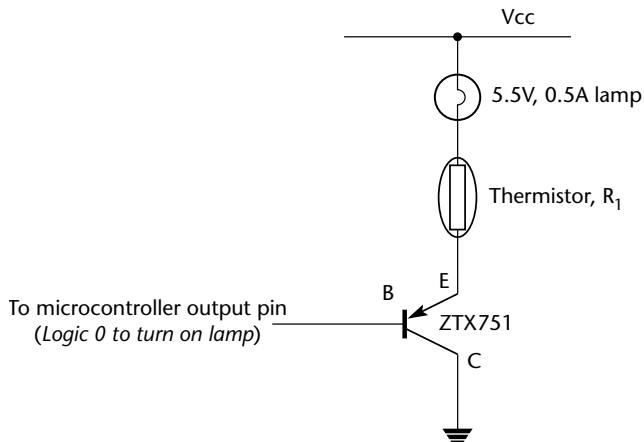


FIGURE 7.20 Using a thermistor to prevent damage due to inrush currents

Switching off inductive DC loads

Just because you have managed to switch on a load safely does not, unfortunately, mean that your problems are solved, if your load is inductive.

An inductive load is anything containing a coil of wire: common examples are electromechanical relays and motors. Switching off such loads must be carried out with great caution because, when the current is removed, the voltage across the inductor will increase rapidly. The rapid increase in voltage occurs because the operation of the inductor is defined by the equation:

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

Here, V is the voltage across the inductor, L is the inductance, and dI/dt is the rate of change of current flow. If we suddenly remove the current (that is, 'switch off' the inductor), the resulting rapid change in current will be translated into a rapid voltage increase. This 'inductive kick' can be enough to destroy logic gates or any other form of drive circuitry linked to the load. To protect these circuits, a diode can be used to block the 'kick' (Figure 7.21).

Please note that the diode used must be able to handle not only the steady-state current but also the turn-off current. The ubiquitous and cheap 1N4004 diode can handle 1A, and is appropriate for many circuits. Where this is not sufficient, a wide range of other capacities are available: for example, the 1N5401 (3A, 100V), the 40HF10 (40A, 100V) or the 70UR60 (250A, 600V). Note that the prices increase dramatically with current capacity: the low-power diodes cost a few cents, while the high-power alternatives may cost \$10.00.

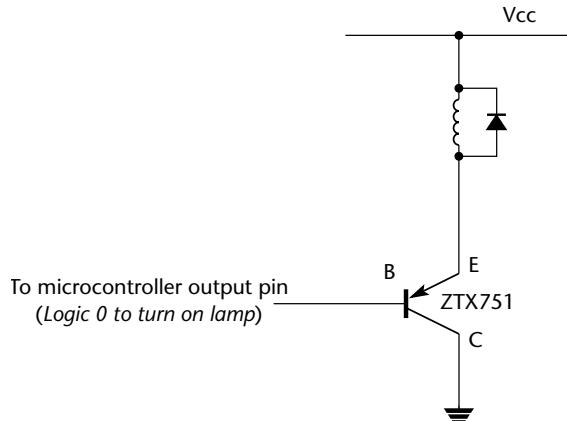


FIGURE 7.21 Protecting transistors against inductive kick with a diode

Note that, for faster current decay after switch off, a resistor can be used in place of the diode (Figure 7.22). At turn off, the voltage will be given by:

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

The voltage drop across the transistor (or your other drive circuit), at turn off, will be:

$$V_{transistor} = V_{cc} + V_{resistor}$$

You must choose a resistor value such that $V_{transistor}$ is within allowed limits.

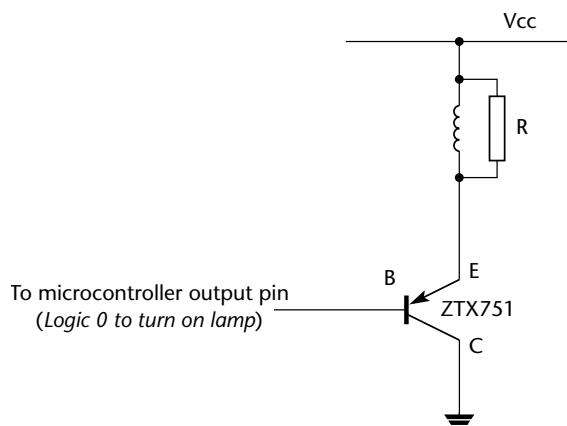


FIGURE 7.22 Protecting transistors against inductive kick with a resistor

Please note (again) that it is easy to overlook or ignore this issue. Not all drive circuits will fail immediately if subjected to excessive loads and your test circuit may operate reliably on the bench. However, stressing any drive circuit beyond its maximum ratings will dramatically shorten its useful life and it will fail in the field. If in any doubt, use a suitable diode or resistor.

Protecting again load faults

The use of thermistors, diodes and inrush resistors is designed to protect the switching circuit under normal conditions. When working with high-power loads, you also need to deal with the problems caused by damage to the load. In particular, we need to consider the possibility that the load will be short-circuited.

A particular problem arises with semiconductor switching devices such as BJTs. The thermal mass of such semiconductor devices is very low: this means that an excessive current flow – caused, for example, by a short or a stalled motor – will very rapidly take the device out of the safe operating region (see Lander, 1993; or Rashid, 1993, for details). To protect against this, it is essential that you use a fuse in series with the load.

Note that an 'ordinary' fuse will not be adequate: the fuse must blow very rapidly to avoid damage to the semiconductor device: such devices are usually labelled 'high speed' and will be described as being suitable for semiconductor protection.

Portability

These techniques work with all 8051s (and most other microcontroller and microprocessor families).

As usual, if working with port pins that do not have internal pull-up resistors, you need to include such resistors (10K will do) in your design.

Overall strengths and weaknesses

- ☺ Software developers sometimes seem to hold the view that discrete transistors are 'old fashioned' and have now been replaced by IC equivalents. This is not the case. In many applications, particularly where a small number of relatively low- and medium-power drivers are required, discrete transistors are widely used, not least because they are available for under \$0.10 each (around half the price of ICs), even in small quantities: as such they can represent a very cost-effective solution.
- ☺ BJTs can operate at very low voltages, compatible with low-voltage microcontrollers.
- ☹ The maximum BJT gain of around 100 means that, in most circumstances, we are restricted to current flows of around 1A–2A, if we use a single transistor. Single MOSFETs, by comparison, can readily switch loads of 100A.

- (?) The switching speed of transistors (and in particular the switch-off speed) can be around 0.5 μ s. This may sound fast, but, particularly in some pulse-width modulation applications (see **HARDWARE PWM** [page 742]), may not be fast enough.

Related patterns and alternative solutions

The main alternatives to the **BJT DRIVER** are **IC DRIVER** [page 134] and **MOSFET DRIVER** [page 139].

Example: Driving a high-power IR LED transmitter

Infra-red (IR) LEDs are widely used in domestic applications for remote control. They are also a component in many security systems. IR LEDs often have higher current requirements than conventional LEDs, but are otherwise connected in the same way.

For example, the Siemens SFH485 IR LED requires a current of 100 mA, at a forward voltage of 1.5V. If we wish to drive this LED using (say) Port 1 of an 8051 microcontroller, then a variation on the circuit from Figure 7.16 can be used.

Here, we will use a low-cost (PNP) transistor: a 2N2905. This can handle a maximum load current of 600 mA. With I_{LED} at 100 mA, a supply voltage of 5V and an LED forward voltage of 1.5V, the required value of R2 is:

$$R2 = \frac{5.0V - 1.5V - 0.4V}{0.100A} = 31\Omega$$

The saturation voltage (V_{CE}) for the transistor of 0.4V is taken from the transistor data sheet.

Here we would use the next standard resistor value, in this case 33Ω . Note that the nearest value is 30Ω : however, by using this we run the slight risk of running the LED at too high a forward voltage, and reducing the life of this component.

The resulting circuit is shown in Figure 7.23.

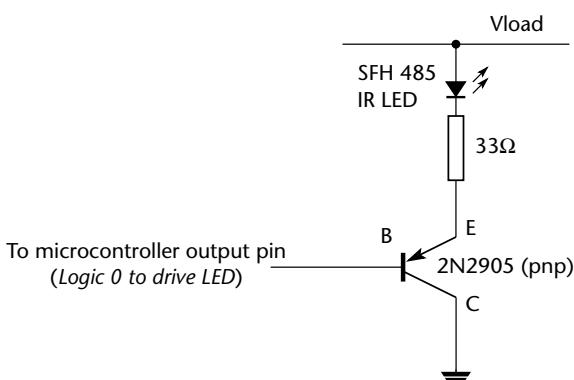


FIGURE 7.23 Driving an IR LED via a transistor (PNP) driver

[Note: A Logic 0 output is required to generate an IR output.]

Example: Driving a large buzzer

Suppose we wish to drive a loud buzzer, as part of an alarm system. The Klaxon KTB buzzer, for example, requires 30 mA and 12V to operate. Figure 7.24 shows a suitable drive circuit.

Here, again, the 2N2905 provides a low-cost solution.

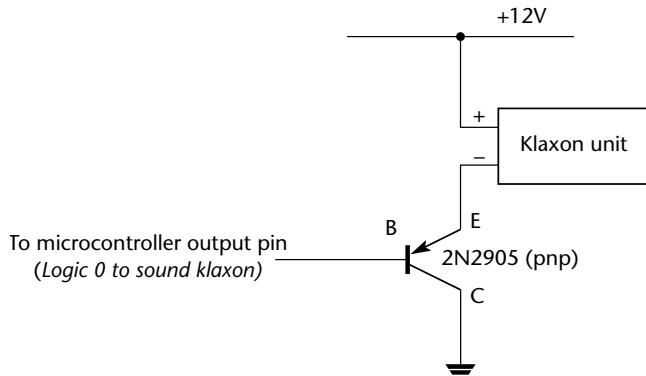


FIGURE 7.24 Controlling a klaxon with a 2N2905 BJT

Further reading

IC DRIVER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you safely control multiple, medium-power DC loads from a single microcontroller port?

Background

See **IC BUFFER** [page 118] for general background material.

Solution

We consider examples of both ‘sink’ and ‘source’ driver ICs in this section.

General-purpose current sinks

Most general-purpose IC ‘sink’ drivers are suitable for switching DC voltages of up to around 50 V (higher voltage variants are available). They are typically capable of sinking currents of around 0.5A to 2A.

One popular driver IC is the ULN2803 series (from various manufacturers). For example, the ULN2803A device contains eight drivers, each capable of switching up to 50V (DC) at 0.5A. Note that each of the drivers is made up of a pair of Darlington-connected (NPN) BJTs: a Darlington arrangement involves a cascading of two transistors to increase the current gain (see Figure 7.25).

Note that this device includes diodes to protect against ‘inductive kick’ on the chip (see ‘Reliability and safety issues’), which can help reduce component counts.

Note also that:

- The ULN2803A requires no explicit power supply connection.
- Pin 9 should be connected to ground.
- If switching inductive loads, the common cathode of the eight built-in diodes (Pin 10) should be connected to the positive rail of the (high-power) supply (up to +50V).
- The switching speed of this device is around 1 μ s.

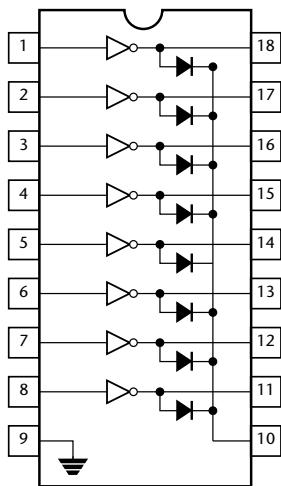


FIGURE 7.25 The pinout and internal details of the Allegro ULN2803A (current sink buffer) (reproduced courtesy of Allegro Microsystems, Inc.)

General-purpose current sources

The ULN2803 family are current ‘sinks’: sometimes, however, we require current sources. Here the UDN2585A is an example of a useful source driver. It can source a current of up to 120 mA on each of the eight output lines, at the same time, at voltages up to 25V (see Figure 7.26).

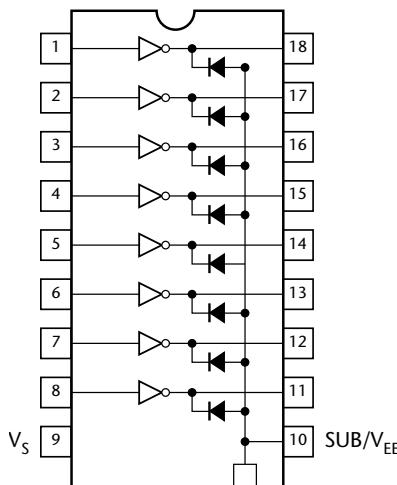


FIGURE 7.26 The UDN2585A (current source driver) (reproduced courtesy of Allegro Microsystems, Inc.)

The UDN2585A will be used in various examples in this book as a driver for multiplexed LED displays: see Chapter 21 for further details.

Note also that:

- Pin 9 on the UDN2585A should be connected to the positive rail of the (high-power) supply (up to +25V).
- If switching inductive loads, then the common anode of the eight built-in diodes (Pin 10) should be connected to ground (it does no harm to do this anyway).
- The switching speed of this device is around 5 µs.

Pull-up resistors

When using this pattern, you may need to incorporate pull-up resistors in your hardware design, at the buffer inputs. See **NAKED LED** [page 110] for further details.

For example, there are no pull-up resistors on pins P1.0, P1.1 on the small Atmel devices, such as the 89C2051. To use these devices, you need to add external pull-up resistors. 10K is a suitable value. Figure 7.27 shows the use of the ULN2803A with an AT89C2051 device.

Note that you will not generally require pull-ups at the buffer outputs.

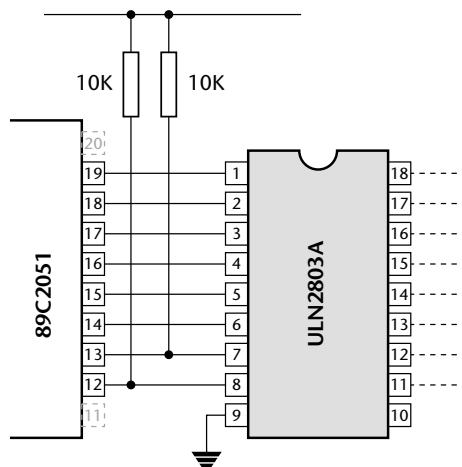


FIGURE 7.27 The need for pull-up resistors on the Atmel AT89C2051

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety implications

There are a number of crucial reliability and safety issues associated with the use of high-power DC loads: these are discussed in the pattern **BJT DRIVER** [page 124]. Please refer to this pattern for further information.

Note that when switching high-power loads in safety-critical applications you may wish to use an **IC BUFFER** [page 118] between microcontroller port and the IC driver chip.

Portability

These techniques work with all 8051s (and most other microcontroller and microprocessor families).

As usual, if working with port pins that do not have internal pull-up resistors, you need to include such resistors (10K will do) in your design.

Overall strengths and weaknesses

- ☺ Can be a cheap and easy to use solution if you require more than two medium-power DC outputs.
- ☹ Unlike transistor drivers, ICs tend to have restricted operating voltages: in many cases, the operating range of such devices is rather less than that of the small 8051s used in many battery-powered applications. This can mean that, if using IC drivers in a battery-powered device, you will be forced to use a more sophisticated (regulated) battery supply: this can add significantly to the application cost. If your application is battery powered, you may find that it is only cost-effective to use an IC driver if four or more output pins require drivers.
- ☹ Most of the cheap driver ICs require 5V supplies and cannot therefore be easily used in 3V designs.

Related patterns and alternative solutions

The other patterns in this chapter provide alternative solutions.

Example: Displaying error codes

A basic requirement in many embedded applications is an ability to inform the user of errors that may have occurred in the application. These may include, for example, errors in sensors, errors in actuators or errors in slave nodes.

A simple, low-cost way of reporting such errors is to make use of an ‘error port’: if the system is operating normally, this port will be used to display a ‘normal’ code: otherwise, an error code will be displayed.

The hardware required to display such codes consists of eight LEDs, each connected to a port pin. To ensure the code is visible under all lighting conditions, high-intensity LEDs may be required. Typical high-intensity LEDs have a current requirement of 25 mA: the total requirement for eight LEDs (200 mA) is therefore far in excess of the current that can be supported by the ‘naked’ microcontroller and is around three times the level that can be supported by an IC buffer.

Figure 7.28 shows a possible hardware solution for displaying the codes, using a ULN2803A as an **IC DRIVER**.

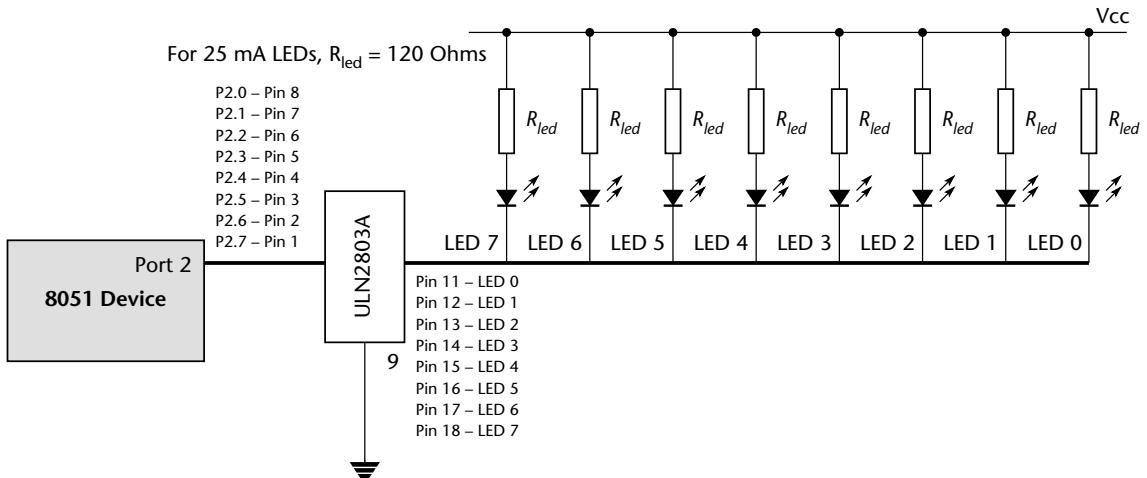


FIGURE 7.28 Hardware for displaying error codes in an embedded application

Note that suitable software for the display of error codes is discussed in Chapter 14.

Further reading

MOSFET DRIVER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you control a DC load with high current requirements (up to around 100A) using a microcontroller?

Background

In **BJT DRIVER** [page 124] we saw that a bipolar junction transistor is a current-controlled switch. By contrast, a metal oxide semiconductor field effect transistor (MOSFET) is a **voltage-controlled** switch (Figure 7.29). With a gate voltage of 0, the device will block current flow between the drain and source, even when a potential of several hundred volts is applied.

MOSFET ‘switch-on’ times are similar to those of the BJT, but – unlike BJTs – the ‘switch-off’ time is also very fast (of the order of 50 ns). As a result the MOSFET is particularly popular in applications requiring high switching frequencies (up to around 1 MHz), such as PWM-based speed control and pulse-rate modulation systems (see, for example, **HARDWARE PWM** [page 808] and **HARDWARE PRM** [page 742]).

One other important characteristic of MOSFETs is that the oxidised metal gate electrode electrically insulates the gate from the channel, which means there is essentially zero current flow between the gate and the channel during any part of the signal cycle. This gives MOSFETs an extremely large input impedance. It also means that the devices are sensitive to static: see ‘Reliability and Safety Issues’.

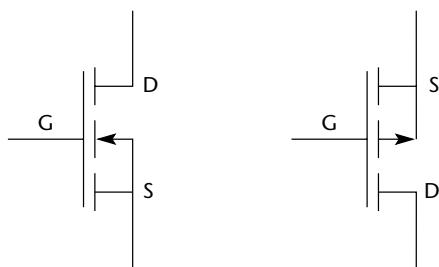


FIGURE 7.29 The two varieties of MOSFET. [Left] An N-channel device. [Right] A P-channel device

The (oxidized metal) gate electrode electrically insulates the gate from the channel, which means there is essentially zero current between the gate and the channel during any part of the signal cycle. This gives the MOSFET an extremely large input impedance. It also means that the devices are sensitive to static: see 'Reliability and safety issues'.

Solution

None of the interface circuits so far considered allow us to switch high DC currents. MOSFETs provide a flexible and cost-effective solution to many high-power applications, able to switch currents of up to 100A or more. Such devices are frequently mounted in a package that allows effective use of heat sinks, to dissipate the heat generated during normal operation of the device (Figure 7.30).

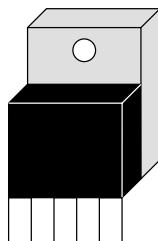


FIGURE 7.30 A typical package (TO-220) for a power MOSFET, providing provision for the use of a heat sink

Increasing numbers of MOSFETs can be driven directly from a processor port. However, in most cases it is necessary to drive the device with a higher voltage (12V +) than is available from the port itself. In addition, as we have discussed in connection with **BJT DRIVER** [page 124], it is generally good policy to have an extra layer of protection between the ports and any high-power load.

In these circumstances, a good solution is to use a 'level-shifter' IC, such as the cheap 74x06, as an interface between the port and the MOSFET (see Figure 7.31).

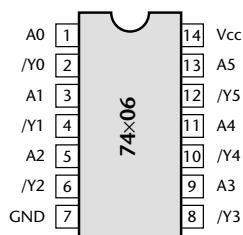


FIGURE 7.31 Pinout of the 74x06 level shifter

[Note: The chip contains six inverting buffers: these link A0 (the input to buffer 0) to /Y0 (the output from buffer 0), A1 to /Y1, and so on. Vcc should be connected to the 5V supply; the voltage at the buffer output can be up to 12V or, in the case of the 74F06A up to 30V, at up to approximately 100 mA.]

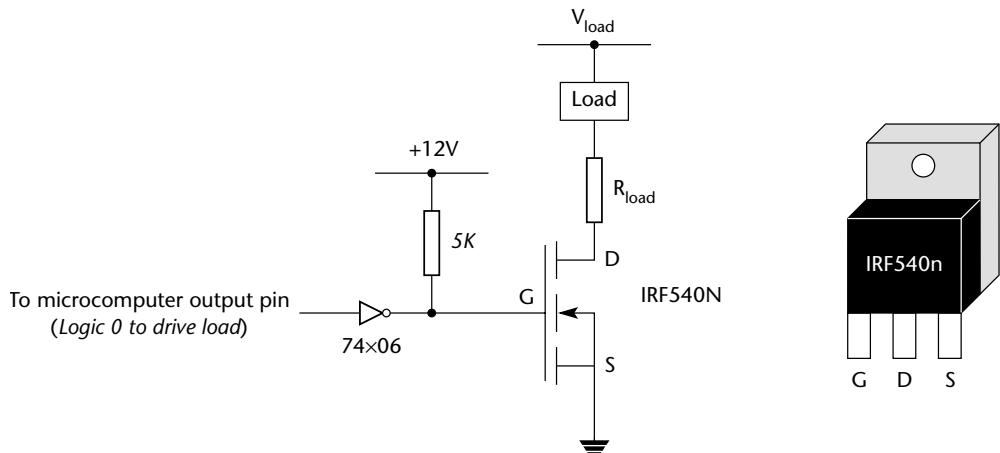


FIGURE 7.32 Using a MOSFET driver

[Note: that a Logic 0 output is required to ‘switch on’ the load. Note that, as we are using the 74x06 for level shifting, the pull-up resistor is required; note that it is connected to the high-power supply rail.]

For example, see Figure 7.32. This illustrates the use of a useful MOSFET, the IRF540, linked to a port via 74x06. Note that that the inverting nature of the 74x06 allows us to ensure that we require ‘Logic 0’ to drive the load: again, this is good policy.

Note also that the IRF540 can switch loads of up to 3A at 100V (DC). Note that (in the N-channel version) it has a very low ‘on’ resistance (0.04 Ohms). This is important as the power loss in the MOSFET is determined by the relationship:

$$P = I^2R$$

At 30A, even a low resistance translates into a substantial power loss. Remember that this power will be dissipated as heat and the larger the loss the greater the cooling requirements will be.

Pull-up resistors

When using this pattern, you may need to incorporate pull-up resistors in your hardware design. See **NAKED LED** [page 110] for further details.

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety implications

There are a number of crucial reliability and safety issues associated with the use of high-power DC loads: these are discussed in the pattern **BJT DRIVER** [page 124]. Please refer to this pattern for further information.

Handling MOSFETs

Because the oxide layer on the gate electrode is extremely thin, the MOSFET is susceptible to destruction by electrostatic discharges. Take suitable precautions when handling them.

Portability

These techniques work with all 8051s (and most other microcontroller and microprocessor families).

As usual, if working with port pins that do not have internal pull-up resistors, you need to include such resistors (10K will do) in your design.

Overall strengths and weaknesses

- ☺ MOSFETs have a large voltage and current capacity.
- ☺ MOSFETs have fast switching speeds.
- ☹ MOSFETs are static sensitive.
- ☹ At low currents, BJTs can be more cost-effective.
- ☹ Solid-state relays, with inbuilt opto-isolation between low- and high-voltage sides of the system, can be more reliable: see **SSR DRIVER (DC)** [page 144].

Related patterns and alternative solutions

We can use power BJTs to do a similar job to the MOSFETs used here (see **BJT DRIVER** [page 124] for background details). However, BJTs tend to be more valuable at lower power levels.

A particular disadvantage of BJTs is that, when in saturation, the voltage drop across the emitter / collector terminals is about 1V. In some applications (for example, H-bridges used for DC motor control) two transistors are required in each 'arm' of the bridge, with the result that the voltage drop becomes around 2V: in typical 12V supplies, this represents a major loss. Using MOSFETs, this loss can be reduced by a factor of around 10. This combined with the general low cost and high switching speeds of the MOSFET solution help to explain why this approach is so popular.

Another alternative, when working with high-current and / or high-voltage loads, is the insulated gate bipolar transistor (IGBT). The IGBT combines some of the best features of BJT and MOSFET approaches. It has a low on-state resistance and a turn-off and turn-on time of around 1 μ s.

Example: Lighting a lamp

Figure 7.33 shows the control of a 12V, 20W lamp using a MOSFET driver.

Note that even this comparatively low-voltage lamp has a large inrush current: the thermistor, R_t , is included in series with the lamp to reduce this current flow: see **BJT DRIVER** [page 124] for further details.

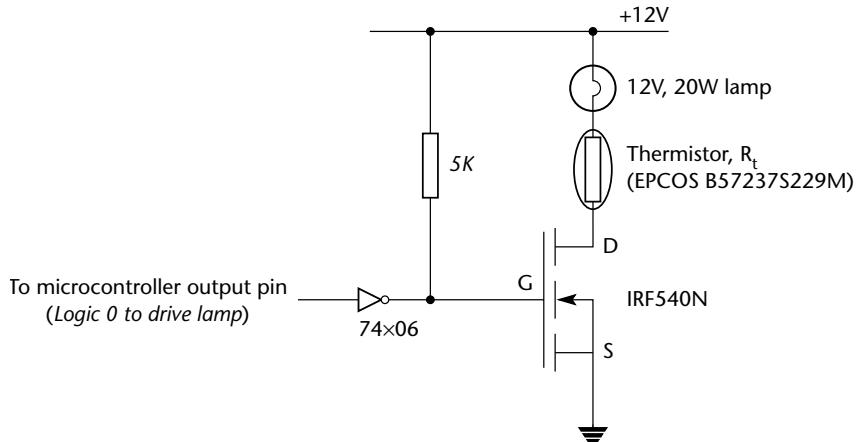


FIGURE 7.33 Controlling a 12V, 20W lamp using a MOSFET driver

Example: Open-loop DC motor control

Figure 7.34 shows the control of a 12V, 2A DC motor using a MOSFET driver.

Note the use of the diode to protect against inductive kick when the motor is switched off: see **BJT DRIVER** [page 124] for further details.

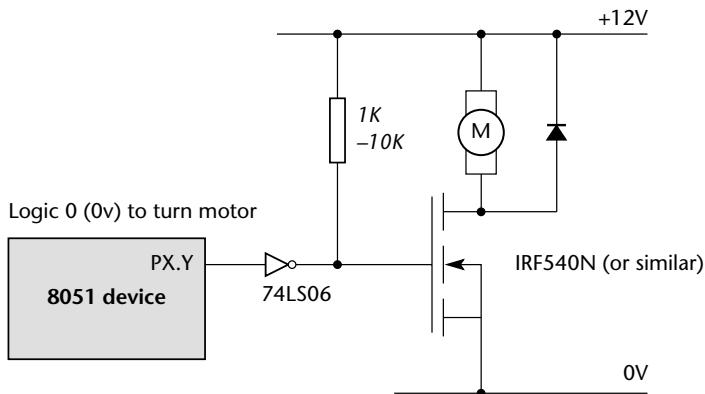


FIGURE 7.34 Control of a DC motor using a MOSFET driver

Further reading

—

SSR DRIVER (DC)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you ‘switch on’ or ‘switch off’ a piece of high-voltage (DC) electrical equipment using a microcontroller?

Background

A solid-state relay is a semiconductor device that was designed to take the place of a conventional electromagnetic (EM) relay. We do not recommend the use of EM relays for switching DC loads and therefore do not consider EM relays until we discuss the switching of AC loads in Chapter 8. If you are unfamiliar with EM relays, you may find it useful to refer to Chapter 8 (and specifically to **EMR DRIVER** [page 149]) for background information on EM relays before considering this pattern.

Solution

Unlike EM relays, solid-state relays (SSRs) are purely electronic in nature: they have no moving (mechanical) switch contacts. Both DC and AC solid-state relays are available: note that, unlike EM relays, a DC relay will **not** switch AC supplies and a DC relay will **not** switch AC supplies: we explain why this is the case later in the pattern.

SSRs provide very high levels of isolation between the ‘control’ and ‘switching’ circuits by optical techniques. The ‘control’ inputs will be connected to one (or more) LEDs. These will then, without any electrical link, control a phototransistor or photo-diode array, which will, in turn, connect to further switching circuitry. In the case of a DC SSR, the switching circuit will typically be based on a MOSFET, and the current- and voltage-switching capabilities will generally be similar to MOSFETs.

Use of SSRs is generally straightforward: the inputs are directly compatible with microcontroller port voltages, and – because of the built-in opto-isolation – there is generally no need to add additional gates between the microcontroller and the SSR.

The examples below will illustrate the use of these devices.

Pull-up resistors

When using this pattern, you may need to incorporate pull-up resistors in your hardware design, at the input to the SSR. See **NAKED LED** [page 110] for further details.

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety issues

There are a number of general reliability and safety issues associated with the use of high-power DC loads: these are discussed in the pattern **BJT DRIVER** [page 124]. Please refer to this pattern for further information.

How robust are SSRs?

SSRs are less electrically robust than EM relays: in the presence of excessive voltages (due to back EMF from inductive loads, for example) or where the SSR is subjected to larger currents (possibly due to ‘inrush’), the device will fail. If in doubt, over rate the device: that is, use a 300 V device where a 200 V device would probably do.

One other issue: we have seen people try to use more than one SSR in parallel in order to increase the current rating. This will only rarely work and is **never** reliable. The problem is that you have no way of ensuring that both relays switch on at **exactly the same time**. When one relay has turned on, the supply voltage drop will usually mean that the second (and subsequent) SSRs do not turn on – until the first relay fails. Thus, the likely result is that, within around a millisecond, all the relays will blow, in rapid succession.

What's in a name?

Despite the name, there are important differences between ‘solid-state’ and ‘electro-mechanical’ relays, particularly when it comes to circuit testing. If you are using an EM relay, you can check to make sure that the contacts are closing by using a multimeter to measure the resistance of the switch contacts: this resistance will be essentially zero when the contacts are closed and essentially infinite when they are open. This behaviour will be observed without connecting up the high-voltage side of the application.

You cannot test an SSR-based circuit in the same manner: most SSRs will always show an infinite resistance when measured with a multimeter. To test an SSR, you need to operate close to the specified operating voltage. Initial tests are therefore best performed (carefully) using a high-voltage load (we usually use an ordinary household lightbulb).

What happens when it goes wrong?

The typical failure mode of an SSR is an output short circuit: this can be dangerous.

Portability

SSRs can be used with any processor type. However, there are other portability issues to consider.

The most important (already briefly mentioned) is that an AC SSR cannot be used to switch DC. The reason for this is that the AC SSR contains zero-crossing detection circuits (see Chapter 8 for details). Because the DC supply never crosses zero, the SSR will never switch on.

Similarly, most DC SSRs are based on MOSFETs. Using a MOSFET to switch AC is ineffective: at best, the device will act as a form of rectifier for a short period, until it overheats.

Overall strengths and weaknesses

- 😊 SSRs do not wear out (in normal use).
- 😊 SSRs are resistant to shock and vibration.
- 😊 SSRs have a high switching speed.
- 😊 SSRs have high levels of isolation between the ‘control’ and ‘switching’ circuits.
- 😊 SSRs generate only very low levels of electrical noise and generate no acoustic noise.
- 😊 SSRs do not exhibit switch bounce.
- 😢 SSRs can be instantly and irrevocably damaged by excessive voltage and / or current.
- 😢 The typical failure mode of an SSR is an output short circuit: this can be dangerous.
- 😢 The ‘switched’ side has a minimum operating voltage and current: these may be quite high, complicating initial testing.
- 😢 EM relays can usually switch higher voltages and currents.
- 😢 Unlike an EM relay, the ‘switched’ side exhibits some leakage current in the off-state.
- 😢 The ‘on’ resistance is typically much larger than that of an EM relays: this translates directly into wasted heat and, hence, the need for heatsinks.

Related patterns and alternative solutions

See the other patterns in this chapter and Chapter 8, for alternative approaches.

Example: SSRs in telecommunication applications

Small DC semiconductor relays are commonly used in telecommunications equipment, such as modems, in place of larger EM relays. Indeed, the telecoms market is so important that special-purpose SSRs, intended to be used for telephone current line sensing, are also available (see, for example, products from Erg components).

In modems and similar devices, these SSRs provide 200-300V (DC) output ratings at around 200 mA. They provide a low on resistance (typically 10Ω) and an 'off' resistance of some $500\text{ M}\Omega$, at voltages up to 4 kV.

Example: Open-loop DC motor control

In **MOSFET DRIVER** [page 139] we presented an example of a MOSFET used for open-loop DC motor control (see Figure 7.34 for details).

If we use an appropriate SSR we can simplify this circuit considerably. For example, our motor required 2A (continuous) at up to 12V for correct operation. Here we can use an IOR PVNO12 SSR. Unlike the majority of SSRs, this can switch AC or DC loads, of up to 20 V, 4.5A. It has no zero-crossing detection. The control current is a maximum of 10mA, which is compatible with our microcontroller. Figure 7.35 shows the required circuit.

Note that an important advantage of the SSR solution is that it provides a greater degree of isolation between the controller and the motor itself. Note also that, to control the speed of this motor, pulse-width modulation may be possible: see **HARDWARE PWM** [page 808] and **SOFTWARE PWM** [page 831].

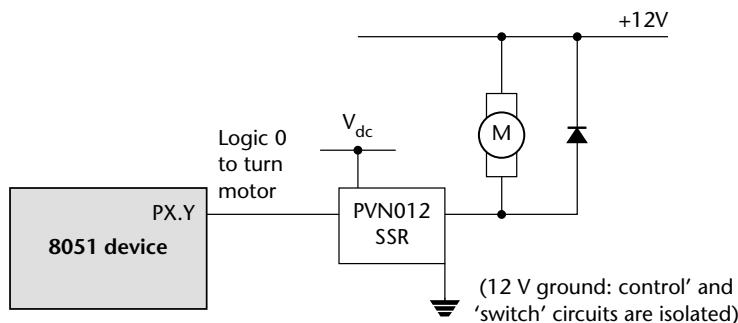


FIGURE 7.35 Open-loop DC motor control using a solid-state relay

Further reading

8

chapter

Driving AC loads

Introduction

Consider some problems:

- A pump in a domestic or industrial heating system must be switched on and off at preset times.
- A series of high-intensity lights around the perimeter of a factory complex are to be activated in the event of a security incident.
- An electrical heater in a brewery must be used to maintain a precise temperature.
- The motor in a washing machine must go through a pre-programmed series of movements.

In these – and many other applications – we need safely and reliably to control comparatively high-level AC supplies. In this chapter we will consider two popular approaches to this problem

- **EMR DRIVER** [page 149]
- **SSR DRIVER (AC)** [page 156]

EMR DRIVER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you ‘switch on’ or ‘switch off’ a piece of high-power (AC) electrical equipment using a microcontroller?

Background

Solution

We consider in this pattern how, where appropriate, we can use an electromechanical (EM) relay to control an AC supply.

An EM relay is, in essence, a mechanical switch controlled by the current flowing through a solenoid. These devices have been used for many years to switch on and off the AC loads, often at very high power levels. In most cases, large electromechanical relays cannot be driven directly from the microcontroller port and will require the use of some form of transistor or IC drive circuit (for example, see Figure 8.1).

More recently, some electromechanical relays have become available that operate at logic levels (a few mA, 5V). Typical examples of such devices are small ‘reed relays’ of the type illustrated in Figure 8.2.

Reed relays of this type are available that will switch mains voltages (up to 250V AC) at powers of up to around 10W. Note that many devices have internal diodes across the relay coil and that various combinations of switch (normally open, normally closed, multiple switches) are available.

Pull-up resistors

When using this pattern, you may need to incorporate pull-up resistors in your hardware design. See **NAKED LED** [page 110] for further details.

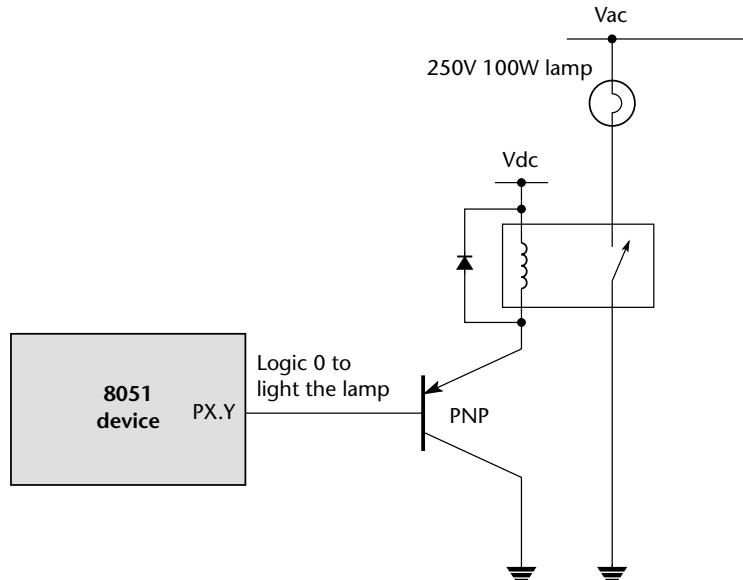


FIGURE 8.1 Using an electromechanical (EM) relay to control an AC bulb

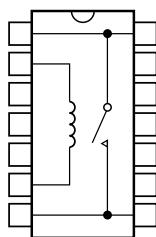


FIGURE 8.2 A typical small (EM) reed relay

Reliability and safety issues

Working with mains voltages

Always keep in mind that people and circuits that deal with mains electricity do not mix. You must design mains switching circuits in such a way that the dangerous voltages are not accessible to the user.

Pin reset values

After the system is reset, the contents of the various port special function registers (SFRs) are set to 0xFF. **This fact has very important safety and reliability implications.**

This issue is discussed in **BJT DRIVER** [page 124]: please refer to this pattern for details. Briefly, because the output pins are ‘reset high’ it is important to ensure that any devices which have safety implications are connected to the microcontroller in such a way that they are ‘active low’: that is, that an output of ‘0’ on the relevant port pin will activate the device.

Zero-crossing detection

Take a radio and hold it next to a (mechanical) light switch in your house or office. Turn the lights on and off while listening to the radio. The ‘crackles’ you hear are a symptom of electromagnetic interference (EMI), generated by an ‘arc’ that forms between the mechanical switch contacts as the switch is opened or closed. This form of EMI can be annoying for radio listeners: for embedded systems, it can be fatal. You therefore need to take care when switching high-power loads. This is a particular problem with AC loads, because such loads tend to be at high voltages.

Solutions are available. To understand how these work, consider a simple source of alternating current illustrated in Figure 8.3. We can greatly reduce the interference caused by switching at an appropriate point in the cycle: in Figure 8.3 switching at Point A will cause maximal interference, while switching at Point B will cause almost no interference. Therefore, to minimize interference, we need to detect the time at which the waveform ‘crosses zero’ and switch at this point.

A key disadvantage of EM relays is that they do not incorporate zero-crossing detection circuitry.

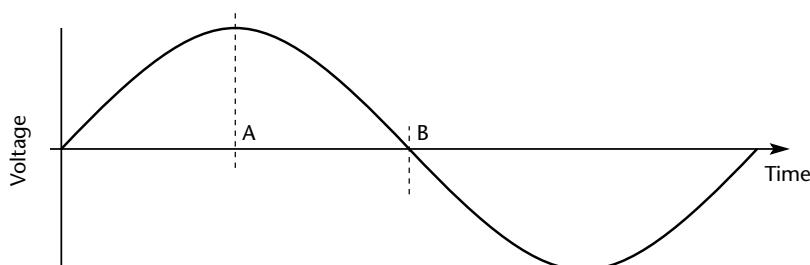


FIGURE 8.3 An AC supply showing sub-optimal (A) and optimal (B) switching points

Switching on lamps and other inductive loads

As we discussed in Chapter 7, not all loads present a fixed resistive load. For example, when controlling lamps or AC motors, the initial current required may be very high. This surge of current may last several hundreds of milliseconds, before it settles to the steady-state value. Your drive circuit needs to be capable surviving the initial current surge.

One way of dealing with inrush currents is to ‘over rate’ your drive circuit. This means that if, for example, your load is a lamp or motor with a steady-state current requirement of (say) 1A, you should rate your drive circuit at (say) 10A or more, so that you can deal with the inrush current. Note that it is seldom possible to guess the likely inrush currents. Check the data sheets – they will provide this information. In the absence of accurate data, assume a factor of at least 10.

Remember also that, as we saw in **BJT DRIVER** [page 124], another way of solving this problem is to use a thermistor in series with the load.

Not all drive circuits will fail immediately if subjected to excessive loads and your test circuit may operate reliably on the bench. However, stressing any drive circuit beyond its maximum ratings will dramatically shorten its useful life and it will fail in the field. If in any doubt: over rate by at least a factor of 10 **and** add a thermistor.

Switching off inductive AC loads

As we discussed in connection with DC loads (Chapter 7), an inductive load is anything containing a coil of wire: common examples are electromechanical relays and motors. Switching off such loads must be carried out with great caution because, when the current is removed, the back EMF across the inductive load can cause damage to the switching device.

To protect any form of switch controlling an inductive DC load, a diode can be used to block the back EMF (‘inductive kick’) as shown in Figure 8.4.

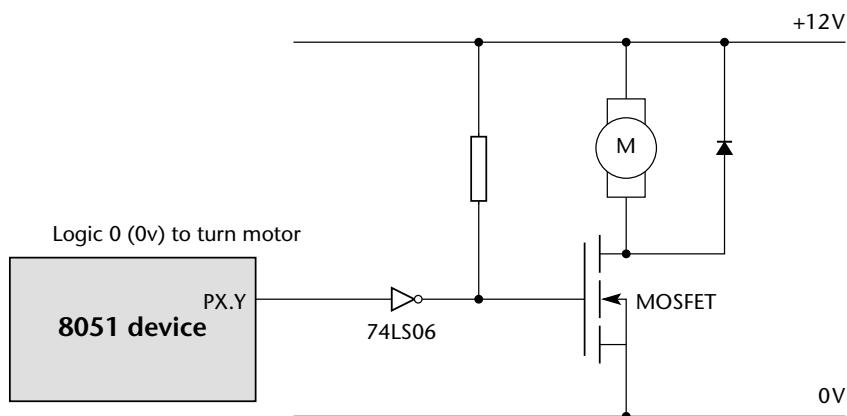


FIGURE 8.4 Protecting switches against back EMF (‘inductive kick’) with a diode

[Note: This approach is only suitable for use with DC loads.]

In the case of AC loads, this approach will not work, since the diode will simply block one phase of the drive voltage. However, we also discussed an alternative approach suitable for use with DC loads in Chapter 7: this involved the use of a resistor to block the back EMF (Figure 8.5).

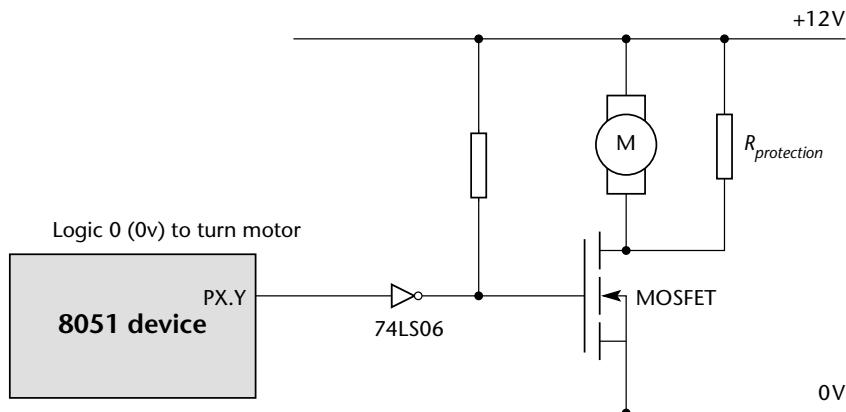


FIGURE 8.5 Using a resistor to block back EMF from an inductive load (in this case a DC motor)

A variation on the resistor-based protection scheme is effective and widely applied to AC load switching: this is known as the RC snubber (Figure 8.6). In most cases, resistor values ($R_{snubber}$) of 10 to 10 K Ω and capacitor ($C_{snubber}$) 0.01 μ F–1 μ F are used (Lander, 1993): values of 100 Ω and 0.05 μ F will be acceptable in many circumstances, but you should investigate this issue carefully for safety-critical applications.

Portability

EM relays may be used with any microcontroller, microprocessor or DSP chip.

Overall strengths and weaknesses

- ☺ EM relays can control both DC and AC loads, with current from a few milliamps to several thousand amps.
- ☺ When the contacts are open, there is no leakage: that is, there is a very high (near infinite) off-state resistance at voltages up to around 1500V.
- ☺ When closed, the switch contacts present a very low resistance, so that the power losses in the relay are very low. The relays do not therefore get hot and usually do not require a heat sink.
- ☺ Purchase costs of EM relays are often lower than semiconductor equivalents (but see comments about maintenance cost).
- ☺ Switching times are typically measured in milliseconds rather than the microsecond values found in semiconductor switches.

- (?) Like all mechanical switches, the relay contacts will usually exhibit ‘bounce’ behaviour when opened or closed (this bounce behaviour is considered in detail in Chapter 19).
- (?) Because EM relays generally do not incorporate zero-crossing detection (see ‘Reliability and safety issues’), they can generate arcs at the switches and, thereby, cause higher levels of EM interference than semiconductor relays.
- (?) Unlike semiconductor switches, relays contain moving parts and moving parts wear out. The mechanical life spans vary, but typical values are around 10 million to 30 million cycles. At ten cycles per day, a 10 million cycle life span is ‘for ever’: however, at one cycle per millisecond, 10 million cycles translates into around three hours.
- (?) If you subject an EM relay to vibration, it is possible to move the switch contacts. This can be dangerous and / or lead to general system reliability problems.

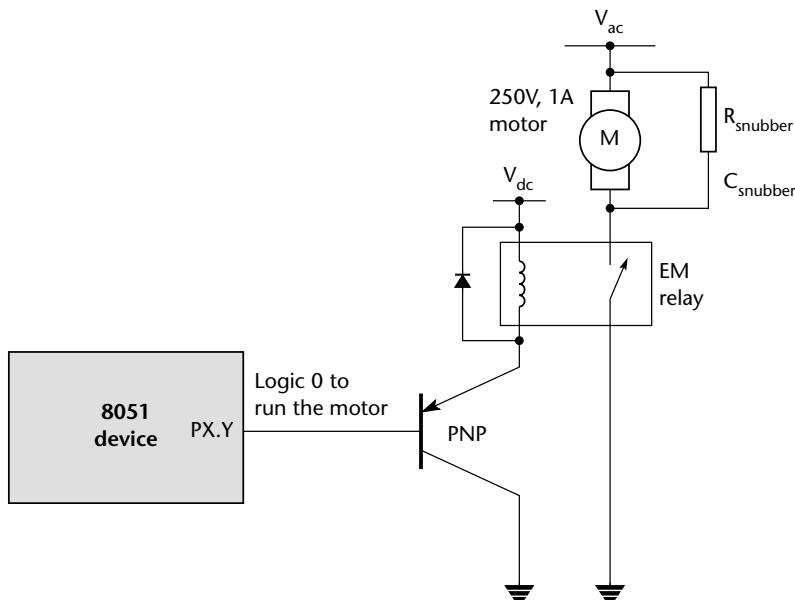


FIGURE 8.6 Using an RC ‘snubber’ network to protect an EM relay when switching an inductive AC load

Related patterns and alternative solutions

It is possible to use EM relays to switch DC loads, but this is rarely a good idea. For better solutions to the DC load control problem, see **BJT DRIVER** [page 124], **IC DRIVER** [page 134] and **MOSFET DRIVER** [page 139].

For alternatives to EM relays for AC load control, see **SSR DRIVER (AC)** [page 156].

Example: Controlling a central-heating pump with an EM relay

Many central-heating systems require the control of small, low-power pumps. An example of a typical pump is a Grundfos Selectric UPS 15–50: this requires 240V AC (50 Hz) and 0.17–0.42A.

Control of central heating is a good use of an EM relay. The heating is switched on a small number of times per day, so relay life will be long. Maintenance is possible, in the event of a relay failure. The occasional EM spikes from the system are unlikely to be troublesome in most domestic environments.

In this case, a reed relay will be capable of carrying the required current. Figure 8.7 shows a possible circuit.

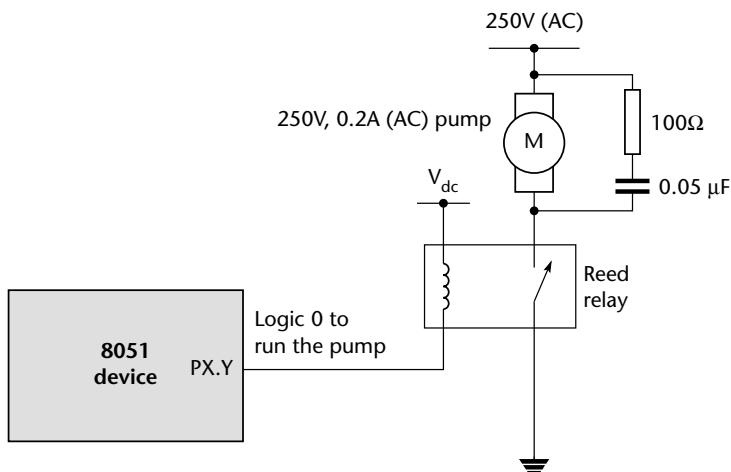


FIGURE 8.7 Use of a reed (EM) relay to control a central-heating pump

Further reading

SSR DRIVER (AC)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you ‘switch on’ or ‘switch off’ a piece of mains-powered (AC) electrical equipment using a microcontroller?

Background

We introduce DC SSRs in **SSR DRIVER (DC)** [page 144]. Please refer to the previous pattern for general background information on SSRs.

The main differences between the two devices is that, while in DC SSRs, the output stage typically consists of a MOSFET, in AC SSRs, the output stage is usually a TRIAC. Very briefly, a TRIAC is a semiconductor switch that allows current to flow in both directions: this is, of course, precisely what we require with AC loads.

Solution

Use of SSRs is generally straightforward: the inputs are directly compatible with microcontroller port values, and – because of the built-in opto-isolation – there is generally no need to add additional gates between the microcontroller and the SSR. The examples that follow will illustrate the use of these devices.

Pull-up resistors

When using this pattern, you may need to incorporate pull-up resistors in your hardware design. See **NAKED LED** [page 110] for further details.

Reliability and safety issues

See the start of this chapter for general reliability and safety issues.

See also **SSR DRIVER (DC)** [page 144] for basic SSR guidelines.

A key difference between AC and DC SSRs is that, while MOSFETs have very low losses, the TRIAC output stages used in AC SSRs typically have a voltage drop of up to 1.5V when they conduct: this translates directly into a power loss of up to 1.5W per Ampere of current. If using a high-power device (greater than around 4A), you will require a heat sink.

When you fit a heat sink, keep it isolated from the case of your application. **Never** bolt it to the chassis: this can have deadly consequences.

Portability

SSRs can be used with any processor type. However, there are other portability issues to consider.

The most important (already briefly mentioned) is that an AC SSR cannot be used to switch DC. The reason for this is that the AC SSR contains zero-crossing detection circuits. Because the DC supply never crosses zero, the SSR will never switch on.

Similarly, most DC SSRs are based on MOSFETs. Using a MOSFET to switch AC is ineffective: at best, the device will act as a form of rectifier for a short period, until it overheats.

Overall strengths and weaknesses

- ☺ SSRs do not exhibit switch bounce.
- ☺ SSRs do not wear out (in normal use).
- ☺ SSRs do not generate acoustic noise.
- ☺ Many (AC) SSRs incorporate 'zero-crossing detection' circuits. This helps to greatly reduce EM emissions.
- ☺ SSRs are resistant to shock and vibration.
- ☺ SSRs have a high switching speed.
- ☺ SSRs have high levels of isolation between the 'control' and 'switching' circuits.
- ☹ EM relays can usually switch higher voltages and currents.
- ☹ The on-resistance of AC SSRs is **much** larger than EM relays. This means extra heat is generated and you must plan for a heat sink or other forms of cooling.
- ☹ Unlike an EM relay, the 'switch' in the AC SSR (usually a TRIAC) exhibits some leakage current in the off-state.
- ☹ Most AC SSRs will not switch DC, partly because of the zero-crossing detection.
- ☹ SSRs can be irrevocably damaged almost instantly by excessive voltage and / or current: EM relays are more forgiving.
- ☹ The typical failure mode of an SSR is an output short circuit: this can be dangerous.
- ☹ The 'switched' side has a minimum operating voltage and current: these may be quite high, which can make it more difficult to perform initial tests on small-scale prototypes.

Related patterns and alternative solutions

See the other patterns in this chapter and Chapter 7 for alternative approaches.

Example: Controlling a central-heating pump with an SSR

In **EMR DRIVER** [page 149] we present an example of a small reed relay used to control a central-heating pump. Here we consider an alternative solution using an AC SSR.

Recall that the pump we wish to control was a Grundfos Selectric UPS 15-50: this required 240V AC (50 Hz), and 0.17–0.42A. In this case, we will use a Crydom MP240D3 SSR. This is suitable for direct interfacing to a microcontroller and has a current capacity of 3A continuous (80A surge) at up to 280V AC. Figure 8.8 shows a suitable circuit.

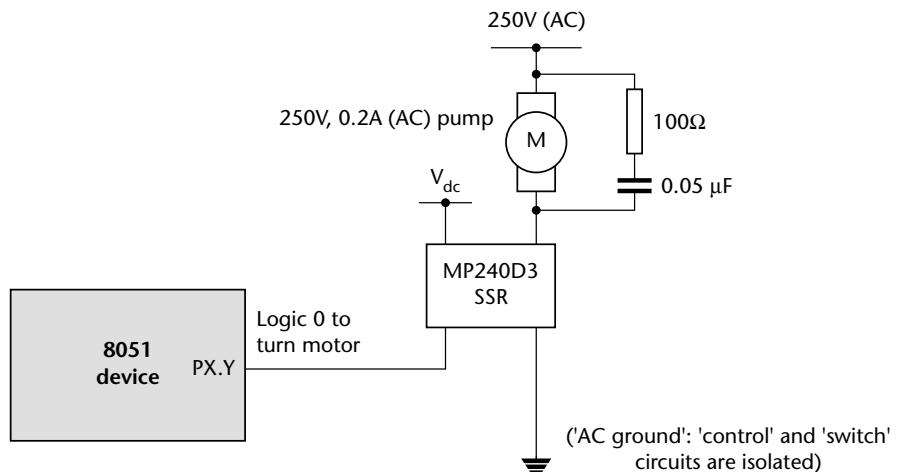


FIGURE 8.8 Use of a Crydom solid-state relay to control a central-heating pump

Overall, control of central heating in this manner is probably best carried out using an EM relay. The heating is switched on a small number of times per day, so relay life will be long. Maintenance is possible, in the event of a relay failure. The occasional EM spikes from the system are unlikely to be troublesome in most domestic environments.

In this case, the EM relay is a better solution, and costs around 20% of the price of the SSR.

Further reading

Software foundations

The chapters in Part A focused on the development of an appropriate hardware environment to meet the needs of your embedded application. In Part B, we consider the corresponding software foundation.

In Chapter 9, we describe the minimum software environment required to create an embedded application.

In Chapter 10, we consider software techniques suitable for use with the AC and DC driver hardware discussed in Chapters 7 and 8.

In Chapter 11, we explore software- and hardware-based techniques for producing delays.

In Chapter 12, we look at the topic of watchdogs timers and consider how these may be used to improve the reliability of your application.

A rudimentary software architecture

Introduction

In the first pattern in this chapter we consider the **minimum** software environment required to create a typical embedded application: this environment is called **SUPER LOOP** [page 162].

Please note that we will use Super Loop in this book primarily to allow us to illustrate some introductory software patterns in Chapters 10, 11 and 12. In Chapter 13 we will demonstrate that a co-operative scheduler provides a more appropriate environment than a **SUPER LOOP** for the great majority of embedded applications.

The second pattern (**PROJECT HEADER** [page 169]) is a practical implementation of a standard software design guideline: ‘Do not duplicate information in numerous files; place the common information in a single file and refer to it where necessary.’ Specifically, **PROJECT HEADER** pulls together the information connected with the particular microcontroller used in your application, along with other key pieces of information that are required by many of the files in your project.

SUPER LOOP

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontroller.
- You are designing an appropriate software foundation for your application.

Problem

What is the minimum software environment you need to create an embedded C program?

Background

Solution

One key difference between embedded systems and desktop computer systems is that the vast majority of embedded systems are required to run only one program. This program will start running when the microcontroller is powered up and will stop running when the power is removed.

A software architecture that is frequently used to generate the required behaviour is illustrated in Listings 9.1 to 9.3.

```
/*-----*  
Main.C  
-----*  
  
Architecture of a simple Super Loop application  
[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
    }
}

/* -----
-----END OF FILE-----
*/

```

Listing 9.1 Part of a simple Super Loop demonstration

```

/* -----
X.H

-----

- see X.C for details.

*/
// Function prototypes
void X_Init(void);
void X(void);

/* -----
----- END OF FILE -----
*/

```

Listing 9.2 Part of a simple Super Loop demonstration

```

/* -----
X.C

-----

'Task' for a demonstration Super Loop application
[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 ...
}

/*-----*
-----END OF FILE-----
*-----*/

```

Listing 9.3 Part of a simple Super Loop demonstration

Listings 9.1 to 9.3 illustrate a simple embedded architecture, capable of running a single task (the function `X()`). After performing some system initialization (through the function `Init_System()`), the application runs the task ‘X’ repeatedly, until power is removed from the system.

Crucially, the ‘Super Loop’, or ‘endless loop’, is required because we have no operating system to return to: our application will keep looping until the system power is removed.

Hardware resource implications

SUPER LOOP has no significant hardware resource implications. It uses no timers, ports or other facilities. It requires only a few bytes of program code. It is impossible to create, in C, a working environment requiring fewer system resources.

Reliability and safety implications

Applications based on **SUPER LOOP** can be both reliable and safe, because the overall architecture is very simple and easy to understand and no aspect of the underlying hardware is hidden from the original developer or from the person who subsequently has to maintain the system. If, by contrast, you are programming for Windows or a similarly complex desktop environment (including Linux or Unix), you are not in complete control: if someone else wrote poor code in a library, it may crash your program. With a ‘super looping’ application, there is nobody else to blame. This can be particularly attractive in safety-related applications.

Please note, however, that just because an application is based on a Super Loop does not mean that it is safe. Indeed, in general, a Super Loop does not provide the facilities needed in an embedded application: in particular, it does not provide a mechanism for calling functions at predetermined time intervals. As we discussed in Chapter 1, these are key characteristics of most embedded applications: if you need such facilities, a scheduler (see Chapter 13) is almost always a more reliable environment.

Portability

Any 'C' compiler intended for embedded applications will compile a Super Loop program: the loop is based entirely on ISO/ANSI 'C'. The code is therefore inherently portable.

Overall strengths and weaknesses

- ☺ The main strength of Super Loop systems is their simplicity. This makes them (comparatively) easy to build, debug, test and maintain.
- ☺ Super Loops are highly efficient: they have minimal hardware resource implications.
- ☺ Super Loops are highly portable.
- ☹ If your application requires accurate timing (for example, you need to acquire data precisely every 2 ms), then this framework will not provide the accuracy or flexibility you require.
- ☹ The basic Super Loop operates at 'full power' (normal operating mode) at all times. This may not be necessary in all applications, and can have a dramatic impact on system power consumption. Again, a scheduler can address this problem.

Related patterns and alternative solutions

In most circumstances, a scheduler will be a more appropriate choice: see Chapter 13.

Example: Central-heating controller

Suppose we wish to develop a microcontroller-based control system to be used as part of the central-heating system in a building. The simplest version of this system might consist of a gas-fired boiler (which we wish to control), a sensor (measuring room temperature), a temperature dial (through which the desired temperature is specified) and the control system itself (Figure 9.1).

We assume that the boiler, temperature sensor and temperature dial are connected to the system via appropriate ports. We further assume that the control system is to be implemented in 'C'.

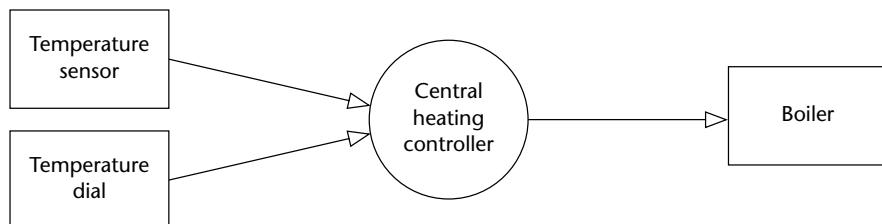


FIGURE 9.1 An overview of the required central heating controller

Here, precise timing is not required, and a Super Loop framework similar to that shown in Listings 9.4 to 9.6 may be appropriate.

```
/*-----*-
Main.C
-----*

Framework for a central heating system using 'Super Loop'.
[Compiles and runs but does nothing useful]
-*-----*/
```

```
#include "Cen_Heat.h"

/*-----*/
void main(void)
{
    // Init the system
    C_HEAT_Init();

    while(1) // 'for ever' (Super Loop)
    {
        // Find out what temperature the user requires
        // (via the user interface)
        C_HEAT_Get_Required_Temperature();

        // Find out what the current room temperature is
        // (via temperature sensor)
        C_HEAT_Get_Actual_Temperature();

        // Adjust the gas burner, as required
        C_HEAT_Control_Boiler();
    }
}

/*-----*
----- END OF FILE -----
-*-----*/
```

Listing 9.4 Part of the code for a simple central-heating system

```
/*-----*-
Cen_Heat.H
-----*
```

```
- see Cen_Heat.C for details.  
- *-----*/  
  
// Function prototypes  
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);  
  
/*-----*  
--- END OF FILE ---  
-*-----*/
```

Listing 9.5 Part of the code for a simple central-heating system

```
/*-----*  
Cen_Heat.C  
-----  
Framework for a central heating system using 'Super Loop'.  
[Compiles and runs but does nothing useful]  
-*-----*/  
/*-----*/  
  
void C_HEAT_Init(void)  
{  
    // User code here ...  
}  
/*-----*/  
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 ...
}

/* -----
----- END OF FILE -----
----- */
```

Listing 9.6 Part of the code for a simple central-heating system

Further reading

PROJECT HEADER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

How do you group together all the information relating to the hardware platform used in your project?

Background

Solution

As we saw in Chapter 3, the 8051 family shares a common set of core facilities. However, it is a *family* – rather than a group of clones – and the different family members have different features and facilities. For example, some devices require 12 oscillator cycles per instruction, while others perform the same instruction in 6, 4 or even 1 oscillator cycle (see Chapters 3 and 4).

If you create an application using a particular 8051 device operating at a particular oscillator frequency, this information will be required when compiling many of the different source files in your project. This information will also be required by anyone who wishes to use your code.

The ‘Project Header’ is simply a header file, included in all projects, that groups all of this information in one place. As such, it is a practical implementation of a standard software design guideline: ‘Do not duplicate information in numerous files; place the common information in a single file, and refer to it where necessary.’

In the case of the great majority of the examples in this book, we use a Project Header file. This is always called `Main.H`. An example of a typical project header file is included in Listing 9.7. Please note that this is a real example and not all of the features of this file have yet been considered in this book.

```
/*-----*-
```

```
Main.H (v1.00)
```

```
-----*
```

```
Project Header for project LCD_KEY (see Chapter 22)
```

```
-----*/  
#ifndef _MAIN_H  
#define _MAIN_H  
  
//-----  
// WILL NEED TO EDIT THIS SECTION FOR EVERY PROJECT  
//-----  
  
// Must include the appropriate microcontroller header file here  
#include <AT89x52.h>  
  
// Must include oscillator / chip details here if delays are used  
// -  
// Oscillator / resonator frequency (in Hz) e.g. (11059200UL)  
#define OSC_FREQ (11059200UL)  
// Number of oscillations per instruction (6 or 12)  
#define OSC_PER_INST (12)  
  
//-----  
// SHOULD NOT NEED TO EDIT THE SECTIONS BELOW  
//-----  
typedef unsigned char tByte;  
typedef unsigned int tWord;  
typedef unsigned long tLong;  
  
// Misc #defines  
#ifndef TRUE  
#define FALSE 0  
#define TRUE (!FALSE)  
#endif  
  
#define RETURN_NORMAL (bit) 0  
#define RETURN_ERROR (bit) 1  
  
//-----  
// Interrupts  
// - see Chapter 12.  
//-----  
  
// Generic 8051 timer interrupts (used in most schedulers)  
#define INTERRUPT_Timer_0_Overflow 1  
#define INTERRUPT_Timer_1_Overflow 3  
#define INTERRUPT_Timer_2_Overflow 5  
  
// Additional interrupts (used in shared-clock schedulers)  
#define INTERRUPT_UART Rx_Tx 4  
#define INTERRUPT_CAN_c515c 17  
// -----
```

```

// Error codes
// - see Chapter 13.
// -----
#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_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
// =====

```

Listing 9.7 An example of a typical project headerfile (Main. H)

Hardware resource implications

There are no hardware resource implications.

Reliability and safety implications

Use of **PROJECT HEADER** can help to improve reliability, not least because it helps to make your code more readable, because anyone using your projects knows where to find key information, such as the model of microcontroller and the oscillator frequency.

Use of **PROJECT HEADER** can help to improve the reliability of applications which are subsequently ported to a different microcontroller, as discussed in the remainder of this chapter.

Portability

The use of a project header can help to make your code more easily portable, by placing some of the key microcontroller-dependent data in one place.

In addition, the `typedef` statements in the file create three key user-defined types which are used in all of the projects in this book:

```

typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;

```

Thus, in the projects you will see code like this:

```
tWord Temperature;
```

Rather than:

```
unsigned int Temperature;
```

If the code is ported into – say – a 16-bit environment, changes to only three `typedef` statements are required in order to adapt the variable sizes to a new compiler. Without the use of these user-defined types, porting the code becomes more complicated and error prone.

Overall strengths and weaknesses

- ☺ PROJECT HEADER can help to make your code more readable, not least because anyone using your projects knows where to find key information, such as the model of microcontroller and the oscillator frequency.
- ☺ PROJECT HEADER can help to make your code more easily portable.

Related patterns and alternative solutions

See PORT HEADER [page 184].

Examples

Almost every example project on the CD includes a project header file. Search for the file Main.H.

Further reading

chapter

10

Using the ports

Introduction

The first pattern in this chapter (**PORT I/O** [page 174]) is concerned with basic software techniques for interacting with the digital ports on an 8051 microcontroller.

The second pattern (**PORT HEADER** [page 184]) encapsulates a design guideline that helps you cope with the fact that many different components in a larger project will each require port access: specifically, **PORT HEADER** demonstrates how the port access for the whole project can be integrated into a single file. Use of these techniques can ease project development, maintenance and porting.

The following points should also be noted:

- This chapter does not consider the hardware that will be connected to the port: see Chapters 7 and 8 for relevant hardware details.
- This chapter does not consider analog input and output: for this, see Part G.

PORT I/O

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

How do you write software to read from and /or write to the ports on an (8051) microcontroller?

Background

The Standard 8051s have four 8-bit ports. All of the ports are bidirectional: that is, they may be used for both input and output.

To limit the size of the device, some of the port pins have alternative functions. For example, as we saw in Chapter 6, Ports 0, 2 (and part of Port 3) together provide the address and data bus used to support access to external memory. Similarly, two further pins on Port 3 (pins 0 and 1) also provide access to the on-chip USART (see Chapter 18). When in their ‘alternative roles’, these pins cannot be used for ordinary input or output. As a result, on the original members of the 8051 family, where external memory is used, only Port 1 is available for general-purpose I/O operations.

These comments all refer to the Standard 8051: the number of available ports on 8051 microcontrollers varies enormously: the Small 8051s have the equivalent of approximately two ports and the Extended 8051s have up to ten ports (see Chapter 3). Despite these differences, the control of ports on all members of the 8051 family is carried out in the same way.

Solution

Control of the 8051 ports through software is carried out using what are known as ‘special function registers’ (SFRs). The SFRs are 8-bit latches: in practical terms, this means that the values written to the port are held there until a new value is written or the device is reset.

Each of the four basic ports on the Standard 8051 family, as well as any additional ports, is represented by an SFR: these are named, appropriately, P0, P1, P2, P3 and so on. Physically, the SFR is a area of memory in the upper areas of internal RAM: P0 is at address 0x80, P1 at address 0x90, P2 at address 0xA0 and P3 at address 0xB0.

If we want to read from the ports, we need to read from these addresses. Assuming that we are using a C compiler, the process of writing to an address is usually by

means of a SFR variable ‘declaration’, hidden in a header file. Thus, a typical SFR header file for an 8051 family device will contain the lines:

```
sfr P0 = 0x80;
sfr P1 = 0x90;
sfr P2 = 0xA0;
sfr P3 = 0xB0;
```

Having declared the SFR variables, we can write to the ports in a straightforward manner. For example, we can send some data to Port 1 as follows:

```
unsigned char Port_data;
Port_data = 0xF;
P1 = Port_data; // Write 00001111 to Port 1
```

Similarly, we can read from (for example) Port 1 as follows:

```
unsigned char Port_data;
P1 = 0xFF;           // Set the port to 'read mode'
Port_data = P1;     // Read from the port
```

Note that, in order to read from a pin, we need to ensure that the last thing written to the pin was a ‘1’. Because the reset value of the ports is 0xFF (see ‘Port reset values’, page 178), it is tempting to assume that writing this value is unnecessary and that we can get away with the following version:

```
unsigned char Port_data;
// Assume nothing written to port since reset
// - DANGEROUS!!!
Port_data = P1;
```

The problem with this code is that, in simple test programs it works: this can lull the developer into a false sense of security. If, at a later date, someone modifies the program to include a routine for writing to all or part of the same port, this code will not generally work as required:

```
unsigned char Port-data:
P1 = 0x00;
...
// Assumes nothing written to port since reset
// - WON'T ALWAYS WORK
Port_data = P1;
```

In general, we use initialization functions to set the ports to a known state at the start of the program. Where this is not possible, it is safer to always write ‘1’ to any port pin before reading from it, as was illustrated in the first example.

Note that, in the last example, we assume that we wished to read from or write to an entire port. More commonly, we might wish (for example) to control an LED is connected to a single output pin. For example, assuming that the LED is connected to Pin 0 on Port 3 of an 8051-family microcontroller, we can flash the diode by controlling the whole port, as follows:

```
P3 = 0xFF;
... // delay
P3 = 0x00;
... // delay
P3 = 0xFF ;
... // etc
```

Alternatively, we can make use of an `sbit` variable in the C51 compiler to provide a finer level of control. At the same time, we consider the fact that – depending on the hardware (see Chapter 7) – the LED may be lit using a logic 1 or a logic 0 output on the port pin and we make the software flexible enough to deal easily with subsequent hardware changes:

```
#define LED_PORT P3
#define LED_ON 0           // Easy to change the logic here
#define LED_OFF 1
sbit Warning_led = LED_PORT^0; // LED is connected to 4.0
...
Warning_led = LED_ON;
... // delay
Warning_led = LED_OFF;
... // delay
Warning_led = LED_ON;
... // etc
```

Reliability and safety implications

Port reset values

After the system is reset, the contents of the various port special function registers (SFRs) are set to 0xFF. This fact has very important safety and reliability implications.

Consider, for example, that you have connected a motorized device to a port and that the device is activated by a ‘logic 1’ output. When the microcontroller is reset, the motorized device will be activated. Even if you change the port outputs to 0 at the start of your program, the motor will be ‘pulsed’ briefly. This can, in some systems, lead to the injury or even death of users of the system or those in the immediate vicinity.

Because the output pins are ‘reset high’ it is important to ensure that any devices which have safety implications are connected to the microcontroller in such a way

that they are ‘active low’: that is, that an output of ‘0’ on the relevant port pin will activate the device.

Port I/O and memory access

One of the most common errors made by inexperienced 8051 developers is to continue to use P0, P2 or P3 as normal I/O ports when using external memory (see Chapter 8 for details of memory usage).

If you use external memory, you cannot safely use P0 and P2 for any other purpose and you must also take care when writing to Port 3.

For example, any statement similar to this:

```
P3 = AD_data;
```

is potentially catastrophic if external memory is being used.

Instead, make use of **sbit** variables to ensure you only write to ‘safe’ port pins (see ‘Example: Reading and writing bits’ for details).

Hardware resource implications

All port I/O involves the use of port pins. As we discussed earlier, if these pins are used for I/O, then they are not generally available for other purposes.

Note that all Extended 8051s provide additional ports. On the 80C515C, for example, there are eight 8-bit ports: these include the ‘standard’ ports (0–3), plus one 8-bit port with alternate A/D conversion functions and three further ports.

Portability

Port access in general and the keywords **bit** and **sbit** in particular are not part of the ISO / ANSI C language: therefore, by definition, this code is not totally portable.

That said, this code can be used (with the standard Keil compiler) across the whole of the 8051 range. With minor modifications it can be used with other 8051 compilers, all of which provide similar facilities.

Overall strengths and weaknesses

- ☺ This pattern allows flexible and efficient access to the 8051 ports, making full use of the internal BDATA memory area (discussed in Chapter 6).
- ☹ As noted earlier, port access in general and the keywords **bit** and **sbit** in particular are not part of the ISO / ANSI C language: therefore, by definition, this code is not totally portable. Note that, while perhaps less than ideal, this problem cannot be avoided.

Related patterns and alternative solutions

Hardware issues

This pattern does not deal with external hardware: see Part A (particularly Chapters 7 and 8) and Part C for patterns that cover these issues.

Interrupt inputs

For reasons discussed in Chapter 1, we make very limited use of interrupt inputs in this book.

Example: Reading and writing bytes

Listing 10.1 illustrates how we can read from a collection of eight switches connected to a port on an 8051-family microcontroller and ‘echo’ these switch settings on an output port: using **SWITCH INTERFACE (SOFTWARE)** [page 399], **NAKED LED** [page 110] and **IC BUFFER** [page 118] we could, for example, use this code to display the switch settings on a panel of LEDs.

Note, however, that you do not require any hardware to try out this code: the Keil hardware simulator (included on the CD) allows you to simulate suitable hardware. Figure 10.1 shows the output from one such simulation.

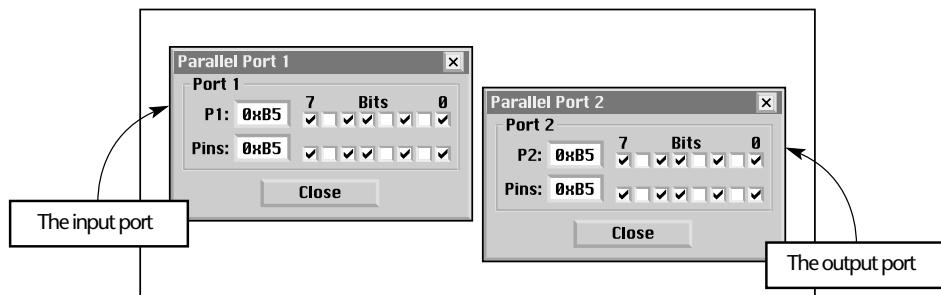


FIGURE 10.1 The output from the program in Listing 10-1 produced using the Keil hardware simulator included on the CD.

```
/*-----*
 * Main.C
 *
 *-----*
 * Test program for pattern PORT I-O
 * Reads from P1 and copies the value to P2.
 *-----*/
```

```

// File Main.H is detailed in Chapter 9
#include <Main.H>

/*.....*/
void main (void)
{
    unsigned char Port1_value;

    // Must set up P1 for reading
    P1 = 0xFF;

    while(1)
    {
        // Read the value of P1
        Port1_value = P1;

        // Copy the value to P2
        P2 = Port1_value;
    }
}

/*-----*
----- END OF FILE -----
-----*/
```

Listing 10.1 A simple Super Loop application which copies the values from P1 to P2

Example: Reading and writing bits

Listing 10.1 demonstrated how to read from or write to an entire port. Consider another common problem: reading and writing individual pins on a port. This problem arises because often the various parts of a port will be serving different purposes.

Suppose, for example, that we have a switch connected to Port 1 (pin 3) and an LED connected to Port 1 (pin 4) and also have other input and output devices connected to the other pins on this port. How do we read from pin 3 and write to pin 4 without disrupting anything else?

We can do this by making use of the bitwise AND, OR and ‘complement’ operators. These, and other, bitwise operators are not widely used by desktop developers. The various bitwise operators allow a number of data manipulations that are invaluable in embedded applications. Some examples of the use of these operators are given in Listing 10.2. Note that this file is written in ISO ‘C’ (‘Desktop C’): it cannot be run on the Keil compiler.

```

/*-----*
Main.C
```

```
----- ~ -----  
Test program for pattern PORT I-O  
Illustrating the use of bitwise operators  
----- /*  
  
#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');
}

/*-----*
----- END OF FILE -----
-*-----*/

```

Listing 10.2 Demonstrating the C bitwise operators

The output from the program in Listing 10.2 is as follows:

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

The use of some of these operators in an embedded application is illustrated in Listing 10.3 which echoes the input on Pin X to Pin Y on Port 1.

```

/*-----*
Main.C

-----*/

Test program for pattern PORT I-O
Illustrating the use of bitwise operators
Reading and writing individual bits
NOTE: Both bits on same port
-*-----*/

```



```
    return (P1 & p); // Read the pin
}

/* -----
----- END OF FILE
----- */

```

Listing 10.3 Reading and writing bits

Example: Displaying error codes in a scheduler

See **CO-OPERATIVE SCHEDULER** [page 255], where error codes are displayed on a bank of LEDs connected to a port.

Example: Controlling an LCD

See **LCD CHARACTER PANEL** [page 467], where control of individual port pins is used to send data to an LCD panel.

Further reading

Further information about the use of the 'C' bitwise operators will be found in any standard textbook on C programming.

PORT HEADER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.

Problem

How do you manage port access allocations in a larger project?

Background

In a typical embedded project, you may have a user interface created using an LCD, a keypad and one or more single LEDs. There may be a serial (RS-485) link to another microcontroller board. There may be one or more high-power devices (say three-phase industrial motors) controlled by your application.

Each of these (software) components in your application will require exclusive access to one or more port pins. The project may include 10–20 different source files. How do you ensure that changes to port access in one component do not impact on another? How do you ensure that it is easy to port the application to an environment where different port pins must be used?

These issues are addressed through the simple **PORT HEADER** design pattern.

Solution

PORT HEADER encapsulates a simple but effective design guideline that can help you cope with the fact that many different components in a larger project will each require port access: specifically, using **PORT HEADER**, you will pull together the different port access features for the whole project into a single (header) file. Use of this technique can ease project development, maintenance and porting.

Port Header is simple to understand and simple to apply.

Consider, for example, that we have three C files in a project (A, B, C), each of which requires access to one or more port pins, or to a complete port.

File A may include the following:

```
// File A  
sbit Pin_A = P3^2;  
...
```

File B may include the following:

```
// File B  
#define Port_B = P0;  
...
```

File C may include the following:

```
// File C
sbit Pin_C = P2^7;
...
...
```

In this version of the code, all of the port access requirements are spread over multiple files. Instead of this, there are many advantages obtained by integrating all port access in a single Port.H header file:

```
//----- Port.H -----
// Port access for File B
#define Port_B = P0;

// Port access for File A
sbit Pin_A = P3^2;

// Port access for File C
sbit Pin_C = P2^7;

...
```

Each of the remaining project files will then `#include Port.H`.

Listing 10.4 shows a complete example of a Port.H file from a real application.

```
/*-----*
Port.H (v1.00)

-----
'Port Header' for the project LCD_KEY (see Chapter 22)
*-----*/
//----- Sch51.C -----
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#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 -----
// NOTE: Any combination of 6 pins may be used (any ports, any order)
// NOTE: Number in [] are pin numbers on *MANY* LCDs

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]

// Connect Vss [1] on LCD to Gnd
// Connect Vcc [2] on LCD to +5V
// Connect Vo [3] on LCD to Gnd
// Connect RW [5] on LCD to Gnd

// ----- LED_Flas.C -----
// Connect LED from +5V (etc) to this pin, via appropriate resistor
// [see Chapter 7 for details]
sbit LED_pin = P1^5;

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 10.4 An example of a real Port Headerfile (Port.H) from a project using an interface consisting of a keypad and liquid crystal display

Hardware resource implications

There are no hardware resource implications.

Reliability and safety implications

Despite its simplicity, Port Header can improve reliability and safety, because it avoids potential conflicts between port pins, particularly during the maintenance phase of the project when developers (who may not have been involved in the original design) are required to make code changes.

Portability

Port Header is itself portable: it can be used with any microcontroller and is not linked to the 8051 family. Use of Port Header also improves portability, by making accessible in one location all the port access requirements of the application.

Overall strengths and weaknesses

 PORT HEADER is both simple and effective – use it!

Related patterns and alternative solutions

See PROJECT HEADER [page 169].

Example: LED bargraph display

Suppose we wish to test an application involving an analog-to-digital converter. Specifically we want to display the voltage reading on a set of 8 LEDs connected to Port 1 of our 8051 device (Figure 10.2).

The key software files required to create this application follow (Listings 10.5 to 10.8): a complete set of the source files for the project is included on the CD.

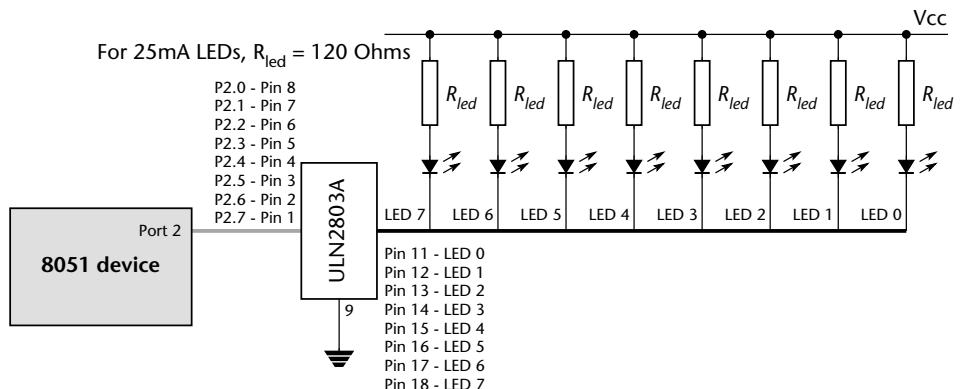


FIGURE 10.2 Hardware for an LED bargraph display.

```
/*
 * Main.H (v1.00)
 *
 * 'Project Header' (see Chapter 9) for the project BARGRAPH
 */
#ifndef _MAIN_H
#define _MAIN_H
```

```
//-----
// WILL NEED TO EDIT THIS SECTION FOR EVERY PROJECT
//-----

// Must include the appropriate microcontroller header file here
#include <AT89x52.h>

// Must include oscillator / chip details here if delays are used
//
// Oscillator / resonator frequency (in Hz) e.g. (11059200UL)
#define OSC_FREQ (12000000UL)
// Number of oscillations per instruction (6 or 12)
#define OSC_PER_INST (12)

//-----
// SHOULD NOT NEED TO EDIT THE SECTIONS BELOW
//-----

typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;

// Misc #defines
#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif

#define RETURN_NORMAL (bit) 0
#define RETURN_ERROR (bit) 1

//-----
// Interrupts
// - see Chapter 12.
//-----

// Generic 8051 timer interrupts (used in most schedulers)
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5

// Additional interrupts (used in shared-clock schedulers)
#define INTERRUPT_UART Rx_Tx 4
#define INTERRUPT_CAN_c515c 17

//-----
// Error codes
// - see Chapter 13.
```

```

// -----
#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_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
//=====

```

Listing 10.5 Part of a small library for creating an LED bargraph display

```

/* ----- *-
Port.H (v1.00)

----- */

'Port Header' for the project BARGRAPH
-*----- */
```

```

//----- Bargraph.C -----
```

```

// Connect LED from +5V (etc) to these pins, via appropriate resistor
// [see Chapter 7 for details]
// The 8 port pins may be distributed over several ports if required
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;

/* ----- */
----- END OF FILE -----
-*----- */
```

Listing 10.6 Part of a small library for creating an LED bargraph display

```

/*-----*
Main.c (v1.00)

-----*/
Demo program for bargraph display
-*-----*/
#include "Main.h"
#include "Bargraph.h"

// ----- Public variable declarations -----
extern tBargraph Data_G;

/* ..... */
/* .. */

void main(void)
{
    tWord x;

    BARGRAPH_Init();

    while(1)
    {
        if (++x == 1000)
        {
            x = 0;
            Data_G++;
        }

        BARGRAPH_Update();
    }
}

/*-----*
----- END OF FILE -----
-*-----*/

```

Listing 10.7 Part of a small library for creating an LED bargraph display

```

/*-----*
Bargraph.c (v1.00)

-----*/

```

Simple bargraph library. See Chapter 10.

```
-----*/  
#include "Main.h"  
#include "Port.h"  
  
#include "Bargraph.h"  
  
//----- Public variable declarations -----  
  
// The data to be displayed  
tBargraph Data_G;  
  
// ----- Private constants -----  
  
#define BARGRAPH_ON (1)  
#define BARGRAPH_OFF (0)  
  
//----- Private variables -----  
  
// These variables store the thresholds  
// used to update the display  
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;
```

```

// Use a linear scale to display data
// Remember: *9* possible output states
// - do all calculations ONCE
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()
   Update the bargraph display.

- */
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);
}

/* -----
   ----- END OF FILE -----
- */

```

Listing 10.8 Part of a small library for creating an LED bargraph display

Further reading

Delays

Introduction

The creation of accurate delays are key requirements in many embedded applications.

In this chapter we will consider two different techniques that may be used to provide such facilities:

- **HARDWARE DELAY** [page 194] which is capable of producing precise delays through the use of one of the on-chip timers. Particularly suitable for generating delays of around 0.1 ms or more.
- **SOFTWARE DELAY** [page 206] which is a simple technique that requires no hardware resources. The most flexible form of delay mechanism that is particularly suitable for generating short delays (measured in microseconds) or where timer resources are not available.

HARDWARE DELAY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

You need to wait for a fixed period of time (measured in milliseconds) before taking some action.

Background

Solution

All members of the 8051 family have at least two 16-bit timer / counters, known as Timer 0 and Timer 1. These timers can be used to generate accurate delays.

We begin by providing brief information on these timers.

Timer 0 and Timer 1

Timer 0 and Timer 1 have much in common and we will consider them together.

To see how these timers operate, we need, first, to introduce the TCON SFR (Table 11.1).

TABLE 11.1 The TCON Special Function Register

Bit	7 (msb)	6	5	4	3	2	1	0 (lsb)
Name	TF1	TR1	TF0	TR0	1E1	IT1	1E0	IT0

[Note: that the grey areas are not connected with these timers.]

The various bits have the following functions:

TF1 Timer 1 overflow flag

Set by hardware on Timer 1 overflow.

(Cleared by hardware if processor vectors to interrupt routine.)

TR1 Timer 1 run control bit

Set/ cleared by software to turn Timer 1 either 'ON' or 'OFF'.

TF0 Timer 0 overflow flag

Set by hardware on Timer 0 overflow.

(Cleared by hardware if processor vectors to interrupt routine.)

TR0 Timer 0 run control bit

Set / cleared by software to turn Timer 0 either 'ON' or 'OFF'.

Note that the overflow of the timers can be used to generate an interrupt. We will not make use of this facility in the Hardware Delay code, but will do so in various scheduler patterns (see Part C and Part F for details).

To disable the generation of interrupts, we can use the C statements:

```
ET0 = 0; // No interrupts (Timer 0)
ET1 = 0; // No interrupts (Timer 1)
```

We also need to introduce the TMOD SFR (Table 11.2).

TABLE 11.2 The TMOD Special Function Register

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			

The first thing to note in TMOD is that there are three main modes of operation (for each timer), set using the M1 and M0 bits. We will only be concerned in this book with Mode 1 and Mode 2, which operate in the same way for both Timer 0 and Timer 1, as follows:

Mode 1 (M1 = 0; M0 = 1)

16-bit timer/counter (with manual reload).¹

Mode 2 (M1 = 1; M0 = 0)

8-bit timer/counter (with 8-bit auto-reload).¹

The remaining bits in TMOD have the following purpose:

GATE Gating control

When set, timer/counter 'x' is enabled only while 'INT x' pin is high and 'TRx' control bit is set. When cleared timer 'x' is enabled whenever 'TRx' control bit is set.

1. See Chapter 13 for a discussion of the difference between 'auto' and 'manual' timer reloads

C/T Counter or timer select bit

Set for counter operation (input from 'Tx' input pin).

Cleared for timer operation (input from internal system clock).

Finally, before we can see how this hardware can be used to create delays, you need to be aware that there are an additional two registers associated with each timer: these are known as TL0 and TH0, and TL1 and TH1. These 'L' and 'H' refer to 'low' and 'high' bytes, as will become clear shortly.

Creating delays with Timer 0 and Timer 1

To see how this all fits together, we will consider a concrete example (Listing 11.1).

```
// Configure Timer 0 as a 16-bit timer
TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

ET0 = 0; // No interrupts

TH0 = 0; // Timer 0 initial value (High Byte)
TL0 = 0; // Timer 0 initial value (Low Byte)

TF0 = 0; // Clear overflow flag
TR0 = 1; // Start Timer 0

while (TF0 == 0): // Loop until Timer 0 overflows (TF0 == 1)

    TR0 = 0; // Stop Timer 0
```

Listing 11.1 Creating a simple hardware delay using Timer 0

In Listing 11.1, these lines set up Timer 0, in Mode 1 (16-bit timer), without gating:

```
TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)
```

We disable interrupt generation, as discussed earlier:

```
ET0 = 0; // No interrupts
```

We then load the timer registers with the initial timer value (we consider this further in the remainder of this pattern):

```
TH0 = 0; // Timer 0 initial value (High Byte)
TL0 = 0; // Timer 0 initial value (Low Byte)
```

Then we are ready to clear the timer flag, and start the timer running:

```
TF0 = 0; // Clear overflow flag
TR0 = 1; // Start timer 0
```

What happens now in the original 8051 (we consider some exceptions under 'Portability') is that the timer will be incremented every 12 oscillator cycles. When

this (16-bit) timer overflows – that is, it is incremented from a value of 65535 – the timer flag (TF1) will be set. In addition, as we have already noted, this overflow can be used to generate an interrupt; we do not use this option here.

This is very useful behaviour. Simply by varying the initial value stored in the timer, we specify the number of oscillations that occur before an overflow takes place and can generate shorter delays.

Building on the material discussed under ‘Background’, **HARDWARE DELAY** calculations generally take the following form:

- We calculate the required starting value for the timer.
- We load this value into Timer 0 or Timer 1.
- We start the timer.
- The timer will be incremented, without software intervention, at a rate determined by the oscillator frequency; we wait for the timer to reach its maximum value and ‘roll over’.
- The rolling over of the timer signals the end of the delay by changing the value of a flag variable.
- With a 12-oscillator per instruction 8051, running at 12 MHz, the longest delay that can be produced with a 16-bit timer is ~65 ms. If we need longer delays, we can repeat the process.

Overall, this is a powerful and reliable technique and can be used to generate repeatable delays with good levels of accuracy. A detailed code example is presented later in this pattern.

Why not use Timer 2?

In many cases, as we saw in Chapter 3, modern 8051 family devices are based on the slightly later 8052 architecture: such devices include an extra, more powerful timer (called, logically, Timer 2).

While Timer 2 can be used to generate delays (in a manner nearly identical to that used with Timer 0 and Timer 1 in this pattern), this is generally an inappropriate use for this resource. This is because Timer 2 is a 16-bit *auto-reload* timer. This auto-reload feature has no value in the generation of delays, but makes it ideally suited as a source of ‘ticks’ for the schedulers we use throughout most of this book.²

Hardware resource implications

HARDWARE DELAY requires non-exclusive use of a timer. There are often competing demands for such resources, since they are essential for driving a scheduler and are often used, for example, to generate timeouts (see **HARDWARE TIMEOUT** [page 305]), for pulse-width modulation (see **3-LEVEL PWM** [page 822]), for pulse-rate modulation

2. See Chapter 13 and **CO-OPERATIVE SCHEDULER** [page 255] for further information on this topic.

(see **HARDWARE PRM** [page 742]) and for pulse counting (see **HARDWARE PULSE-COUNT** [page 728]).

Reliability and safety implications

The techniques discussed in **HARDWARE DELAY** are generally more portable and more accurate than software-based delays, but they are still not suitable for generating precisely timed delays without careful hand-tuning to take into account factors such as the time taken to call the delay function and the time taken to load the timer with the initial count value.

Delays generated through multiple calls to a delay function will increase the impact of these factors; long delays generated using this technique are likely to be particularly inaccurate.

Portability

We consider two main portability issues here.

Differences in timer increment rates

In the original 8051 (and in most current 8051s), Timer 0 and Timer 1 are incremented every 12 oscillator cycles: that is, at 1 MHz in a device using a 12 MHz crystal oscillator. In more recent 8051 devices, factors of 6, 4 or 1 are also used. You must check the data sheet to ensure that your calculations of the initial reload values take these differences into account.

Note that the library code (listed later) is itself highly portable, because it makes use of information provided in the **PROJECT HEADER** [page 169] file.

Porting within the 8051 family

There are limited timers available in the 8051 family. Careful use of particular timers can help make your code easier to port.

For example, in many applications presented in this book, we will use Timer 2 (where available) to drive a scheduler (see, for example, Chapter 13). In addition, in some applications, Timer 1 will be required to generate baud rates, for a serial network. As a result, your delay code will be particularly portable if you base it on Timer 0.

Note that, where you will use a scheduler and a serial link and your chosen microcontroller does not have Timer 2 available, you may need to use both the available timers for the scheduler (T0) and baud rate generation (T1). You will then be forced to use an implementation of **SOFTWARE DELAY** [page 179] for any necessary delay generation. Note also that, in many cases, the use of a scheduler can remove the need for most delay calculations.

Finally, note that, in addition to Timer 2, some extended 8051s have an additional internal timer, intended for use as an internal baud rate generator: this can free Timer 1 for other purposes, such as delay generation. (See Chapter 3 for details.)

Porting beyond the 8051 family

Like the 8051 family, most microcontrollers have on-board timers: where such timers are available, this pattern may be adapted without great difficulty. If your chosen microcontroller does not have an on-board timer, the pattern **SOFTWARE DELAY** [page 206] offers an alternative solution.

Overall strengths and weaknesses

- ☺ These basic time delay techniques have the great advantage that they are very simple and can be implemented in a few lines of code. As a result, they are widely applicable and are frequently used in applications where very accurate timing is not of great concern.
- ☹ They are not suitable for generating very short delays; see ‘Related patterns and alternative solutions’ for alternative suggestions.
- ☹ Because of the need to manually reload the initial timer value, the delays obtained may not be precisely as expected. This is of particular concern where, for example, an attempt is made to delay for (say) a second by invoking a 50 ms delay 20 times: this will **not** be accurate. Do not attempt to use **HARDWARE DELAY** to implement a real-time clock!
- ☹ They require access to an important hardware resource (a timer).
- ☹ The timings are not very portable: even different members of the 8051 family have different relationships between crystal frequency and instruction cycle frequency (note, however, that the code that follows addresses this problem for a wide range of delays).
- ☹ As implemented here, the processor is tied up waiting for the timer to overflow. Where processor power is limited, this may not be an acceptable solution: use of a scheduler (see Chapter 13) can often reduce the need to waste CPU time in this way.

Related patterns and alternative solutions

The code we present in this pattern is designed to generate delays on N millisecond duration, which is a key requirement in many applications. If this is not what you require, then some alternatives are as follows:

- To generate delays from ~10 µs to about ~10+ ms, **HARDWARE TIMEOUT** [page 305] can be used very effectively.
- For generating delays less than ~10 µs, neither **HARDWARE DELAY** nor **HARDWARE TIMEOUT** – both of which use Timer 0 / Timer 1 – is suitable. For example, in an original 12 MHz 8051, the minimum timer increment is, in theory, 1 µs: that is the oscillator frequency / 12. However, this delay will often be less than the time taken to call the delay function, set up and start the timer. In general, delays less than

around 10 µs are better implemented in software: see **SOFTWARE DELAY** [page 206] (and **LOOP TIMEOUT** [page 298]).

- Delays waste CPU time and it is better to avoid using them at all, if possible: see **CO-OPERATIVE SCHEDULER** [page 255] for a delay-free alternative that will work in many circumstances.

Example: Generic delay code

Flashing an LED can be useful as a means of drawing attention to a particular warning message or error condition. It can also be used as a means of saving power. There are, of course, numerous different ways of implementing such behaviour: here, we illustrate the use of the **HARDWARE DELAY** pattern.

Hardware

The single LED (or a similar device, such as a buzzer) is assumed to be connected to an 8051 microcontroller on Port 1 (P1.2), using positive logic: that is, +5V lights the LED (Figure 11.1).

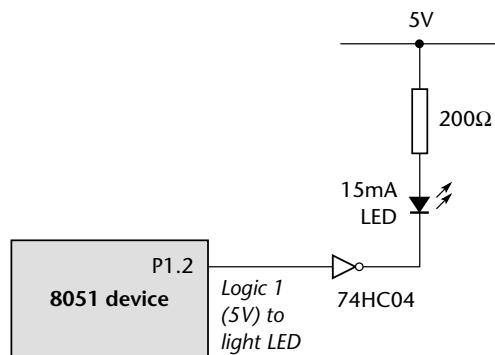


FIGURE 11.1 The LED hardware

[Note: See **IC BUFFER** [page 118] for further details.]

Software

Here we use some generic delay code. This allows the initial timer values to be ‘automatically’ determined for a wide range of different hardware and oscillator combinations, by means of the project header file (**Main.H**), and some appropriate use of the C pre-processor directives.

The key files required in the project follow (Listings 11.2 to 11.4). As usual complete set of files are included on the CD.

```
/*-----*
Main.H (v1.00)

-----'
'Project Header' (see Chap 9) for project DELAY_H (see Chap 11)
-----*/
#ifndef _MAIN_H
#define _MAIN_H

//-----
// WILL NEED TO EDIT THIS SECTION FOR EVERY PROJECT
//-----

// Must include the appropriate microcontroller header file here
#include <reg52.h>

// Include oscillator / chip details here
// (essential if generic delays / timeouts are used)
// -
// Oscillator / resonator frequency (in Hz) e.g. (11059200UL)
#define OSC_FREQ (120000000UL)

// Number of oscillations per instruction (4, 6 or 12)
// 12 - Original 8051 / 8052 and numerous modern versions
// 6 - Various Infineon and Philips devices, etc.
// 4 - Dallas, etc.
//
// Take care with Dallas devices
// - Timers default to *12* osc ticks unless CKCON is modified
// - If using generic code on a Dallas device, use 12 here
#define OSC_PER_INST (12)

//-----
// SHOULD NOT NEED TO EDIT THE SECTIONS BELOW
//-----

typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;

// Misc #defines
#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif
```

```
#define RETURN_NORMAL (bit) 0
#define RETURN_ERROR (bit) 1

//-----
// Interrupts
// - see Chapter 13.
//-----

// Generic 8051 timer interrupts (used in most schedulers)
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5

// Additional interrupts (used in shared-clock schedulers)
#define INTERRUPT_UART_Rx_Tx 4
#define INTERRUPT_CAN_c515c 17

//-----
// Error codes
// - see Chapter 14.
//-----

#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
/*-
--- END OF FILE ---
*/
```

Listing 11.2 Part of the generic delay code (Hardware Delay) example

```

/* -----
Main.C (v1.00)

-----
Simple test program for hardware delay library.

-*----- */

#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);
    }
}

/* -----
-- END OF FILE --
-*----- */

```

Listing 11.3 Part of the generic delay code (Hardware Delay) example

```

/* -----
Delay_T0.C (v1.00)

-----
Simple hardware delays based on T0.

-*----- */

#include "Main.H"

// ----- Private constants -----
// Timer preload values for use in simple (hardware) delays
// - Timers are 16-bit, manual reload ('one shot').

// NOTE: These values are portable but timings are *approximate*
//       and *must* be checked by hand if accurate timing is required.
//

```

```

// Define Timer 0 / Timer 1 reload values for ~1 msec delay
// NOTE: Adjustment made to allow for function call overhead etc.
#define PRELOAD01 (65536 - (tWord)(OSC_FREQ / (OSC_PER_INST * 1063)))
#define PRELOAD01H (PRELOAD01 / 256)
#define PRELOAD01L (PRELOAD01 % 256)

/*-----*
Hardware_Delay_T0()

Function to generate N millisecond delay (approx).

Uses Timer 0 (easily adapted to Timer 1).

*-----*/
void Hardware_Delay_T0(const tWord N)
{
    tWord ms;

    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Delay value is *approximately* 1 ms per loop
    for (ms = 0; ms < N; ms++)
    {
        TH0 = PRELOAD01H;
        TL0 = PRELOAD01L;

        TF0 = 0; // clear overflow flag
        TR0 = 1; // start timer 0

        while (TF0 == 0); // Loop until Timer 0 overflows (TF0 == 1)

        TR0 = 0; // Stop Timer 0
    }
}

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 11.4 Part of the generic delay code (Hardware Delay) example

The output from this project is shown running in the Keil hardware simulator in Figure 11.2. Note that the delay value obtained is within 0.001% of the required value.

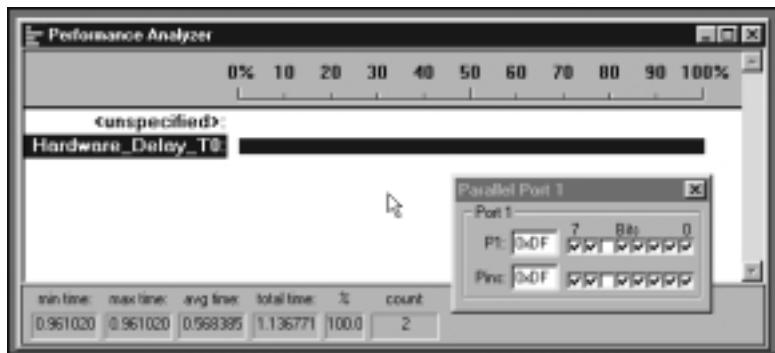


FIGURE 11.2 Output from the Hardware Delay example running in the Keil hardware simulator

Further reading

SOFTWARE DELAY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

How do you create a simple delay without using any hardware (timer) resources?

Background

See **HARDWARE DELAY** [page 194] for background information.

Solution

Suppose we want to flash the LEDs connected to Port 1 on an 8051 microcontroller with a two second cycle time (so that they are on for 1 second then off for 1 second, *ad infinitum*). The basic program structure we require could be based on a **SUPER LOOP** [page 162] as follows:

```
...
while (1)
{
    P1 = 0xFF;
    // Delay one second
    P1 = 0x00;
    // Delay one second
}
```

If we wish to implement these delays and have no available timers (see **HARDWARE DELAY** [page 194]), we could use a 'Software Delay', implemented as follows:

```
Loop_Delay()
{
    unsigned int x;

    for (x=0; x <= 65535; x++);
}
```

We could then measure the pulse frequency we obtained, using an oscilloscope or a software simulator. If we found that these delays were not long enough, we could easily extend them by adding additional layers, as shown in `Longer_Loop_Delay()`

```

Longer_Loop_Delay()
{
    unsigned int x;
    unsigned int y;

    for (x=0; x<=65535; x++)
    {
        for (y=0; y<=65535; y++);
    }
}

```

Hardware resource implications

Unlike **HARDWARE DELAY** [page 194], **SOFTWARE DELAY** uses no timer resources. Note, however, that the CPU time spent in the delay calculation is wasted: using a scheduler (see Chapter 13) can, in many circumstances, avoid the waste of CPU time.

Reliability and safety implications

Software delays are not suitable for use in applications where precise timing is required.

Portability

Software delays can be used even on a microcontroller / microprocessor without a built-in timer. However, the precise delay duration obtained varies (enormously) with differences in hardware and software.

Overall strengths and weaknesses

- ☺ **SOFTWARE DELAY** can be used to produce very short delays.
- ☺ **SOFTWARE DELAY** requires no hardware timers.
- ☺ **SOFTWARE DELAY** will work on any microcontroller.
- ☹ It is very difficult to produce precisely timed delays.
- ☹ The loops must be returned if you decide to use a different processor, change the clock frequency or even change the compiler optimization settings.

Related patterns and alternative solutions

In most circumstances, it is better to avoid using delays at all: see **CO-OPERATIVE SCHEDULER** [page 255] for a delay-free alternative that will work in many circumstances.

If you do require delays then **HARDWARE DELAY** [page 194] is often a better alternative.

Example: Creating a 5 μ s delay in an I²C library

As we discuss in Chapter 23, software implementation of the I²C serial protocol can require small delays. In Listing 11.5 we illustrate how a delay of around 5 μ s can be created in software for use in such a library.

```
/* -----
   _I2C_Delay()
A short software delay.

Adjust this for a minimum of 5.425  $\mu$ s to work with
'standard' I2C devices. Any delay longer than this will also work.
With modern devices shorter delays may also be used.

NOTE: Cannot do this with a Hardware Delay!!!

----- */
void _I2C_Delay(void)
{
    int x;
    x++;
    x++;
}
```

Listing 11.5 Creating a very short delay

Example: Flashing an LED

We will repeat the ‘flashing LED’ example used in **HARDWARE DELAY** [Page 194] to illustrate the **SOFTWARE DELAY** pattern.

To control the flashing of the LED, we will use an endless loop involving one **SOFTWARE DELAY** and a ‘flash LED’ function.

We assume that we are using a Standard 8051 device with 12 oscillations per instruction cycle and an oscillator frequency of 12 MHz. (see Chapter 4 for further details.)

The key files required in the project follow (Listings 11.6 to 11.11). As usual complete set of files are included on the CD.

```
/* -----
Main.C (v1.00)

-----
Simple test program for SOFTWARE DELAY pattern.

----- */

```

```

#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);
    }
}

/* -----
   ---- END OF FILE -----
*/
```

Listing 11.6 Part of the Software Delay (flashing LED) example

```

/* -----
   Loop_Del.H (v1.00)

-----
- See Loop_Del.C for details.

*/
void Loop_delay(const unsigned int);

/* -----
   ---- END OF FILE -----
*/
```

Listing 11.7 Part of the Software Delay (flashing LED) example

```

/* -----
   Loop_Del.c (v1.00)

-----
Implementation of the pattern SOFTWARE DELAY.

*/
void Loop_Delay(const unsigned int);
```

```

/*-----*/
Loop_Delay()

Delay duration varies with parameter.

Parameter is, *ROUGHLY*, the delay, in milliseconds,
on 12MHz 8051 (12 osc cycles).

You need to adjust the timing for your application!

-*-----*/
void Loop_Delay(const unsigned int DELAY)
{
    unsigned int x, y;

    for (x = 0; x <= DELAY; x++)
    {
        for (y = 0; y <= 120; y++);
    }
}

/*-----*/
----- END OF FILE -----
-*-----*/

```

Listing 11.8 Part of the Software Delay (flashing LED) example

```

/*-----*/
LED_flas.H (v1.00)

-----*/

- See LED_flas.C for details.

-*-----*/
//----- Public function prototypes -----
void LED_Flash_Init(void);
void LED_Flash_Update(void);

/*-----*/
----- END OF FILE -----
-*-----*/

```

Listing 11.9 Part of the Software Delay (flashing LED) example

```
/*-----*
 LED_flas.C (v1.00)

-----
 Simple 'Flash LED' test function.

-*-----*/
#include "Main.h"
#include "LED_flas.h"

// ----- Port pins -----
// Connect LED from +5V (etc) to this pin, via appropriate resistor
// [see Chapter 7 for details]
sbit LED_Pin = P1^2;

// ----- Public variable definitions -----
static bit LED_state_G;

/*-----*
 LED_Flash_Init()
 - See below.

-*-----*/
void LED_Flash_Init(void)
{
    LED_state_G = 0;
}

/*-----*
 LED_Flash_Update()
 Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

Must schedule at twice the required flash rate: thus, for 1 Hz
flash (on for 0.5 seconds, off for 0.5 seconds) must schedule
at 2 Hz.

-*-----*/
void LED_Flash_Update(void)
{
    // Change the LED from OFF to ON (or vice versa)
    if (LED_state_G == 1)
    {
```

```

LED_state_G 0;
LED_pin = 0;
}
else
{
LED_state_G = 1;
LED_pin = 1;
}
}

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 11.10 Part of the Software Delay (flashing LED) example

```

/*-----*
Main.H (v1.00)

-----*
'Project Header' (see Chap 9) for project S_Delay
*-----*/
#ifndef _MAIN_H
#define _MAIN_H

//-----
// WILL NEED TO EDIT THIS SECTION FOR EVERY PROJECT
//-----

// Must include the appropriate microcontroller header file here
#include <reg52.h>

// Include oscillator / chip details here
// (essential if generic delays / timeouts are used)
// -
// Oscillator / resonator frequency (in Hz) e.g. (11059200UL)
#define OSC_FREQ (12000000UL)

// Number of oscillations per instruction (e.g. 1, 4, 6 or 12)
// 12 - Original 8051 / 8052 and numerous modern versions
// 6 - Various Infineon and Philips devices, etc.
// 4 - Dallas, etc.
//
// Take care with Dallas devices

```

```
// - Timers default to *12* osc ticks unless CKCON is modified
// - If using generic code on a Dallas device, use 12 here
#define OSC_PER_INST (12)

//-----
// SHOULD NOT NEED TO EDIT THE SECTIONS BELOW
//-----

typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;

// Misc #defines
#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif

#define RETURN_NORMAL (bit) 0
#define RETURN_ERROR (bit) 1

//-----
// Interrupts
// - see Chapter 12.
//-----

// Generic 8051 timer interrupts (used in most schedulers)
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5

// Additional interrupts (used in shared-clock schedulers)
#define INTERRUPT_UART_Rx_Tx 4
#define INTERRUPT_CAN_c515c 17

//-----
// Error codes
// - see Chapter 13.
//-----

#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
/* -----
----- END OF FILE -----
----- */
```

Listing 11.11 Part of the Software Delay (flashing LED) example

Figure 11.3 shows this application running in the Keil hardware simulator. As can be seen from the screen shot, the delays are approximately 1 second with these parameters and compiler settings.

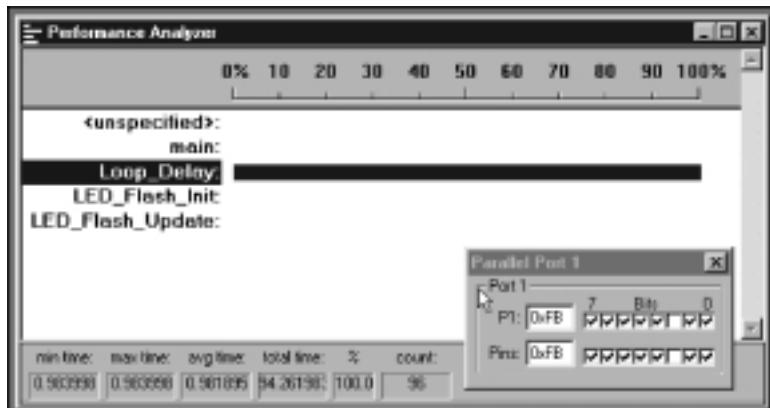


FIGURE 11.3 Output from the Software Delay (flashing LED) example running on the Keil hardware simulator

Further reading

chapter **12**

Watchdogs

Introduction

Suppose there is a hungry dog guarding a house (Figure 12.1), and someone wishes to break in. If the burglar's accomplice repeatedly throws the guard dog small pieces of meat at 2-minute intervals, then the dog will be so busy concentrating on the food that he will ignore his guard duties and will not bark. However, if the accomplice runs out of meat or forgets to feed the dog for some other reason, the animal will start barking, thereby alerting the neighbours, property occupants or police.



FIGURE 12.1 The origins of the 'watchdog' analogy

This same basic approach is followed in computerized ‘watchdog timers’. Very simply, these are timers which, if not refreshed at regular intervals, will overflow. In most cases, overflow of the timer will reset the system. Such watchdogs are intended to deal with the fact that, even with meticulous planning and careful design, embedded systems can ‘hang’ due to unexpected problems. The use of a watchdog can be used to recover from this situation, in certain circumstances.

The pattern **HARDWARE WATCHDOG** [page 217] considers how to apply these techniques to good effect in your embedded application.

HARDWARE WATCHDOG

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

How can you ensure that – if your application ‘hangs’ due to an unexpected software or hardware error – the system will automatically reset itself?

Background

See the introduction to this chapter for an explanation of the watchdog analogy.

Solution

Working with **HARDWARE WATCHDOG** means using either an internal or external (hardware) timer.

We have seen in many previous cases that, where available, the use of on-chip components is to be preferred to the use of equivalent off-chip components. Specifically, on-chip components generally offer the following benefits:

- Reduced hardware complexity, which tends to result in increased system reliability.
- Reduced application cost.
- Reduced application size.

In the case of watchdog timers, the situation is more complex, because external watchdog chips typically provide some useful facilities that are not available in most on-chip versions.

For example, the popular ‘1232’ watchdogs (available, in various versions, from Dallas Semiconductors, Maxim, Linear Technology and Analog Devices) are low-cost, low-power devices. In addition to functioning as a watchdog timer, they also provide power system monitoring capabilities (see **ROBUST RESET** [page 77] for details of this). If, as in many designs, you intend to use an external ‘robust reset’ circuit anyway, then the 1232 chips allow you to incorporate an external watchdog facility for minimal addition cost and only a very minor increase in hardware complexity.

Another beneficial feature of external watchdogs is that they are inherently portable: you can generally use the same external watchdog with any member of the 8051 family. By contrast, code written to work with an internal watchdog will generally have to be rewritten for use with a different hardware.

One situation in which on-chip watchdogs (such as those in the Infineon c515x devices) can be beneficial is where they allow you to determine whether the system has undergone a normal reset or a reset caused by a watchdog overflow. This may allow you to modify the system behaviour to match these circumstances. Without this information (which is not generally available through external watchdogs without some complex coding) your system may be continually reset by the watchdog timer overflow.

We can summarize by saying that, if you require watchdog facilities, you need to consider both internal and external solutions carefully. There is no single ‘ideal’ solution and – considering the issues mentioned earlier – you need to find the best match to your requirements.

Reliability and safety implications

Before using either an internal or external watchdog, you need to be sure that the use of such a timer will increase (rather than decrease) the reliability of your application.

The first thing to bear in mind is that watchdog behaviour should be for **disaster recovery**. In a well-designed system the occurrence of a watchdog reset should be a noteworthy event that occurs rarely. If you think of the use of watchdogs in terms of ‘if all else fails, then we will have to let the watchdog reset the system’, then you are taking a realistic view of the capabilities of this approach.

Used without due care at the design phase and/or adequate testing, watchdogs can reduce the system reliability dramatically. A particular problem with a badly designed watchdog can occur in the presence of sustained hardware faults. In these circumstances, a badly implemented watchdog can mean that your system constantly resets itself. This can be extremely dangerous.

You also need to appreciate that watchdogs are unsuitable for many applications, because the time taken to react to an error is too long. Suppose, for example, the braking system in an automotive application uses a 500 ms watchdog and the vehicle encounters a problem when it is travelling at 70 miles per hour (110 km per hour). In these circumstances, the vehicle and its passengers will have travelled some 16 yards / 15 metres – right into the car in front – before the vehicle even begins to reset the braking system. In short, where fast recovery is required, watchdogs are rarely the best solution.

Portability

As already noted, internal watchdogs are based on hardware that is not part of the 8051/52 core. As a result, different forms of watchdog now exist on the various different 8051 derivatives and code written for one on-chip watchdog will generally need to be adapted for use with a different device. By contrast, software written for external watchdogs can be more portable.

Overall strengths and weaknesses

- ☺ Watchdogs can provide a 'last resort' form of error recovery. If you think of the use of watchdogs in terms of 'if all else fails, then reset the system', then you are taking a realistic view of the capabilities of this approach.
- ☺ In the presence of intermittent faults, e.g. rare bursts of EMI, watchdogs can be very effective.
- ☺ Watchdogs with long timeout periods are unsuitable for many applications.
- ☺ Used without due care at the design phase and / or adequate testing, watchdogs can reduce the system reliability dramatically.
- ☺ In the presence of sustained hardware faults, badly implemented watchdogs can mean that your system constantly resets itself. This can be very dangerous.

Related patterns and alternative solutions

In certain restricted circumstances, a software watchdog may also be useful.

This can be created from two components:

- A timer ISR
- A refresh function

Essentially, we set a timer to overflow in (say) 60 ms. Under normal circumstances, this timer will never overflow, because we will call the refresh function regularly and thereby restart the timer. If, however, the program is 'jammed', the refresh function will not be called. When the timer overflows, the ISR will be called: this will implement an 'appropriate' error recovery strategy.

We have used software watchdogs in several applications. The main problem with this approach is that some software errors (for example, those induced by EMI) can disrupt the watchdog timer as well as the main application code: this rarely happens with hardware watchdogs, which tend to be more robust.

The main advantage with software watchdogs is that different forms of error recovery (not just a complete chip reset) are possible. However, use of an on-chip hardware watchdog can provide flexible reset behaviour and is, in many circumstances, a more reliable solution.

Example: Using the '1232' external watchdog timer

In this example we assume that we will be developing a simple central-heating control system and will be using an external '1232' watchdog chip to improve the reliability of the application.

The use of the 1232 is very straightforward:

- We wire up the watchdog to the microcontroller reset pin, as illustrated in Figure 12.2.

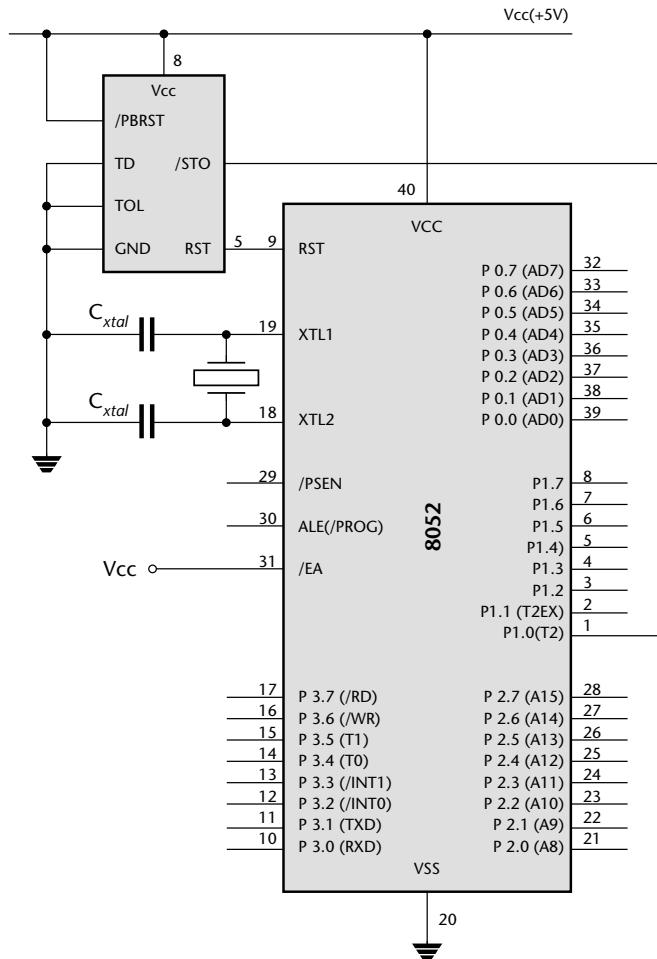


FIGURE 12.2 Simple demonstration circuit for 1232 watchdog

- We choose from one of three (nominal) possible timeout periods, and connect the TD pin on the 1232 to select an appropriate period (see Table 12.1).
- We pulse the ST line on the 1232 regularly, with a pulse interval less than the timeout period.

TABLE 12.1 Timings for the ubiquitous '1232' watchdog

	Minimum timeout	Typical timeout	Maximum timeout
TD to GND	62.5 ms	150 ms	250 ms
TD floating	250 ms	600 ms	1000 ms
TD to Vcc	500 ms	1200 ms	2000 ms

A code example, using the hardware in Figure 12.2 (with a 150 ms timeout), is given in Listings 12.1 to 12.2.

```
/*-----*  
Main.C (v1.00)  
-----*  
  
Framework for a central heating system using 'Super Loop'.  
*** Assumes external '1232' watchdog timer on P1^0 ***  
-*-----*/  
  
#include "Cen_Heat.h"  
#include "Dog_1232.h"  
  
/*-----*/  
void main(void)  
{  
    // Init the system  
    C_HEAT_Init();  
  
    // Watchdog automatically starts  
  
    while(1)  
    {  
        // Find out what temperature the user requires  
        // (via the user interface)  
        C_HEAT_Get_Required_Temperature();  
  
        // Find out what the current room temperature is  
        // (via temperature sensor)  
        C_HEAT_Get_Actual_Temperature();  
  
        // Adjust the gas burner, as required  
        C_HEAT_Control_Boiler();  
  
        // Feed the watchdog  
        WATCHDOG_Feed();  
    }  
}  
/*-----*  
----- END OF FILE -----  
-*-----*/
```

Listing 12.1 Part of a central-heating demo using Super Loop and Hardware Watchdog

```

/* -----
Dog_1232.C (v1.00)

-----
Watchdog timer library for external 1232 WD.

----- */

#include "Dog_1232.h"

#include "Main.h"

// ----- Port pins -----

// Connect 1232 (pin /ST) to the WATCHDOG_pin
sbit WATCHDOG_pin = P1^0;

/* -----
WATCHDOG_Feed()

'Feed' the external 1232-type watchdog chip.

----- */

void WATCHDOG_Feed(void)
{
    static bit WATCHDOG_state;

    // Change the state of the watchdog pin
    if (WATCHDOG_state == 1)
    {
        WATCHDOG_state = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state = 1;
        WATCHDOG_pin = 1;
    }
}

/* -----
----- END OF FILE -----
----- */

```

Listing 12.2 Part of a central-heating demo using Super Loop and Hardware Watchdog

Example: Using the internal watchdog timer on the Atmel 89S53

The Atmel 89S53 is an example of a Standard 8051 microcontroller with a good on-chip watchdog timer.

A key feature of this timer is that it operates from an independent oscillator: as a result, it allows the system to respond to (intermittent) failures of the main crystal oscillator or resonator.

The key register used to control the watchdog timer is the WCON register, shown in Table 12.2.

TABLE 12.2 The WCON SFR used to control the on-chip watchdog timer in the Atmel AT89S53

Bit	7	6	5	4	3	2	1	0
Name	PS2	PS1	PS0	NOT WD	NOT WD	NOT WD	WDTRST	WDTEN

The role of the individual bits in WCON is explained in Table 12.3.

TABLE 12.3 A more detailed look at the WCON SFR in the Atmel AT89S53

Symbol	Function
PS2	Prescaler Bits for the Watchdog Timer. When all three bits are set to '0', the watchdog timer has a nominal period of 16 ms. When all three bits are set to '1', the nominal period is 2048 ms. See Table 12.4 for more detailed information. <i>Note:</i> Actual timings can vary by as much as $\pm 30\%$ of the nominal values.
WDTRST	Watchdog Timer Reset. Each time this bit is set to '1' by user software, a pulse is generated to reset the watchdog timer. The WDTRST bit is then automatically reset to '0' in the next instruction cycle. The WDTRST bit is Write-Only.
WDTEN	Watchdog Timer Enable Bit. WDTEN = 1 enables the watchdog timer and WDTEN = 0 disables the watchdog timer.

The prescaler bits, PS0, PS1 and PS2 in SFR WCON are used to set the period of the Watchdog Timer from 16 ms to 2048 ms. The available timer periods are shown in Table 12.4 and the actual timer periods (at Vcc = 5V) are within $\pm 30\%$ of the nominal.

The WDT is disabled by power-on reset and during power-down. It is enabled by setting the WDTEN bit in SFR WCON (address = 96H). The WDT is reset by setting the WDTRST bit in WCON. When the WDT times out without being reset or disabled, an internal RST pulse is generated to reset the CPU.

Listings 12.3 to 12.7 show how we might use this watchdog in the simple central-heating system discussed in the previous example.

TABLE 12.4 Watchdog timer period selection in the AT89S53

PS2	PS1	PS0	Period (nominal)
0	0	0	16 ms
0	0	1	32 ms
0	1	0	64 ms
0	1	1	128 ms
1	0	0	256 ms
1	0	1	512 ms
1	1	0	1024 ms
1	1	1	2048 ms

```
/*-----*
Main.C (v1.00)

-----*
Central heating demo using 'Super Loop' and 'Hardware Watchdog'.
[Compiles and runs but does nothing useful]
-*-----*/
```

```
#include "Cen_Heat.h"
#include "Dog_AT.h"

/*-----*/
void main(void)
{
    // Init the system
    C_HEAT_Init();

    // Start the watchdog
    WATCHDOG_Init();

    while(1) // 'for ever' (Super Loop)
    {
        // Find out what temperature the user requires
        // (via the user interface)
        C_HEAT_Get_Required_Temperature();
```

```

    // Find out what the current room temperature is
    // (via temperature sensor)
    C_HEAT_Get_Actual_Temperature();

    // Adjust the gas burner, as required
    C_HEAT_Control_Boiler();

    // Feed the watchdog
    WATCHDOG_Feed();
}
}

/* -----
   ----- END OF FILE -----
   */

```

Listing 12.3 Part of a central-heating demo using Super Loop and Hardware Watchdog

```

/* -----
   Dog_AT.H (v1.00)

-----

   - see Dog_AT.C for details

   */

#include <At89S53.h>

// Function prototypes
void WATCHDOG_Init(void); // Start the watchdog
// We use a macro to feed the watchdog (for speed)
#define WATCHDOG_Feed() WMCON |= 0x02

/* -----
   ----- END OF FILE -----
   */

```

Listing 12.4 Part of a central-heating demo using Super Loop and Hardware Watchdog

```

/* -----
   Dog_AT.C (v1.00)

-----
```

Demonstration of watchdog timer facilities on Atmel 89S53.

[Compiles and runs but does nothing useful]

```
/*-----*/
#include "Dog_AT.h"
/*-----*/
void WATCHDOG_Init(void)
{
    // Set 512 ms watchdog
    // PS2 = 1; PS1 = 0; PS0 = 1
    // Set WDTRST = 1
    // Set WDTEN = 1 - start the Watchdog
    //
    // WMCON |= 10100011
    WMCON |= 0xA3;
}
/*-----*
----- END OF FILE -----
*-----*/
```

Listing 12.5 Part of a central-heating demo using Super Loop and Hardware Watchdog

```
/*-----*
Cen_Heat.H (v1.00)

-----
- see Cen_Heat.C for details.

*-----*/
// Function prototypes
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);

/*-----*
----- END OF FILE -----
*-----*/
```

Listing 12.6 Part of a central-heating demo using Super Loop and Hardware Watchdog

```
/*-----*  
Cen_Heat.C (v1.00)  
-----  
Framework for a central heating system using 'Super Loop'.  
[Compiles and runs but does nothing useful]  
-----*/  
/*-----*/  
  
void_C_HEAT_Init(void)  
{  
    // User code here ...  
}  
/*-----*/  
  
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 ...  
}  
/*-----*  
----- END OF FILE -----  
-----*/
```

Listing 12.7 Part of a central-heating demo using Super Loop and Hardware Watchdog

Further reading



Time-triggered architectures for single-processor systems

This book is primarily concerned with the creation of time-triggered embedded systems. Having laid the foundations, we are now in a position to consider in detail the ways in which time-triggered software architectures may be created, in 'C', for the 8051 family.

In Chapter 13, we introduce schedulers and explain how they may be used to create efficient time-triggered applications.

In Chapter 14, we describe, in detail, the construction of **CO-OPERATIVE SCHEDULER** [page 255]. This simple but flexible environment is suitable for use with a very wide range of embedded applications.

Using a co-operative scheduler in your application has a number of benefits, one of which being that the development process is simplified. However, to get the maximum benefit from the scheduler you need to approach the design in a slightly different ('task oriented') way. We discuss how to do this in Chapter 15 and Chapter 16.

Finally, in Chapter 17, we present **HYBRID SCHEDULER** [page 333]. Like all of the scheduler patterns in this book, **HYBRID SCHEDULER** is based on a co-operative scheduler; however, this version is also able to support a single pre-emptive task.

chapter **13**

An introduction to schedulers

In this chapter, we see that schedulers can play a similar role in embedded systems to that played by ‘Windows’ (or other operating systems) in many modern desktop applications.

13.1 Introduction

Having laid the foundations in Parts A and B, we are now in a position to look in detail at the ways in which time-triggered applications may be created with the 8051 family of microcontrollers.

To produce such applications, we will use a scheduler: this is a very simple operating environment for embedded applications. In this introductory chapter, we explain what a scheduler is and the differences between co-operative and pre-emptive scheduling. We will also explain why the use of a co-operative scheduler can help to make even the smallest of embedded applications easier to develop and more reliable in operation.

To place the discussions in the rest of the chapter in context, we begin by briefly reviewing the reasons why desktop systems employ an operating system and explaining why such an OS is not appropriate for use with the type of embedded systems considered in this book.

13.2 The desktop OS

As stated in the preface, it is assumed in this book that readers will have had previous experience of software development for desktop computer systems. As we discussed in Chapter 1, the desktop / workstation environment plays host to many information systems, as well as general-purpose desktop applications, such as word processors. A

common characteristic of modern desktop environments is that the user interacts with the application through a high-resolution graphics screen, plus a keyboard and a mouse. Support for this complex user interface is provided by the operating system and its associated libraries.

In such an environment, the program the user requires (such as a word processor) is usually loaded from disk on demand, along with any required data (such as a word processor file). Figure 13.1 shows a typical operating environment for such a word processor. Here the application is well insulated from the underlying hardware. For example, when the user wished to save her latest novel on disk, the word processor delegates most of the necessary work to the operating system, which in turn may delegate many of the hardware-specific commands to the BIOS (basic input/output system).

The desktop PC does not *require* an operating system (or BIOS). However, for most users, the main advantage of a personal computer is its flexibility: that is, that the same piece of equipment has the potential to run many thousands of different programs. If the PC had no operating system, each of these programs would need to be able to carry out all the low-level functions for itself. This would be very inefficient and would tend to make applications more expensive. It would also be likely to lead to errors, as many functions would have to be duplicated in even the smallest of programs.

We can get a feel for the type of problems that would result in a world without Windows (or UNIX) if we consider 'DOS', an early operating system widely used on PCs. Readers old enough to have used DOS applications will remember that *every* program needed to provide a suitable printer driver: if the printer was subsequently changed, this generally meant that every application on the PC needed to be upgraded in order to take advantage of the new hardware. With Windows, this problem does not arise: when a new printer is purchased, a single driver is required. When this had been installed, every program on the computer can immediately make use of the new hardware.

One way of viewing this is that a desktop operating system is used to run multiple programs, and the operating systems provides the 'common code' (for printing, file storage, graphics, and so forth) that is required by this set of programs: this reduces the need to duplicate identical program components, reducing the opportunity for errors and making the overall system more reliable and easier to maintain.

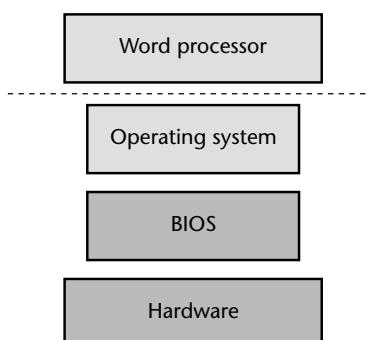


FIGURE 13.1 A schematic representation of the BIOS/OS sandwich from a desk-bound computer system running some word processor software

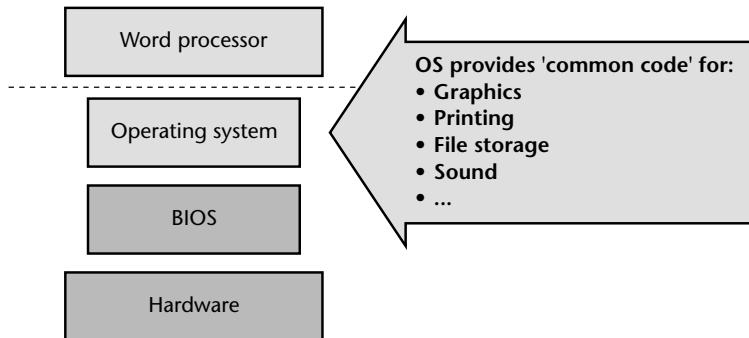


FIGURE 13.2 A schematic representation of the role played by the operating system in a desktop application

13.3 Assessing the Super Loop architecture

Many of the features of the modern desktop OS, such as graphics capability, printing and disk access, are of little value in embedded applications, where sophisticated graphics screens, printers and disks are unavailable.

As a result, as we saw in Chapter 9, the software architecture used in many simple embedded applications is a form of **SUPER LOOP** (Listing 13.1).

```
/* -----
Main.C
-----
Architecture of a simple Super Loop application
*/
#include "X.h"
/*
void main(void)
{
    // Prepare for Task X
    X_Init();

    while(1) // 'for ever' (Super Loop)
    {
        X(); // Perform the task
    }
}
/*
----- END OF FILE -----
*/

```

Listing 13.1 Part of a simple Super Loop demonstration

The main advantages of the Super Loop architecture illustrated in Listing 13.1 are (1) that it is simple, and therefore easy to understand, and (2) that it consumes virtually no system memory or CPU resources.

However, we get ‘nothing for nothing’: Super Loops consume little memory or processor resources because they provide few facilities to the developer. A particular limitation with this architecture is that it is very difficult to execute Task X at precise intervals of time: as we will see, this is a very significant drawback.

For example, consider a collection of requirements assembled from a range of different embedded projects (in no particular order):

- The current speed of the vehicle must be measured at 0.5 second intervals.
- The display must be refreshed 40 times every second.
- The calculated new throttle setting must be applied every 0.5 seconds.
- A time-frequency transform must be performed 20 times every second.
- If the alarm sounds, it must be switched off (for legal reasons) after 20 minutes.
- If the front door is opened, the alarm must sound in 30 seconds if the correct password is not entered in this time.
- The engine vibration data must be sampled 1,000 times per second.
- The frequency-domain data must be classified 20 times every second.
- The keypad must be scanned every 200 ms.
- The master (control) node must communicate with all other nodes (sensor nodes and sounder nodes) once per second.
- The new throttle setting must be calculated every 0.5 seconds.
- The sensors must be sampled once per second.

We can summarize this list by saying that many embedded systems must carry out tasks at particular instants of time. More specifically, we have two kinds of activity to perform:

- **Periodic tasks**, to be performed (say) once every 100 ms
- **One-shot tasks**, to be performed once after a delay of (say) 50 ms

This is very difficult to achieve with the primitive architecture shown in Listing 13.1. Suppose, for example, that we need to start Task X every 200 ms, and that the task takes 10 ms to complete. Listing 13.2 illustrates one way in which we might adapt the code in Listing 13.1 in order to try to achieve this.

```
/*-----*/
void main(void)
{
    Init_System();
    while(1) // 'for ever' (Super Loop)
    {
        X(); // Perform the task (10 ms duration)
    }
}
```

```

    Delay_190ms(); // Delay for 190 ms
}
}
}

```

Listing 13.2 Trying to use the Super Loop architecture to execute tasks at regular intervals

The approach illustrated in Listing 13.2 is not generally adequate, because it will only work if the following conditions are satisfied:

- 1 We know the precise duration of Task X
- 2 This duration never varies

In practical applications, determining the precise task duration is rarely straightforward. Suppose we have a very simple task that does not interact with the outside world but, instead, performs some internal calculations. Even under these rather restricted circumstances, changes to compiler optimization settings – even changes to an apparently unrelated part of the program – can alter the speed at which the task executes. This can make fine-tuning the timing very tedious and error prone.

The second condition is even more problematic. Often in an embedded system the task will be required to interact with the outside world in a complex way. In these circumstances the task duration will vary according to outside activities in a manner over which the programmer has very little control.

13.4 A better solution

A better solution to the problems outlined is to use timer-based interrupts as a means of invoking functions at particular times.

Timer-based interrupts and interrupt service routines

As we saw in Chapter 1, an interrupt is a hardware mechanism used to notify a processor that an ‘event’ has taken place: such events may be internal events or external events. Altogether the core 8051 / 8052 architecture supports seven interrupt sources:

- Three timer/counter interrupts (related to Timer 0, Timer 1 and – where available – Timer 2)
- Two UART-related interrupts (**note:** these share the same interrupt vector, and can be viewed as a single interrupt source)
- Two external interrupts

In addition, there is one additional interrupt source over which the programmer has minimal control:

- The ‘power-on reset’ (POR) interrupt

When an interrupt is generated, the processor ‘jumps’ to an address at the bottom of the CODE memory area. These locations must contain suitable code with which the microcontroller can respond to the interrupt or, more commonly, the locations will include another ‘jump’ instruction, giving the address of suitable ‘interrupt service routine’ located elsewhere in (CODE) memory.

While the process of handling interrupts may seem rather complicated, creating interrupt service routines (ISRs) in a high-level language is a straightforward process, as illustrated in Listing 13.3.

```
/*-----*
Main.c
-----*

Simple timer ISR demonstration program
*/
#include <AT89x52.h>

#define INTERRUPT_Timer_2_Overflow 5

// Function prototype
// NOTE: ISR is not explicitly called and does not require a prototype
void Timer_2_Init(void);

/* ----- */
void main(void)
{
    Timer_2_Init(); // Set up Timer 2

    EA = 1;          // Globally enable interrupts

    while(1);        // An empty Super Loop
}

/* ----- */
void Timer_2_Init(void)
{
    // Timer 2 is configured as a 16-bit timer,
    // which is automatically reloaded when it overflows
    //
    // This code (generic 8051/52) assumes a 12 MHz system osc.
    // The Timer 2 resolution is then 1.000 µs
    // (see Chapter 11 for details)
    //
    // Reload value is FC18 (hex) = 64536 (decimal)
    // Timer (16-bit) overflows when it reaches 65536 (decimal)
    // Thus, with these setting, timer will overflow every 1 ms
```

```

T2CON    = 0x04; // Load Timer 2 control register
T2MOD    = 0x00; // Load Timer 2 mode register

TH2      = 0xFC; // Load Timer 2 high byte
RCAP2H   = 0xFC; // Load Timer 2 reload capt. reg. high byte
TL2      = 0x18; // Load Timer 2 low byte
RCAP2L   = 0x18; // Load Timer 2 reload capt. reg. low byte

// Timer 2 interrupt is enabled, and ISR will be called
// whenever the timer overflows - see below.

ET2      = 1;

// Start Timer 2 running
TR2 = 1;
}

/* -----
void X(void) interrupt INTERRUPT_Timer_2_Overflow
{
// This ISR is called every 1 ms

// Place required code here...
}

/* -----
---- END OF FILE -----
*/

```

Listing 13.3 The framework of an application using a timer ISR to invoke a regular task

The result of running the program shown in Listing 13.3 in the Keil hardware simulator is shown in Figure 13.3.

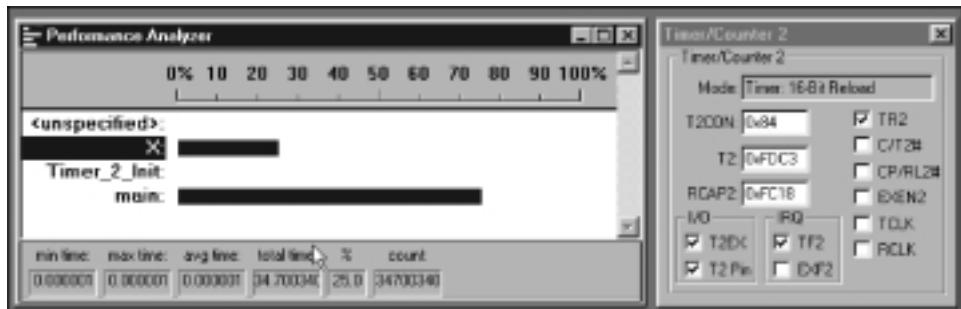


FIGURE 13.3 The result of running the program shown in Listing 13.3 in the Keil hardware simulator

Much of Listing 13.3 should be familiar. The code to set up Timer 2 in the function `Timer_2_Init()` is the same as the delay code discussed in Chapter 11, the two main differences being that, in this case:

- 1 The timer will generate an interrupt when it overflows
- 2 The timer will be automatically reloaded, and will immediately begin counting again

We discuss both of these differences in the following subsections.

The interrupt service routine (ISR)

The interrupt generated by the overflow of Timer 2, invokes the ISR called, here, `X()`.

```
/* -----
void X(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // This ISR is called every 1 ms
    // Place required code here...
}
```

The link between this function and the timer overflow is made using the Keil keyword `interrupt` (included after the function header in the function definition):

```
void X(void) interrupt INTERRUPT_Timer_2_Overflow
```

plus the following `#define` directive:

```
#define INTERRUPT_Timer_2_Overflow 5
```

To understand where the '5' comes from, note that the interrupt numbers used in ISRs directly correspond to the enable bit index of the interrupt source in the 8051 IE SFR. That is, bit 0 of the IE register will be linked to a function using 'interrupt 0'. Table 13.1 shows the link between the interrupt sources and the required interrupt numbers for the original 8051/8052.

Overall, the use of interrupts linked to timer overflows is a safe and powerful technique which will be applied throughout this book.

Automatic timer reloads

As noted earlier, when Timer 2 overflows, it is automatically reloaded and immediately begins counting again. In this case, the timer is reloaded using the contents of the 'capture' registers (note that the names of these registers vary slightly between chip manufacturers):

```
RCAP2H = 0xFC; // Load Timer 2 reload capt. reg. high byte
RCAP2L = 0x18; // Load Timer 2 reload capt. reg. low byte
```

TABLE 13.1 8051 interrupt sources

Interrupt source	Address	IE index
Power-On Reset	0x00	-
External Interrupt 0	0x03	0
Timer 0 Overflow	0x0B	1
External Interrupt 1	0x13	2
Timer 1 Overflow	0x1B	3
UART Receive/Transmit	0x23	4
Timer 2 Overflow	0x2B	5

[Note: that many 8051s have further interrupt sources: refer to the manufacturer's documentation for details of the required interrupt numbers.]

This automatic reload facility ensures that the timer keeps generating the required ticks, at precisely 1 ms intervals, without any software load, and without any intervention from the user's program.

The ability to 'automatically reload' Timer 2 simplifies the use of this timer as a source of regular ticks. Note that Timer 0 and Timer 1 also have an auto-reload capability, **but only when operating as an 8-bit timer**. In most applications, an 8-bit timer can only be used to generate interrupts at intervals of around 0.25 ms (or less); this is not generally useful.

13.5 Example: Flashing an LED

The example just given is rather abstract. Here we present another example of a timer-driven interrupt service routine. In this case, we use the timer to flash an LED on and off at regular time intervals (Listing 13.4).

Note that in this application we are using Timer 1 overflows to invoke the ISR. As we discussed in Chapter 11, Timer 1 does not have a 16-bit 'auto reload' mode; as a consequence, the timer must be manually reloaded every time it overflows: the function `Timer_1_Manual_Reload()` carries out this operation.

```
/*-----*
Main.c
-----
Simple timer ISR demonstration program
- Flashes an LED
-*-----*/
// Standard '8052' device
#include <AT89x52.h>

#define INTERRUPT_Timer_1_Overflow 3

bit LED_state_G;

// Flash LED on this pin
sbit LED_pin = P1^7;

// Function prototypes
// NOTE: ISR is not explicitly called and does not require a prototype
void Timer_1_Init(void);
void Timer_1_Manual_Reload(void);
void LED_Flash_Init(void);

/* ----- */
void main(void)
{
    Timer_1_Init(); // Set up Timer 2
    LED_Flash_Init(); // Prepare to flash the LED
    EA = 1; // Globally enable interrupts

    while(1)
    {
        PCON |= 0x01; // Go to sleep (idle mode)
    }
}

/* ----- */
void Timer_1_Init(void)
{
    // Timer 1 is configured as a 16-bit timer,
    // which is manually reloaded when it overflows
    TMOD &= 0x0F; // Clear all T1 bits (T0 left unchanged)
    TMOD |= 0x10; // Set required T1 bits (T0 left unchanged)

    // Sets up timer reload values
    Timer_1_Manual_Reload();
```

```
// Interrupt Timer 1 enabled
ET1 = 1;
}

/* -----
void Timer_1_Manual_Reload(void)
{
// Stop Timer 1
TR1 = 0;

// This code (generic 8051/52) assumes a 500 KHz system osc.
// The Timer 1 resolution is then 0.000024 seconds
// (see Chapter 11 for details)
//
// We want to generate an interrupt every second:
// this takes 1.0 / 0.000024 timer increments
// i.e. 41667 timer increments (approx)
//
// Reload value (decimal) is 65536 - 41667 -> 23869 = 0x5D3D
TL1 = 0x3D;
TH1 = 0x5D;

// Start Timer 1
TR1 = 1;
}

/* -----
void LED_Flash_Init(void)
{
// Prepare to flash the LED
LED_state_G = 0;
}

/* -----
void LED_Flash_Update(void) interrupt INTERRUPT_Timer_1_Overflow
{
// Flashes an LED (or pulses a buzzer) on a specified port pin.
// Rate determined by timer settings.

// Must manually reload the timer in this version
// (Cannot perform 16-bit auto reload with Timer 0 or Timer 1)
Timer_1_Manual_Reload();

// Change the LED from OFF to ON (or vice versa)
// (Do this every second)
if (LED_state_G == 1)
```

```

    {
        LED_state_G = 0;
        LED_pin = 0;
    }
    else
    {
        LED_state_G = 1;
        LED_pin = 1;
    }
}

/* -----
----- END OF FILE -----
----- */

```

Listing 13.4 Using a timer-driven ISR to flash an LED on and off at regular time intervals

Figure 13.4 shows the program in Listing 13.4 executing in the Keil hardware simulator.

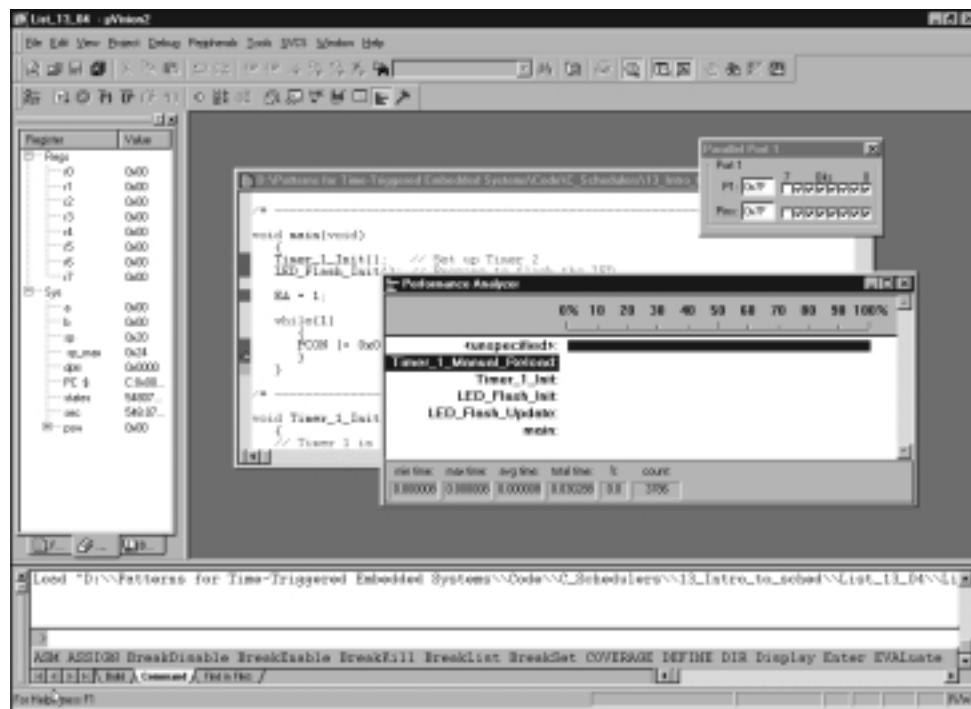


FIGURE 13.4 The program in Listing 13.4 executing in the Keil hardware simulator

13.6 Executing multiple tasks at different time intervals

While the great majority of embedded systems are required to run only one program, they do need to run multiple *tasks* (implemented as 'C' functions in this book): these tasks must, as mentioned earlier, run on a periodic or one-shot basis. These tasks will typically have different durations and will run at different time intervals. For example, we might need to read the input from an ADC every millisecond, read one or more switches every 200 milliseconds and update an LCD display every 3 milliseconds.

We can try to run more than one task by extending the technique discussed in Section 13.5. For example, suppose that we have a microcontroller device with (say) three timers available and wanted to use these timers to control the execution of three tasks, by using a separate interrupt service routine to perform each task (Listing 13.5).

```
/* -----
Main.c

-----
Multi-timer ISR program framework
----- */

#include <AT89x52.h>
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5

// Function prototypes
// NOTE: ISR is not explicitly called and does not require a prototype
void Timer_0_Init(void);
void Timer_1_Init(void);
void Timer_2_Init(void);

/* ----- */
void main(void)
{
    Timer_0_Init(); // Set up Timer 0
    Timer_1_Init(); // Set up Timer 1
    Timer_2_Init(); // Set up Timer 2

    EA = 1; // Globally enable interrupts

    while(1);
}

/* ----- */
void Timer_0_Init(void)
```

```
{  
    // Details omitted  
}  
  
/* ----- */  
  
void Timer_1_Init(void)  
{  
    // Details omitted  
}  
  
/* ----- */  
  
void Timer_2_Init(void)  
{  
    // Details omitted  
}  
  
/* ----- */  
  
void X(void) interrupt INTERRUPT_Timer_0_Overflow  
{  
    // This ISR is called every 1 ms  
  
    // Place required code here...  
}  
  
/* ----- */  
  
void Y(void) interrupt INTERRUPT_Timer_1_Overflow  
{  
    // This ISR is called every 2 ms  
  
    // Place required code here...  
}  
  
/* ----- */  
  
void Z(void) interrupt INTERRUPT_Timer_2_Overflow  
{  
    // This ISR is called every 5 ms  
  
    // Place required code here...  
}  
  
/*-----*  
----- END OF FILE -----*  
-----*/
```

Listing 13.5 The framework of an application using three independent timers to perform three tasks

Provided we have sufficient timers available, this approach will generally work. However, it would breach some basic software design guidelines.

For example, in Listing 13.5, we have three different timers to manage and – if we had 100 tasks – we would require 100 timers. This would make system maintenance very difficult; for example, 100 changes would be required if we changed the oscillator frequency. It would also be difficult to extend; for example, how can we add another task if there are no further hardware timers available?

In addition to contravening one of the most basic software design guidelines, there is a more specific problem with Listing 13.5. This arises in situations where more than one interrupt occurs simultaneously. As we saw in Chapter 1, having more than one active interrupt in a system can result in unpredictable – and hence, unreliable – patterns of behaviour.

Looking back at Listing 13.5 we can see that there will inevitably be occasions when more than one interrupt is generated at the same time. Dealing with this situation is not impossible, but it would add greatly to the complexity of the application.

Overall, as we will see in the next section, use of a scheduler provides a much cleaner solution.

13.7 What is a scheduler?

There are two ways of viewing a scheduler:

- At one level, a scheduler can be viewed as a simple operating system that allows tasks to be called periodically or (less commonly) on a one-shot basis.
- At a lower level, a scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks. As a result, only one timer needs to be initialized, and any changes to the timing generally requires only one function to be altered. Furthermore, we can generally use the same scheduler whether we need to execute one, ten or 100 different tasks. Note that this ‘shared ISR’ is very similar to the shared printing facilities (for example) provided by a desktop OS.

For example, Listing 13.6 shows how we might schedule the three tasks shown in Listing 13.5, this time using a scheduler.

```
/* ----- */
void main(void)
{
    // Set up the scheduler
    SCH_Init();

    // Add the tasks (1ms tick interval)
    // Function_A will run every 2 ms
    SCH_Add_Task(Function_A, 0, 2);

    // Function_B will run every 10 ms
    SCH_Add_Task(Function_B, 1, 10);
}
```

```

// Function_C will run every 15 ms
SCH_Add_Task(Function_C, 3, 15);

SCH_Start();
while(1)
{
    SCH_Dispatch_Tasks();
}
}

/* -----
--- END OF FILE ---
----- */

```

Listing 13.6 Running three periodic tasks (at different time intervals) using a scheduler

13.8 Co-operative and pre-emptive scheduling

We have discussed in very general terms the use of a scheduler to execute functions at particular times. Before we begin to consider the creation and use of a scheduler in detail in the next chapter, we need to appreciate that there are two broad classes of scheduler:

- The co-operative scheduler
- The pre-emptive scheduler

The features of the two types of scheduler are compared in Figures 13.5 and 13.6.

The co-operative scheduler

- A co-operative scheduler provides a **single-tasking system** architecture

Operation:

- Tasks are scheduled to run at specific times (either on a periodic or one-shot basis)
- When a task is scheduled to run it is added to the waiting list
- When the CPU is free, the next waiting task (if any) is executed
- The task runs to completion, then returns control to the scheduler

Implementation:

- The scheduler is simple and can be implemented in a small amount of code
- The scheduler must allocate memory for only a single task at a time
- The scheduler will generally be written entirely in a high-level language (such as 'C')
- The scheduler is not a separate application; it becomes part of the developer's code

Performance:

- Obtaining rapid responses to external events requires care at the design stage

Reliability and safety:

- Co-operate scheduling is simple, predictable, reliable and safe

FIGURE 13.5 Features of co-operative schedulers

The pre-emptive scheduler

- A pre-emptive scheduler provides a **multitasking system** architecture

Operation:

- Tasks are scheduled to run at specific times (either on a periodic or one-shot basis)
- When a task is scheduled to run it is added to the waiting list
- Waiting tasks (if any) are run for a fixed period then, if not completed, are paused and placed back in the waiting list. The next waiting task is then run for a fixed period, and so on

Implementation:

- The scheduler is comparatively complicated, not least because features such as semaphores must be implemented to avoid conflicts when 'concurrent' tasks attempt to access shared resources
- The scheduler must allocate memory to hold all the intermediate states of pre-empted tasks
- The scheduler will generally be written (at least in part) in Assembly language
- The scheduler is generally created as a separate application

Performance:

- Rapid responses to external events can be obtained

Reliability and safety:

- Generally considered to be less predictable, and less reliable, than co-operative approaches

FIGURE 13.6 Features of pre-emptive schedulers

In this book, we use mainly co-operative schedulers and will make limited use of hybrid schedulers (Figure 13.7). Together, these two forms of scheduler will provide the facilities we require (the ability to share a timer between multiple tasks, the ability to run both 'periodic' and 'one-shot' tasks): they do this while avoiding the complexities inherent in (fully) pre-emptive environments.

The key reason why the co-operative schedulers are both reliable and predictable is that only one task is active at any point in time: this task runs to completion, and then returns control to the scheduler. Contrast this with the situation in a fully pre-emptive system with more than one active task. Suppose one task in such a system which is reading from a port and the scheduler performs a 'context switch', causing a different task to access the same port: under these circumstances, unless we take action to prevent it, data may be lost or corrupted.

This problem arises frequently in multitasking environments where we have what are known as 'critical sections' of code. Such critical section are code areas that – once started – must be allowed to run to completion without interruption. Examples of critical sections include:

- Code which modifies or reads variables, particularly global variables used for inter-task communication. In general, this is the most common form of critical section, since inter-task communication is often a key requirement.

The hybrid scheduler

- A **hybrid scheduler** provides limited **multitasking** capabilities

Operation:

- Supports any number of co-operatively-scheduled tasks
- Supports a single pre-emptive task (which can interrupt the co-operative tasks)

Implementation:

- The scheduler is simple and can be implemented in a small amount of code
- The scheduler must allocate memory for two tasks at a time
- The scheduler will generally be written entirely in a high-level language (such as 'C')
- The scheduler is not a separate application; it becomes part of the developer's code

Performance:

- Rapid responses to external events can be obtained

Reliability and safety:

- With **careful design**, can be as reliable as a (pure) co-operative scheduler

FIGURE 13.7 Features of hybrid schedulers

- Code which interfaces to hardware, such as ports, analog-to-digital converters (ADCs) and so on. What happens, for example, if the same ADC is used simultaneously by more than one task?
- Code which calls common functions. What happens, for example, if the same function is called simultaneously by more than one task?

In a co-operative system, these problems do not arise, since only one task is ever active at the same time. To deal with such critical sections of code in a pre-emptive system, we have two main possibilities:

- 'Pause' the scheduling by disabling the scheduler interrupt before beginning the critical section; re-enable the scheduler interrupt when we leave the critical section.
- Or use a 'lock' (or some other form of 'semaphore mechanism') to achieve a similar result.

The first solution is that, when we start accessing the shared resource (say Port X), we disable the scheduler. This solves the immediate problem since (say) Task A will be allowed to run without interruption until it has finished with Port X. However, this 'solution' is less than perfect. For one thing, by disabling the scheduler, we will no longer be keeping track of the elapsed time and all timing functions will begin to drift – in this case by a period up to the duration of Task A every time we access Port X. This simply is not acceptable.

The use of locks is a better solution and appears, at first inspection, easy to implement. Before entering the critical section of code, we 'lock' the associated resource;

when we have finished with the resource we ‘unlock’ it. While locked, no other process may enter the critical section.¹

This is one way we might try to achieve this:

- 1 Task A checks the ‘lock’ for Port X it wishes to access.
- 2 If the section is locked, Task A waits.
- 3 When the port is unlocked, Task A sets the lock and then uses the port.
- 4 When Task A has finished with the port, it leaves the critical section and unlocks the port.

Implementing this algorithm in code also seems straightforward, as illustrated in Listing 13.7.

```
#define UNLOCKED    0
#define LOCKED      1

bit Lock:    //Global lock flag

// ...

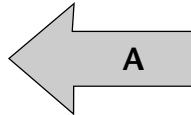
// Ready to enter critical section
// - Wait for lock to become clear
// (FOR SIMPLICITY, NO TIMEOUT CAPABILITY IS SHOWN)
while(Lock == LOCKED);

// Lock is clear
// Enter critical section
// Set the Lock
Lock = LOCKED;

// CRITICAL CODE HERE //

Ready to leave critical section
// Release the lock
Lock = UNLOCKED;

// ...
```



Listing 13.7 Attempting to implement a simple locking mechanism in a pre-emptive scheduler

However, this code cannot be guaranteed to work correctly under all circumstances.

1. Of course, this is only a partial solution to the problems caused by multitasking. If the purpose of Task A is to read from an ADC, and Task B has locked the ADC when the Task A is involved, then Task A cannot carry out its required activity. Use of locks, or any other mechanisms, will not solve this problem; however, they may prevent the system from crashing. Of course, by using a co-operative scheduler, these problems do not arise.

Consider the part of the code labelled 'A' in Listing 13.7. If our system is fully pre-emptive, then our task can reach this point at the same time as the scheduler performs a context switch and allows (say) Task B access to the CPU. If Task Y also wants to access the Port X

We can then have a situation as follows:

- Task A has checked the lock for Port X and found that the port is not locked; Task A has, however, not yet changed the lock flag.
- Task B is then 'switched in'. Task B checks the lock flag and it is still clear. Task B sets the lock flag and begins to use Port X.
- Task A is 'switched in' again. As far as Task A is concerned, the port is not locked; this task therefore sets the flag and starts to use the port, unaware that Task B is already doing so.
- ...

As we can see, this simple lock code violates the principle of mutual exclusion: that is, it allows more than one task to access a critical code section. The problem arises because it is possible for the context switch to occur after a task has checked the lock flag but before the task changes the lock flag. **In other words, the lock 'check and set code' (designed to control access to a critical section of code), is itself a critical section.**

This problem can be solved. For example, because it takes little time to 'check and set' the lock code, we can disable interrupts for this period. However, this is not in itself a complete solution: because there is a chance that an interrupt may have occurred even in the short period of 'check and set', we may then need to check the relevant interrupt flag(s) and, if necessary, call the relevant ISR(s). This can be done, but it adds to the complexity of the operating environment.

13.9 A closer look at pre-emptive schedulers

The discussion in this section is more technical than the previous sections in this chapter and may be omitted on a first reading of the book.

Various research studies have demonstrated that, compared to pre-emptive schedulers, co-operative schedulers have a number of desirable features. For example, Nissanke (1997, p. 237) notes:

[Pre-emptive] schedules carry greater runtime overheads because of the need for context switching – storage and retrieval of partially computed results. [Co-operative] algorithms do not incur such overheads. Other advantages of [co-operative] algorithms include their better understandability, greater predictability, ease of testing and their inherent capability for guaranteeing exclusive access to any shared resource or data.

Similarly, Allworth (1981, pp. 53–4) notes:

Significant advantages are obtained when using this [co-operative] technique. Since the processes are not interruptable, poor synchronisation does not give rise to the problem of shared data. Shared subroutines can be implemented without producing re-entrant code or implementing lock and unlock mechanisms.

Also, in a recent presentation, Bates (2000) identified the following four advantages of co-operative scheduling, compared to pre-emptive alternatives:

- 1 The scheduler is simpler
- 2 The overheads are reduced
- 3 Testing is easier
- 4 Certification authorities tend to support this form of scheduling

Despite these observations, all the authors cited and the vast majority of other workers in this area focus on the use of pre-emptive schedulers. At least part of the reason why pre-emptive approaches are more widely discussed is because of confusion over the options available. For example, Bennett (1994, p. 205) states:

If we consider the scheduling of time allocation on a single CPU there are two basic alternatives: [1] cyclic, [2] pre-emptive.

In fact, contrary to Bennett's assertion, what he refers to as cyclic scheduling is essentially a form of **SUPER LOOP** [page 162]. As we saw in Chapter 9, this type of architecture is suitable only for use in a restricted range of very simple applications, in particular those where accurate timing is not a key requirement and limited memory and CPU resources are available: **SUPER LOOP** is not representative of the broad range of co-operative scheduling architectures that are available.

Bennett is, however, not alone: other researchers make similar assumptions (see Barnett, 1995). For example, Locke (1992, p. 37) – in a widely cited publication – suggests that:

Traditionally, there have been two basic approaches to the overall design of application systems exhibiting hard real-time deadlines: the cyclic executive ... and the fixed priority [pre-emptive] architecture.

Similarly, Cooling (1991, pp. 292–3) compares co-operative and pre-emptive scheduling approaches. Again, however, his discussion of co-operative schedulers is restricted to a consideration of the special case of cyclic scheduling: as a result, his conclusion that a pre-emptive approach is more effective is unsurprising.

Where the different characteristics of pre-emptive and co-operative scheduling are compared equitably, the main concern expressed is often that long tasks will have an impact on the responsiveness of a co-operative scheduler. This concern is succinctly summarized by Allworth (1981):

[The] main drawback with this [co-operative] approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired.

There are four main technical reasons why such concerns are, generally, misplaced:

- In many embedded applications, the task duration *is* extremely brief. For example, consider one of the more complex algorithms considered in this book: proportional integral differential (PID) control. Even the most basic 8051 microcontroller can carry out a PID calculation in around 0.4 ms (see p. 872): even in flight control – where PID algorithms remain in widespread use – sample rates of around 10 ms are common, and a 0.4 ms calculation does not impose a significant processor load.
- Where the system does have long tasks, this is often because the developer is unaware of some simple techniques that can be used to break down these tasks in an appropriate way and – in effect – convert ‘long tasks called infrequently’ into ‘short tasks called frequently’. Such techniques are used throughout this book; they are introduced and explained in Chapter 16.
- In many cases, the increased power of microcontrollers has more than kept up with performance demands in many embedded systems. For example, the PID performance figures just given assumed an original 8051 microcontroller, with a 1 MIPS performance level. As we saw in Chapter 3, numerous low-cost members of this family now have performance levels between 5 and 50 MIPS. Often a simple, cost-effective, way of addressing performance concerns is not to use a more complex software architecture, but, instead, to update the hardware.
- If upgrades to the task design or microcontroller do not provide sufficient performance improvements, then more than one microcontroller can be used. This is now very common. For example, a typical automotive environment containing more than 40 embedded processors (Leen *et al.*, 1999). With the increased availability of such processing elements, long tasks may be readily ‘migrated’ to another processor, leaving the main CPU free to respond rapidly, if necessary, to other events. (See Part F of this book for numerous examples of this process.)

Finally, it should be noted that the reasons why pre-emptive schedulers have been more widely discussed and used may not be for technical reasons at all: in fact, the use of pre-emptive environments can be seen to have clear commercial advantages for some companies. For example, a co-operative scheduler may be easily constructed, entirely in a high-level programming language, in around 300 lines of ‘C’ code, as we demonstrate in Chapter 9. The code is highly portable, easy to understand and to use and is, in effect, freely available. By contrast, the increased complexity of a pre-emptive operating environment results in a much larger code framework (some ten times the size, even in a simple implementation: Labrosse, 1998). The size and complexity of this code makes it unsuitable for ‘in-house’ construction in most situations and therefore provides the basis for a commercial ‘RTOS’ products to be sold, generally at high prices and often with expensive run-time royalties to be paid. The continued promotion and sale of such environments has, in turn, prompted further academic interest in this area. For example, according to Liu and Ha, (1995):

[An] objective of reengineering is the adoption of commercial off-the-shelf and standard operating systems. Because they do not support cyclic scheduling, the adoption of these operating systems makes it necessary for us to abandon this traditional approach to scheduling.

13.10 Conclusions

In this chapter, we have explained what a scheduler is and outlined the differences between co-operative and pre-emptive scheduling. We have argued that a co-operative scheduler provides a simple and reliable operating environment that matches precisely the needs of most embedded applications.

Over recent years we have used versions of the co-operative schedulers presented in this book in numerous 'real' applications. We have also helped many student groups use this architecture in their first embedded systems. We have no doubt that the correct use of these schedulers not only results in simple, transparent and reliable designs, but also make it easier for 'desktop' developers to adapt rapidly to the challenges of embedded system development.

13.11 Further reading

- Allworth, S.T. (1981) *An Introduction to Real-Time Software Design*, Macmillan, London.
- Bates, I. (2000) "Introduction to scheduling and timing analysis", in *The Use of Ada in Real-Time System* (6 April, 2000). IEE Conference Publication 00/034.
- Burns, A. and Wellings, A. (1997) *Real-Time Systems and Programming Languages*, Addison-Wesley, London.
- Cooling, J.E. (1991) *Software Design for Real-Time Systems*, Chapman and Hall, London.
- Kopetz, H. (1997) *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic, New York.
- Leen, G., Heffernan, D. and Dunne, A. (1999) 'Digital networks in the automotive vehicle', *Computing and Control*, 10 (6): 257–266.
- Leveson, N.G. (1995) *Safeware: System Safety and Computers*, Addison-Wesley, Reading, MA.
- 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', *The Journal of Real-Time Systems*, 4: 37–53.
- Nissanke, N. (1997) *Realtime Systems*, Prentice Hall, London.
- Shaw, A.C. (2001) *Real-Time Systems and Software*, Wiley, New York.
- Storey, N. (1996) *Safety-Critical Computer Systems*, Addison-Wesley, London.
- 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.

chapter 14

Co-operative schedulers

Introduction

In this chapter, we discuss techniques for creating co-operative schedulers suitable for use in single-processor environments. These provide a very flexible and predictable software platform for a wide range of embedded applications, from the simplest consumer gadget up to and including aircraft control systems.

The following pattern is presented in this chapter:

- **CO-OPERATIVE SCHEDULER** [page 255]

A co-operative scheduler provides a simple, highly predictable environment. The scheduler is written entirely in 'C' and becomes part of the application: this tends to make the operation of the whole system more transparent and eases development, maintenance and porting to different environments. Memory overheads are seven bytes per task and CPU requirements (which vary with tick interval) are low.

CO-OPERATIVE SCHEDULER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application is to have a time-triggered architecture, constructed using a scheduler.

Problem

How do you create and use a co-operative scheduler?

Background

We present some links to relevant background material in this section.

What is a co-operative scheduler?

See Chapter 13 for an introduction to co-operative schedulers.

Function pointers

One area of the language with which many ‘C’ programmers are unfamiliar is the function pointer. While comparatively rarely used in desktop programs, this language feature is crucial in the creation of schedulers: we therefore provide a brief introductory example here.

The key point to note is that – just as we can, for example, determine the starting address of an array of data in memory – we can also find the address in memory at which the executable code for a particular function begins. This address can be used as a ‘pointer’ to the function; most importantly, it can be used to call the function.

Used with care, function pointers can make it easier to design and implement complex programs. For example, suppose we are developing a large, safety-critical, application, controlling an industrial plant. If we detect a critical situation, we may wish to shut down the system as rapidly as possible. However, the appropriate way to shut down the system will vary, depending on the system state. What we can do is create a number of different recovery functions and a function pointer. Every time the system state changes, we can alter the function pointer so that it is always pointing to the most appropriate recovery function. In this way, we know that – if there is ever an emergency situation – we can rapidly call the most appropriate function, by means of the function pointer.

The example in Listings 14.1 and 14.2 illustrates some of the basic features of function pointers.

```
/*-----*
Main.C (v1.00)

-----
Demonstration of function pointers.

-*-----*/
#include "Main.h"
#include "Printf51.h"
#include <stdio.h>

// ----- Private function prototypes -----
void Square_Number(int, int*);

/* ..... */ /* Declares pFn to be a pointer to fn with
   int and int pointer parameters
   (returning void) */

int main(void)
{
    int a = 2, b = 3;
    void (* pFn)(int, int*); /* Declares pFn to be a pointer to fn with
                               int and int pointer parameters
                               (returning void) */

    int Result_a, Result_b;

    // Prepare to use printf() [in Keil hardware simulator]
    Printf51_Init();

    pFn = Square_Number; // pFn holds address of Square_Number

    printf("Function code starts at address: %u\n", (tWord) pFn);
    printf("Data item a starts at address: %u\n\n", (tWord) &a);

    // Call 'Square_Number' in the conventional way
    Square_Number(a,&Result_a);

    // Call 'Square_Number' using function pointer
    (*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)
{

```

```

// Demo - calculate square of a
*b = a * a;
}

/*
-----*
----- END OF FILE -----
*----- */

```

Listing 14.1 Part of an example introducing the use of function pointers

```

/*
-----*
----- Printf51.C (v1.00)
-----*

A simple serial initialization routine to allow Keil hardware
simulator to be used to run 'desktop' C examples.

[Full details of a complete serial interface library are given
in Chapter 18. This code is for demo purposes only!!! ]

*/
-----*
-----#include "Main.h"
#include "Printf51.h"
-----*
----- Printf51_Init()
A simple serial initialization routine to allow Keil hardware
simulator to be used to run 'desktop' C examples.

*/
-----*
-----void Printf51_Init(void)
{
    const tWord BAUD_RATE = 9600;
    PCON &= 0x7F; // Set SMOD bit to 0 (don't double baud rates)
    // Receiver enabled.
    // 8-bit data, 1 start bit, 1 stop bit,
    // Variable baud rate (asynchronous)
    // Receive flag will only be set if a valid stop bit is received
    // Set TI (transmit buffer is empty)
    SCON = 0x72;
    TMOD |= 0x20; // T1 in mode 2, 8-bit auto reload
    // See Main.H for details of OSC_FREQ and OSC_PER_INST
    TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
        / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));

```

```
TL1 = TH1;  
TR1 = 1; // Run the timer  
  
TI = 1; // Send dummy byte  
  
// Interrupt *NOT* enabled  
ES = 0;  
}  
  
/*-----*  
----- END OF FILE -----  
-----*/
```

Listing 14.2 Part of an example introducing the use of function pointers

[Note: that this listing prepares the UART on the 80x52 microcontroller, so that we can use the Keil printf() library function. This example is for illustrative purposes only; we present a complete serial library example in Chapter 18.]

The program in Listings 14.1 and 14.2 generate the output shown in Figure 14.1 in the Keil hardware simulator.

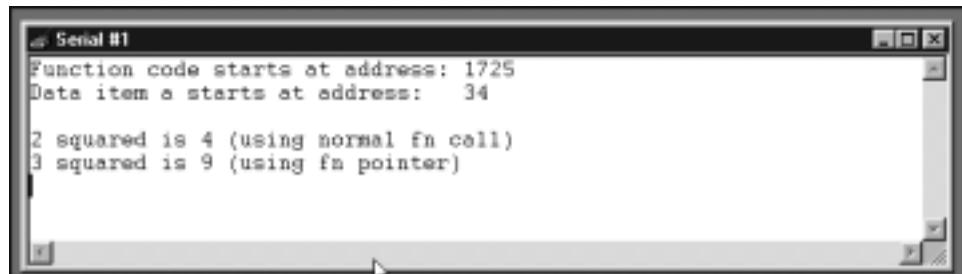


FIGURE 14.1 Using function pointers

Solution

A scheduler has the following key components:

- The scheduler data structure.
- An initialization function.
- A single interrupt service routine (ISR), used to update the scheduler at regular time intervals.
- A function for adding tasks to the scheduler.
- A dispatcher function that causes tasks to be executed when they are due to run.
- A function for removing tasks from the scheduler (not required in all applications).

We consider each of the required components in this section.

Overview

Before discussing the scheduler components, we consider how the scheduler will typically appear to the user. To do this we will use a simple example: a scheduler used to flash a single LED on and off repeatedly: on for one second off for one second etc.

Listing 14.3 shows how we might achieve this

```
/*-----*/
void main(void)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Prepare for the 'Flash_LED' task
    LED_Flash_Init();

    // Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
    // - timings are in ticks (1 ms tick interval)
    // (Max interval / delay is 65535 ticks)
    SCH_Add_Task(LED_Flash_Update, 0, 1000);

    // Start the scheduler
    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

/*-----*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // Update the task list
    ...
}
```

Listing 14.3 The key components from a simple scheduled application

Listing 14.3 operates as follows:

- 1 We assume that the LED will be switched on and off by means of a ‘task’ `LED_Flash_Update()`. Thus, if the LED is initially off and we call `LED_Flash_Update()` twice, we assume that the LED will be switched on and then switched off again.
To obtain the required flash rate, we therefore require that the scheduler calls `LED_Flash_Update()` every second *ad infinitum*.
- 2 We prepare the scheduler using the function `SCH_Init_T2()`.

- 3 After preparing the scheduler, we add the function `LED_Flash_Update()` to the scheduler task list using the `SCH_Add_Task()` function. At the same time we specify that the LED will be turned on and off at the required rate as follows:

```
// Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
// - timings are in ticks (1 ms tick interval)
// (Max interval / delay is 65535 ticks)
SCH_Add_Task(LED_Flash_Update, 0, 1000);
```

(We will shortly consider all the parameters of `SCH_Add_Task()`, and examine its internal structure).

- 4 The timing of the `LED_Flash_Update()` function will be controlled by the function `SCH_Update()`, an interrupt service routine triggered by the overflow of Timer 2:

```
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // Update the task list
    ...
}
```

- 5 The ‘Update’ ISR does not execute the task: it calculates when a task is due to run and sets a flag. The job of executing `LED_Flash_Update()` falls to the dispatcher function (`SCH_Dispatch_Tasks()`), which runs in the main (‘super’) loop:

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

Before considering these components in detail, we should acknowledge that this is, undoubtedly, a complicated way of flashing an LED: if our intention were to develop an LED flasher application that required minimal memory and minimal code size, this would **not** be a good solution. However, the key point is that we will be able to use the **same scheduler architecture** in all our subsequent examples, including a number of substantial and complex applications and the effort required to understand the operation of this environment will be rapidly repaid.

It should also be emphasized that the scheduler is a ‘low-cost’ option: it consumes a small percentage of the CPU resources (we will consider precise percentages shortly). In addition, the scheduler itself requires no more than 7 bytes of memory for each task. Since a typical application will require no more than four to six tasks, the task – memory budget (around 40 bytes) is not excessive, even on an 8-bit microcontroller.

The scheduler data structure and task array

At the heart of the scheduler is the scheduler data structure: this is a user-defined data type which collects together the information required about each task.

The data structure is reproduced (from file Sch51.H) here:

```
// Store in DATA area, if possible, for rapid access
// Total memory per task is 7 bytes
typedef data struct
{
    // Pointer to the task (must be a 'void (void)' function)
    void (code * pTask)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tWord Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tWord Period;

    // Incremented (by scheduler) when task is due to execute
    tByte RunMe;
} sTask;
```

File Sch51.H also includes the constant SCH_MAX_TASKS:

```
// The maximum number of tasks required at any one time
// during the execution of the program
//
// MUST BE ADJUSTED FOR EACH NEW PROJECT
#define SCH_MAX_TASKS (1)
```

Both the sTask data type and the SCH_MAX_TASKS constant are used to create – in the file Sch51.C – the array of tasks that is referred to throughout the scheduler:

```
// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];
```

The size of the task array

You **must** ensure that the task array is sufficiently large to store the tasks required in your application, by adjusting the value of SCH_MAX_TASKS.

For example, if you schedule three tasks as follows:

```
SCH_Add_Task(Function_A, 0, 2);
SCH_Add_Task(Function_B, 1, 10);
SCH_Add_Task(Function_C, 3, 15);
```

then SCH_MAX_TASKS must have a value of three (or more) for correct operation of the scheduler.

Note also that, if this condition is not satisfied, the scheduler will generate an error code (see page 274).

The initialization function

Like most of the tasks we wish to schedule, the scheduler itself requires an initialization function. While this performs various important operations – such as preparing the scheduler array (discussed earlier) and the error code variable (discussed later) – the main purpose of this function is to set up a timer that will be used to generate the regular ‘ticks’ that will drive the scheduler.

As we saw in Chapter 11, most 8051 devices have three timers (Timer 0, Timer 1 and Timer 2) and any of these may be used to drive the scheduler. However, only Timer 2 can be used as an auto-reload timer with 16-bit resolution: as a result, use of this timer is appropriate, if it is available.

Note that there are numerous different 8051 schedulers included on the CD accompanying this book: the different schedulers illustrate the use of all three of the available timers.

One possible initialization function (using Timer 2) is illustrated in Listing 14.4.

```
/*-----*
SCH_Init_T2()
Scheduler initialization function. Prepares scheduler
data structures and sets up timer interrupts at required rate.
Must call this function before using the scheduler.

*-----*/
void SCH_Init_T2(void)
{
    tByte i;

    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // Now set up Timer 2
    // 16-bit timer function with automatic reload

    // Crystal is assumed to be 12 MHz
    // The Timer 2 resolution is 0.000001 seconds (1 µs)
    // The required Timer 2 overflow is 0.001 seconds (1 ms)
```

```

// - this takes 1000 timer ticks
// Reload value is 65536 - 1000 = 64536 (dec) = 0xFC18

T2CON = 0x04; // load Timer 2 control register
T2MOD = 0x00; // load Timer 2 mode register

TH2 = 0xFC; // load timer 2 high byte
RCAP2H = 0xFC; // load timer 2 reload capture reg, high byte
TL2 = 0x18; // load timer 2 low byte
RCAP2L = 0x18; // load timer 2 reload capture reg, low byte

ET2 = 1; // Timer 2 interrupt is enabled

TR2 = 1; // Start Timer 2
}

```

Listing 14.4 One possible scheduler initialization function

When using any of the schedulers in this book, you will generally need to adapt the initialization code to match your needs. In particular, you will need to ensure that:

- 1 The oscillator / resonator frequency assumed in the initialization function matches your hardware. In Listing 14.4 the frequency assumed is 12 MHz.
- 2 The tick interval of the scheduler matches your requirements. In Listing 14.4 the tick interval is 1 ms.

Guidance on the choice of the tick interval is provided below in ‘Reliability and safety implications’.

The ‘one interrupt per microcontroller’ rule

The scheduler initialization function enables the generation of interrupts associated with the overflow of one of the microcontroller timers.

For reasons discussed in Chapter 1, it is assumed throughout this book that only the ‘tick’ interrupt source is active: specifically, it is assumed that no other interrupts are enabled.

If you attempt to use the scheduler code with additional interrupts enabled, the system cannot be guaranteed to operate at all: at best, you will generally obtain very unpredictable, and unreliable, system behaviour.

The ‘Update’ function

The ‘Update’ function is the scheduler ISR. It is invoked by the overflow of the timer (set up using the ‘Init’ function, as discussed in the previous section).

Like most of the scheduler, the update function is not complex (see Listing 14.5).

When it determines that a task is due to run, the update function increments the RunMe field for this task: the task will then be executed by the dispatcher, as we discuss later.

```
/* -----
SCH_Update()

This is the scheduler ISR. It is called at a rate
determined by the timer settings in the 'init' function.

This version is triggered by Timer 2 interrupts:
timer is automatically reloaded.

----- */

void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;

    TF2 = 0; // Have to manually clear this.

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Inc. the 'RunMe' flag

                if (SCH_tasks_G[Index].Period)
                {
                    // Schedule periodic tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}
```

Listing 14.5 A scheduler ‘update’ function

The 'Add Task' function

As the name suggests, the 'Add Task' function is used to add tasks to the task array, to ensure that they are called at the required time(s).

The parameters for the 'Add Task' function are described in Figure 14.2.

Here are some examples.

This set of parameters causes the function Do_X() to be executed once after 1,000 scheduler ticks:

```
SCH_Add_Task(Do_X,1000,0);
```

This does the same, but saves the task ID (the position in the task array) so that the task may be subsequently deleted, if necessary (see SCH_Delete_Task() for further information about the removal of tasks from the task array):

```
Task_ID = SCH_Add_Task(Do_X,1000,0);
```

This causes the function Do_X() to be executed regularly every 1,000 scheduler ticks; the task will first be executed as soon as the scheduling is started:

```
SCH_Add_Task(Do_X,0,1000);
```

This causes the function Do_X() to be executed regularly every 1,000 scheduler ticks; task will be first executed at T = 300 ticks, then 1,300, 2,300 etc:

```
SCH_Add_Task(Do_X,300,1000);
```

The function itself is shown in Listing 14.6.

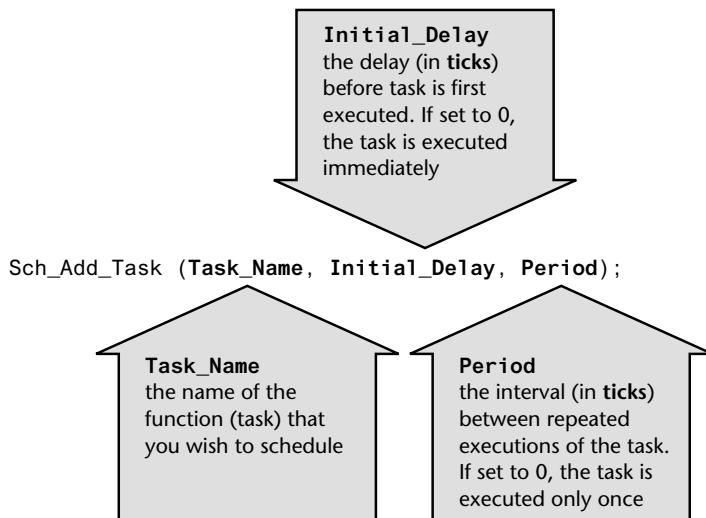


FIGURE 14.2 The parameters of the SCH_Add_Task() function

```

/* -----
SCH_Add_Task()
Causes a task (function) to be executed at regular intervals
or after a user-defined delay
----- */

tByte SCH_Add_Task(void * pFunction(),
                   const tWord DELAY,
                   const tWord PERIOD)
{
    tByte Index = 0;

    // First find a gap in the array (if there is one)
    while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
    {
        Index++;
    }

    // Have we reached the end of the list?
    if (Index == SCH_MAX_TASKS)
    {
        // Task list is full
        //
        // Set the global error variable
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;

        // Also return an error code
        return SCH_MAX_TASKS;
    }

    // If we're here, there is a space in the task array
    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; // return position of task (to allow later deletion)
}

```

Listing 14.6 An implementation of the scheduler ‘add task’ function

The ‘Dispatcher’

As we have seen, the ‘Update’ function does not execute any tasks: the tasks that are due to run are invoked through the ‘Dispatcher’ function.

Details of the dispatcher are given in Listing 14.7.

```

/* -----
SCH_Dispatch_Tasks()

This is the 'dispatcher' function. When a task (function)
is due to run, SCH_Dispatch_Tasks() will run it.
This function must be called (repeatedly) from the main loop.

----- */
void SCH_Dispatch_Tasks(void)
{
    tByte Index;

    // Dispatches (runs) the next task (if one is ready)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        if (SCH_tasks_G[Index].RunMe > 0)
        {
            (*SCH_tasks_G[Index].pTask)(); // Run the task

            SCH_tasks_G[Index].RunMe -= 1; // Reset / reduce RunMe flag

            // Periodic tasks will automatically run again
            // - if this is a 'one shot' task, remove it from the array
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }

    // Report system status
    SCH_Report_Status();

    // The scheduler enters idle mode at this point
    SCH_Go_To_Sleep();
}

```

Listing 14.7 An implementation of the scheduler 'dispatch task' function

The dispatcher is the only component in the Super Loop:

```

/* -----
void main(void)
{
    ...

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

```

Do we need a Dispatch function?

At first inspection, the use of both the 'Update' and 'Dispatch' functions may seem a rather complicated way of running the tasks. Specifically, it may appear that the Dispatch function is unnecessary and that the Update function could invoke the tasks directly. However, the split between the Update and Dispatch operations is necessary, to maximize the reliability of the scheduler in the presence of long tasks.

Suppose we have a scheduler with a tick interval of 1 ms and, for whatever reason, a scheduled task sometimes has a duration of 3 ms.

If the Update function runs the functions directly then – all the time the long task is being executed – the tick interrupts are effectively disabled. Specifically, two 'ticks' will be missed. This will mean that all system timing is seriously affected and may mean that two (or more) tasks are not scheduled to execute at all.

If the Update and Dispatch function are separated, system ticks can still be processed while the long task is executing. This means that we will suffer task 'jitter' (the 'missing' tasks will not be run at the correct time), but these tasks will, eventually, run.

Function pointers and Keil linker options

When we write:

```
SCH_Add_Task(Do_X,1000,0);
```

the first parameter of the 'Add Task' function is *a pointer* to the function Do_X(). This function pointer is then passed to the Dispatch function and it is through this function that the task is executed:

```
if (SCH_tasks_G[Index].RunMe > 0)
{
    (*SCH_tasks_G[Index].pTask)(); // Run the task
```

We discussed the use of function pointers in 'Background'. The use of the 'C' function pointers on small microcontrollers presents a challenge. This is particularly true when function pointers are used as function arguments.

On desktop systems, function arguments are generally passed on the stack using the push and pop assembly instructions. Since the 8051 has a size limited stack (only 128 bytes at best and as low as 64 bytes on some devices), function arguments must be passed using a different technique: in the case of Keil C51, these arguments are stored in fixed memory locations. When the linker is invoked, it builds a call tree of the program, decides which function arguments are mutually exclusive (that is, which functions cannot be called at the same time) and overlays these arguments.

The linker has difficulty determining the correct call tree when function pointers are used as function arguments, as is the case with the 'Add Task' function. To deal with this situation, you have two realistic options:

- 1 You can prevent the compiler from using the OVERLAY directive by disabling overlays as part of the linker options for your project.

Note that, compared to applications using overlays, you will generally require more RAM to run your program.

- 2 You can tell the linker how to create the correct call tree for your application by explicitly providing this information in the linker 'Additional Options' dialogue box.

This solution generally uses less memory, but the compiler often cannot tell if you provide incorrect information: if you get this option wrong, your program can generate unpredictable results.

The linker options required are not difficult to understand. Suppose we have run our simple flashing LED example presented earlier in this chapter and we are scheduling a single task, as follows:

```
void main(void)
{
    ...

    // Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
    // - timings are in ticks (1 ms tick interval)
    // (Max interval / delay is 65535 ticks)
    SCH_Add_Task(LED_Flash_Update, 0, 1000);

    ...
}
```

The linker assumes – because the pointer `LED_Flash_Update` appears in `main()` – that the function is called from `main()`. Instead, the function is called from `SCH_Dispatch_Tasks`.

We make this change explicit using the linker options thus:

```
OVERLAY
(main ~ (LED_Flash_Update),
SCH_Dispatch_Tasks ! (LED_Flash_Update))
```

If you look at the various project files on the CD, you will find that the required linker options have already been implemented.

For further information on function pointers, refer to the Keil compiler documentation and to Keil Application Note 129 (included on the CD).

Are function pointers safe?

Is it dangerous to use a scheduler based on function pointers? Will the use of function pointers make your program less reliable? The answer to both questions is 'no', with some caveats.

Before concluding that use of function pointers is not safe, we need to consider the alternatives. You can create a scheduler without using function pointers (and we have created several in this way). However, the resulting code tends to be larger, less flexible, less easy to adapt and less easy to read than the pointer-based version. In our experience, these factors together have a detrimental impact on reliability that far outweighs the advantages gained by avoiding function pointers.

If you opt not to use the Keil OVERLAY directive, you do not need to be concerned with the complexities of function pointers. If you do not feel comfortable with the use of such pointers or if your code may subsequently be maintained by less experienced developers, then – with current versions of the Keil compiler – this is the most appropriate option.

The 'Start' function

The 'Start' function is very simple (see Listing 14.8). After all the tasks have been added, this function is called to begin the scheduling process. The function achieves this by globally enabling interrupts.

```
/*-----*/
void SCH_Start(void)
{
    EA = 1;
}
```

Listing 14.8 An implementation of the scheduler 'start' function

The 'Delete Task' function

When tasks are added to the task array, `SCH_Add_Task()` returns the position in the task array at which the task has been added:

```
Task_ID = SCH_Add_Task(Do_X,1000,0);
```

Sometimes it can be necessary to delete tasks from the array. To do so, `SCH_Delete_Task()` can be used as follows:

```
SCH_Delete_Task(Task_ID);
```

Details of `SCH_Delete_Task()` are given in Listing 14.9.

```
/*-----*/
bit SCH_Delete_Task(const tByte TASK_INDEX)
{
    bit Return_code;

    if (SCH_tasks_G[TASK_INDEX].pTask == 0)
    {
        // No task at this location...
        //
        // Set the global error variable
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;
```

```

    // ...also return an error code
    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; // return status
}

```

Listing 14.9 An implementation of the scheduler ‘delete task’ function

Reducing power consumption

An important feature of scheduled applications is that they can lend themselves to low-power operation. This is possible because all current members of the 8051 family provide an ‘idle’ mode, where the CPU activity is halted, but the state of the processor is maintained. In this mode, the power required to run the processor is typically reduced by around 50%.

This idle mode is particularly effective in scheduled applications because it may be entered under software control (as shown in Listing 14.10), and the microcontroller returns to the normal operating mode when any interrupt is received. Because the scheduler generates regular timer interrupts as a matter of course, we can put the system ‘to sleep’ at the end of every dispatcher call: it will then wake up when the next timer tick occurs.

```

/*-----*/
void SCH_Go_To_Sleep()
{
    PCON |= 0x01;      // Enter idle mode (generic 8051 version)

    // Entering idle mode requires TWO consecutive instructions
    // on 80c515 / 80c505 - to avoid accidental triggering
    // PCON |= 0x01;      // Enter idle mode (#1)
    // PCON |= 0x20;      // Enter idle mode (#2)
}

```

Listing 14.10 An implementation of the scheduler ‘sleep’ function

Reporting errors

Hardware fails; software is never perfect; errors are a fact of life.

To report errors at any part of the scheduled application, we use an (8-bit) error code variable `Error_code_G`, which is defined in Sch51.C as follows:

```
// Used to display the error code
tByte Error_code_G = 0;
```

To record an error we include lines such as:

```
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;
```

These error codes are given in the file Main.H which is an example of the pattern **PROJECT HEADER** [page 169].

To report these error codes, the scheduler has a function `SCH_Report_Status()`, which is called from the Update function.

One possible implementation is shown in Listing 14.11.

```
/*-----*/
void SCH_Report_Status(void)
{
#ifndef SCH_REPORT_ERRORS
    // ONLY APPLIES IF WE ARE REPORTING ERRORS
    // Check for a new error code
    if (Error_code_G != Last_error_code_G)
    {
        // Negative logic on LEDs assumed
        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; // Reset error code
            }
        }
    }
#endif
}

```

Listing 14.11 An implementation of the scheduler ‘report status’ function

Note that error reporting may be disabled via the Port.H header file:

```
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS
```

Where error reporting is required, the port on which error codes will be displayed is also determined via Port.H:

```
#ifdef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif
```

Note that, in this implementation, error codes are reported for 60,000 ticks (1 minute at a 1 ms tick rate).

The simplest way of displaying these codes is to attach eight LEDs (with suitable buffers) to the error port, as discussed in **IC DRIVER** [page 134]: Figure 14.3 illustrates one possible approach.

What does that error code mean?

The forms of error reporting discussed here are low-level in nature and are primarily intended to assist the developer of the application or a qualified service engineer performing system maintenance. An additional user interface may also be required in your application to notify the user of errors, in a more user-friendly manner.

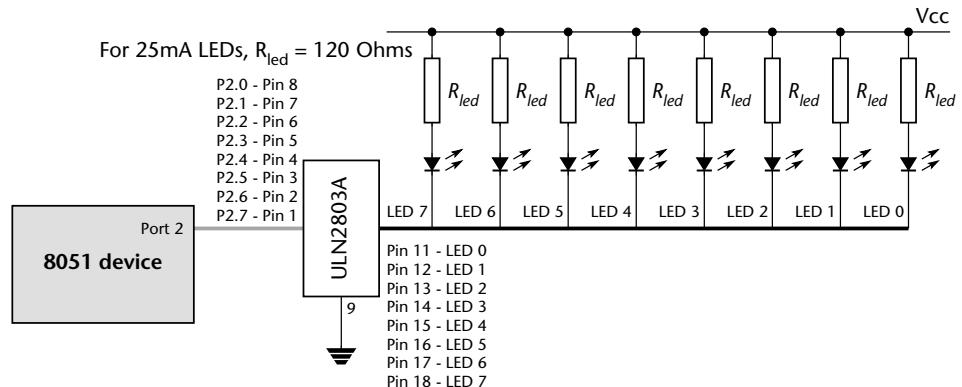


FIGURE 14.3 Hardware for reporting error codes

Adding a watchdog

The basic scheduler presented here does not provide support for a watchdog timer.

Such support can be useful and is easily added, as follows:

- Start the watchdog in the scheduler Start function.
- Refresh the watchdog in the scheduler Update function.

Hardware resource implications

We consider the hardware resource implications under three main headings: timers, memory and CPU load.

Timer

This pattern requires one hardware timer. If possible, this should be a 16-bit timer, with auto-reload capabilities, such as Timer 2 (see Chapter 13 for details).

Memory

This main scheduler memory requirement is seven bytes of memory per task. Most applications require around six tasks or fewer. Even in a standard 8051/8052 with 256 bytes of internal memory the total memory overhead is small.

CPU load

Probably the main reason why some developers still do not use a scheduler is concern about the CPU load this architecture will impose. Such concerns are generally misplaced since, in a typical application, an average of around 5% of the available CPU time is consumed by the scheduler.

To illustrate this consider, first, the worst-case scenario: a Standard 8051, operating at 12 machine cycles per oscillator cycle used to schedule a single, simple, task (flashing an LED on and off). We will further assume that the scheduler is required to provide a 1 ms tick interval and the oscillator frequency is 12 MHz.

We can simulate this system using the Keil hardware simulator (included on the CD). This reveals that the largest CPU load is imposed by the scheduler ‘update’ ISR. In total, all the scheduler processing consumes around 14% of the available CPU time (Figure 14.4).

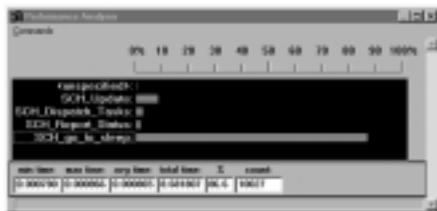


FIGURE 14.4 Using the Keil hardware simulator to evaluate the CPU load imposed by a scheduler with 1 ms ticks, running on a 12 MHz (12 oscillations per instruction) 8051 86% idle

[Note: **One task is being executed.** The test reveals that the CPU is 86% idle and that the maximum possible task duration is therefore approximately 0.86 ms.]

If we add another task, the scheduler is still 80% ‘idle’ (Figure 14.5).



FIGURE 14.5 Using the Keil hardware simulator to evaluate the CPU load imposed by a scheduler with 1 ms ticks, running on a 12 MHz (12 oscillations per instruction) 8051 80% idle

[Note: **Two tasks are being executed.** The test reveals that the CPU is 80% idle and that the maximum possible task duration is therefore approximately 0.80 ms.]

In most situations, around 12 tasks is the maximum number we would expect to be able to schedule in such an application. Figure 14.6 shows this situation.



FIGURE 14.6 Using the Keil hardware simulator to evaluate the CPU load imposed by a scheduler with 1 ms ticks, running on a 12 MHz (12 oscillations per instruction) 8051 11% idle.

[Note: **Twelve tasks are being executed.** The test reveals that the CPU is 11% idle and that the maximum possible task duration is therefore approximately 0.11 ms.]

Of course, by modern standards, this 12 MHz / 12 Osc cycles 8051 is a slow device. Figure 14.7 shows the same application (running a single task), adapted to operate on a 24 MHz version of the scheduler: this application is 93% idle.



FIGURE 14.7 Using the Keil hardware simulator to evaluate the CPU load imposed by a scheduler with 1 ms ticks, running on a 24 Mhz (12 oscillations per instruction) 8051

[Note: **One task is being executed.** The test reveals that the CPU is 93% idle and that the maximum possible task duration is therefore approximately 0.93 ms.]

Similarly, the same application adapted for the Dallas 320/520 (32 MHz) version is seen to be 98% idle when running in Figure 14.8 (left). The same system can very comfortably run 12 tasks (Figure 14.8 (right)) and remains 85% idle under these circumstances.



FIGURE 14.8 Using the Keil hardware simulator to evaluate the CPU load imposed by a scheduler with 1 ms ticks, running on a 32 MHz (4 oscillations per instruction) 8051

[Note: [Left] **One task is being executed.** The test reveals that the CPU is 97% idle and that the maximum possible task duration is therefore approximately 0.97 ms. [Right] **Twelve tasks are being executed.** The test reveals that the CPU is 85% idle and that the maximum possible task duration is therefore approximately 0.85 ms.]

Finally, Figure 14.9 shows a 12 Mhz / 12 clocks 8051 running a scheduler with a 10 ms tick. On the right, the system is running one task and is 98% idle: on the right, the system is running 12 tasks and is 91% idle.

Reliability and safety implications

In this section we consider some key reliability and safety implications.



FIGURE 14.9 Using the Keil hardware simulator to evaluate the CPU load imposed by a scheduler with 10 ms ticks, running on a 12 MHz (12 oscillations per instruction) 8051

[Note: [Left] **One task is being executed.** The test reveals that the CPU is 98% idle and that the maximum possible task duration is therefore approximately 9.8 ms. [Right] **Twelve tasks are being executed.** The test reveals that the CPU is 91% idle and that the maximum possible task duration is therefore approximately 9.1 ms.]

Make sure the task array is large enough

See 'Solution' for details.

Take care with function pointers

See 'Background' and 'Solution' for details.

Dealing with task overlap

Suppose we have two tasks in our application (Task A, Task B). We further assume that Task A is to run every second and Task B every three seconds. We assume also that each task has a duration of around 0.5 ms.

Suppose we schedule the tasks as follows (assuming a 1ms tick interval):

```
SCH_Add_Task(TaskA, 0, 1000);
SCH_Add_Task(TaskB, 0, 3000);
```

In this case, the two tasks will sometimes be due to execute at the same time. On these occasions, both tasks will run, but Task B will always execute after Task A (see Listing 14.5 and Listing 14.6 for details). This will mean that if Task A varies in duration, then Task B will suffer from 'jitter': it will not be called at the correct time when the tasks overlap.

Alternatively, suppose we schedule the tasks as follows:

```
SCH_Add_Task(TaskA, 0, 1000);
SCH_Add_Task(TaskB, 5, 3000);
```

Now, both tasks still run every 1,000 ms and 3,000 ms (respectively), as required. However, Task A is explicitly scheduled always to run 5 ms before Task B. As a result, Task B will always run on time.

In many cases, we can avoid all (or most) task overlaps simply by the judicious use of the initial task delays.

Determining the required tick interval

Throughout this book, our main focus is in applications which operate on a millisecond timescale. Thus, the various tasks you will be adding to the scheduler will typically have task intervals of (say) 12 ms, 3 ms and 1,000 ms.

In most instances, the simplest way of meeting the needs of the various task intervals is to allocate a scheduler tick interval of 1 ms. This is easily done: see **HARDWARE DELAY** [page 194] and Chapter 13 for details.

Remember, however, that the scheduler itself will impose a CPU load on the microcontroller and that this load will increase dramatically at low tick intervals (see 'Hardware resource implications'). To keep the scheduler load as low as possible (and to reduce the power consumption: see the following), it can help to use a long tick interval.

If you want to reduce overheads and power consumption to a minimum, the scheduler tick interval should be set to match the 'greatest common factor' of all the task (and offset intervals). This is easily calculated, if you remember some simple high school mathematics.

Suppose we have three tasks (X,Y,Z), and Task X is to be run every 10 ms, Task Y every 30 ms and Task Z every 25 ms. The scheduler tick interval needs to be set by determining the relevant factors, as follows:

- The factors² of the Task X interval (10 ms) are: 1 ms, 2 ms, 5 ms and 10 ms.
- Similarly, the factors of the Task Y interval (30 ms) are as follows: 1 ms, 2 ms, 3 ms, 5 ms, 6 ms, 10 ms, 15 ms and 30 ms.
- Finally, the factors of the Task Z interval (25 ms) are as follows: 1 ms, 5 ms and 25 ms.

In this case, therefore, the greatest common factor is 5 ms: this is the required tick interval.

Note that it may seem that if you have task intervals of (say) 5 ms, 25 ms and 1,000 ms, this process will be extremely tedious, because 1,000 will have many factors. However, in practice, we are only concerned with the factors up to and including the smallest of the task intervals. In this case, therefore, we would be only interested in the factors of 5, 25 and 1,000 between 1 and 5. The largest common factor being, in this case, 5 ms.

The situation becomes slightly more complicated if we consider the initial task delays.

If we go back to our earlier example, suppose we have decided to use a 5 ms scheduler. We are adding three tasks to the scheduler as follows:

```
SCH_Add_Task(X, 0, 2);
SCH_Add_Task(Y, 0, 6);
SCH_Add_Task(Z, 0, 5);
```

Clearly, these tasks are going to frequently overlap. For example, every time Task Y is scheduled to run, so is Task X; on some occasions, all three tasks are due to run simultaneously. To avoid this, we can add some initial task delays, as follows:

2. Remember: the factors are integers (between 1 and X) by which we can divide X and obtain a remainder of 0.

- Task X is to be run every 10 ms: we start this task immediately.
- Task Z is to be run every 25 ms: we start this task after 2 ms.
- Task Y is to be run every 30 ms; we start this task after 1 ms.

When determining the required scheduler interval, we must now take into account both the task intervals and the initial delays. This, in this case, we now need to find the greatest common factor of 10, 25, 30, 1 and 2: this suggests that a scheduler interval of 1 ms is now required.

Guidelines for predictable and reliable scheduling

- 1 For precise scheduling, the scheduler tick interval should be set to match the 'greatest common factor' of all the task intervals (see earlier).
- 2 All tasks should have a duration less than the schedule tick interval, to ensure that the dispatcher is always free to call any task that is due to execute. Software simulation can often be used to measure the task duration.
- 3 In order to meet Condition 2, all tasks **must** 'timeout' so that they cannot block the scheduler under any circumstances. Note that this condition can often be met by incorporating, where necessary, a **LOOP TIMEOUT** [page 298] or a **HARDWARE TIMEOUT** [page 305] in scheduled tasks.

Please remember that this condition also applies to any functions called from within a scheduled task, including any library code provided by your compiler manufacturer. In many cases, standard functions (like `printf()`) do not include timeout features. They must not be used in situations where predictability is required.

- 4 The total time required to execute all the scheduled tasks must be less than the available processor time. Of course, the total processor time must include both this 'task time' and the 'scheduler time' required to execute the scheduler update and dispatcher operations.
- 5 Tasks should be scheduled so that they are never required to execute simultaneously: that is, task overlaps should be minimized. Note that where **all** tasks are of a duration much less than the scheduler tick interval, and that some task jitter can be tolerated, this problem may not be significant.

Portability

A co-operative scheduler, like that described in this pattern, can be written entirely in 'C' and need use only core 8051 hardware features. A scheduler created for one 8051 family member may therefore be readily ported for use on any other 8051 device.

The techniques described here may also be used without difficulty on other microcontrollers.

Overall strengths and weaknesses

The overall strengths and weaknesses of a co-operative scheduler may be summarized as follows:

- ☺ The scheduler is simple and can be implemented in a small amount of code.
- ☺ The applications based on the scheduler are inherently predictable, safe and reliable.
- ☺ The scheduler is written entirely in 'C': it is not a separate application, but becomes part of the developer's code.
- ☺ The scheduler supports team working, since individual tasks can often be developed largely independently and then assembled into the final system.
- ☹ Obtain rapid responses to external events requires care at the design stage.
- ☹ The tasks cannot safely use interrupts: the only interrupt that should be active in the application is the timer-related interrupt that drives the scheduler itself.

Related patterns and alternative solutions

For alternative solutions see:

- HYBRID SCHEDULER [page 333]
- ONE-TASK SCHEDULER [page 911]
- ONE-YEAR SCHEDULER [page 919]
- STABLE SCHEDULER [page 932]
- Chapter 13, for details of pre-emptive schedulers

Example: The core scheduler library (generic)

The schedulers in this book are all constructed using a library of core files, presented here. Specified schedulers (designed for different hardware, clock frequencies and / or tick intervals) then add a small number of additional files on top of this core.

Most of the material in the core library has already been described earlier in this chapter. The complete library is presented in Listings 14.12 and 14.13 to illustrate how the various components fit together.

```
/* ----- * -
SCH51.h (v1.00)

-----
- see SCH51.C for details
----- */
```

```

#ifndef _SCH51_H
#define _SCH51_H

#include "Main.h"

// ----- Public data type declarations -----
// Store in DATA area, if possible, for rapid access
// Total memory per task is 7 bytes
typedef data struct
{
    // Pointer to the task (must be a 'void (void)' function)
    void (code * pTask)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tWord Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tWord Period;

    // Incremented (by scheduler) when task is due to execute
    tByte RunMe;
} sTask;

// ----- Public function prototypes -----
// Core scheduler functions
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);

// ----- Public constants -----
// The maximum number of tasks required at any one time
// during the execution of the program
//
// MUST BE ADJUSTED FOR EACH NEW PROJECT
#define SCH_MAX_TASKS (1)

#endif

/* -----
--- END OF FILE -----
*/

```

Listing 14.12 Part of the core scheduler library

```
/*-----*  
SCH51.C (v1.00)  
-----  
*** THESE ARE THE CORE SCHEDULER FUNCTIONS ***  
--- These functions may be used with all 8051 devices ---  
*** SCH_MAX_TASKS *must* be set by the user ***  
--- see "Sch51.h" ---  
*** Includes power-saving mode ***  
--- You *MUST* confirm that the power-down mode is adapted ---  
--- to match your chosen device (usually only necessary with  
--- Extended 8051s, such as c515c, c509, etc ---  
-*-----*/  
  
#include "Main.h"  
#include "Port.h"  
  
#include "Sch51.h"  
  
// ----- Public variable definitions -----  
  
// The array of tasks  
sTask SCH_tasks_G[SCH_MAX_TASKS];  
  
// Used to display the error code  
// See Main.H for details of error codes  
// See Port.H for details of the error port  
tByte Error_code_G = 0;  
  
// ----- Private function prototypes -----  
  
static void SCH_Go_To_Sleep(void);  
  
// ----- Private variables -----  
  
// Keeps track of time since last error was recorded (see below)  
static tWord Error_tick_count_G;  
  
// The code of the last error (reset after ~1 minute)  
static tByte Last_error_code_G;  
  
/*-----*  
SCH_Dispatch_Tasks()  
  
This is the 'dispatcher' function. When a task (function)  
is due to run, SCH_Dispatch_Tasks() will run it.  
-----*
```

This function must be called (repeatedly) from the main loop.

```
-----*/
void SCH_Dispatch_Tasks(void)
{
    tByte Index;

    // Dispatches (runs) the next task (if one is ready)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        if (SCH_tasks_G[Index].RunMe > 0)
        {
            (*SCH_tasks_G[Index].pTask)(); // Run the task

            SCH_tasks_G[Index].RunMe -= 1; // Reset / reduce RunMe flag

            // Periodic tasks will automatically run again
            // - if this is a 'one shot' task, remove it from the array
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }

    // Report system status
    SCH_Report_Status();

    // The scheduler enters idle mode at this point
    SCH_Go_To_Sleep();
}
```

```
-----*/
```

SCH_Add_Task()

Causes a task (function) to be executed at regular intervals or after a user-defined delay

Fn_P - The name of the function which is to be scheduled.
 NOTE: All scheduled functions must be 'void, void' - that is, they must take no parameters, and have a void return type.

DELAY - The interval (TICKS) before the task is first executed

PERIOD - If 'PERIOD' is 0, the function is only called once, at the time determined by 'DELAY'. If PERIOD is non-zero, then the function is called repeatedly at an interval determined by the value of PERIOD (see below for examples which should help clarify this).

RETURN VALUE:

Returns the position in the task array at which the task has been added. If the return value is SCH_MAX_TASKS then the task could not be added to the array (there was insufficient space). If the return value is < SCH_MAX_TASKS, then the task was added successfully.

Note: this return value may be required, if a task is to be subsequently deleted - see SCH_Delete_Task().

EXAMPLES:

Task_ID = SCH_Add_Task(Do_X,1000,0);

Causes the function Do_X() to be executed once after 1000 sch ticks.

Task_ID = SCH_Add_Task(Do_X,0,1000);

Causes the function Do_X() to be executed regularly, every 1000 sch ticks.

Task_ID = SCH_Add_Task(Do_X,300,1000);

Causes the function Do_X() to be executed regularly, every 1000 ticks.

Task will be first executed at T = 300 ticks, then 1300, 2300, etc.

```
-----*/  
tByte SCH_Add_Task(void * pFunction(),  
                  const tWord DELAY,  
                  const tWord PERIOD)  
{  
    tByte Index = 0;  
  
    // First find a gap in the array (if there is one)  
    while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))  
    {  
        Index++;  
    }  
  
    // Have we reached the end of the list?  
    if (Index == SCH_MAX_TASKS)  
    {  
        // Task list is full  
        //  
        // Set the global error variable  
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;  
  
        // Also return an error code  
        return SCH_MAX_TASKS;  
    }  
}
```

```

// If we're here, there is a space in the task array
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; // return position of task (to allow later deletion)
}

/*-----*/
SCH_Delete_Task()

Removes a task from the scheduler. Note that this does
*not* delete the associated function from memory:
it simply means that it is no longer called by the scheduler.

TASK_INDEX - The task index. Provided by SCH_Add_Task().

RETURN VALUE: RETURN_ERROR or RETURN_NORMAL

/*-----*/
bit SCH_Delete_Task(const tByte TASK_INDEX)
{
    bit Return_code;

    if (SCH_tasks_G[TASK_INDEX].pTask == 0)
    {
        // No task at this location...
        //

        // Set the global error variable
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;

        // ...also return an error code
        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; // return status
}

```

```
/*-----*
SCH_Report_Status()

Simple function to display error codes.

This version displays code on a port with attached LEDs:
adapt, if required, to report errors over serial link, etc.

Errors are only displayed for a limited period
(60000 ticks = 1 minute at 1ms tick interval).
After this the the error code is reset to 0.

This code may be easily adapted to display the last
error 'for ever': this may be appropriate in your
application.

See Chapter 10 for further information.

-----*/
```

```
void SCH_Report_Status(void)
{
#ifndef SCH_REPORT_ERRORS
    // ONLY APPLIES IF WE ARE REPORTING ERRORS
    // Check for a new error code
    if (Error_code_G != Last_error_code_G)
    {
        // Negative logic on LEDs assumed
        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; // Reset error code
    }
}
}
#endif
}

/*-----*
SCH_Go_To_Sleep()

This scheduler enters 'idle mode' between clock ticks
to save power. The next clock tick will return the processor
to the normal operating state.

Note: a slight performance improvement is possible if this
function is implemented as a macro, or if the code here is simply
pasted into the 'dispatch' function.

However, by making this a function call, it becomes easier
- during development - to assess the performance of the
scheduler, using the 'performance analyser' in the Keil
hardware simulator. See Chapter 14 for examples for this.

*** May wish to disable this if using a watchdog ***

*** ADAPT AS REQUIRED FOR YOUR HARDWARE ***

*-----*/
void SCH_Go_To_Sleep()
{
    PCON |= 0x01; // Enter idle mode (generic 8051 version)

    // Entering idle mode requires TWO consecutive instructions
    // on 80c515 / 80c505 - to avoid accidental triggering
    // PCON |= 0x01;      // Enter idle mode (#1)
    // PCON |= 0x20;      // Enter idle mode (#2)
}

/*-----*
---- END OF FILE -----
*-----*/

```

Listing 14.13 Part of the core scheduler library

Example: A generic scheduler for the 8051/52 with 16-bit timing

This example demonstrates how the core scheduler files (presented in the previous example) can be used as the basis of a complete scheduler. Here, Timer 2 is used to generate the scheduler ‘ticks’.

Note that, because Timer 2 is used, this code cannot be used with the most basic 8051 devices which lack this timer: however, it will run on the great majority of current 8051s. Note also that other schedulers, using T0 and T1 for tick generation, are included on the CD.

The particular example here uses the scheduler to flash an LED on and off (on for one second, off for one second) *ad infinitum*.

All the files are presented here (Listings 14.14 to 14.18), with the exception of the core scheduler files presented in Listings 14.12 and 14.13. All the files, including the core files, are included on the CD.

```
/*-----*
Main.c (v1.00)

-----*
Demonstration program for:
Generic 16-bit auto-reload scheduler (using T2).
Assumes 12 MHz oscillator (-> 01 ms tick interval).
*** All timing is in TICKS (not milliseconds) ***

Required linker options:
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)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Prepare for the 'Flash_LED' task
    LED_Flash_Init();
```

```

// Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
// - timings are in ticks (1 ms tick interval)
// (Max interval / delay is 65535 ticks)
SCH_Add_Task(LED_Flash_Update, 0, 1000);

// Start the scheduler
SCH_Start();

while(1)
{
    SCH_Dispatch_Tasks();
}

/* -----
--- END OF FILE ---
*/

```

Listing 14.14 Part of the library for a generic scheduler for the 8051/52 with 16-bit timing

```

/* -----
LED_flas.H (v1.00)

-----
- See LED_flas.C for details.
* -----
// ----- Public function prototypes -----
void LED_Flash_Init(void);
void LED_Flash_Update(void);

/* -----
--- END OF FILE ---
*/

```

Listing 14.15 Part of the library for a generic scheduler for the 8051/52 with 16-bit timing

```

/* -----
LED_flas.C (v1.00)

-----
```

Simple 'Flash LED' test function for scheduler.

```
- *-----*/  
#include "Main.h"  
#include "Port.h"  
#include "LED_flas.h"  
  
// ----- Private variable definitions -----  
  
static bit LED_state_G;  
  
/*-----*/  
  
LED_Flash_Init()  
  
- See below.  
  
/*-----*/  
void LED_Flash_Init(void)  
{  
    LED_state_G = 0;  
}  
  
/*-----*/  
  
LED_Flash_Update()  
  
Flashes an LED (or pulses a buzzer, etc) on a specified port pin.  
  
Must schedule at twice the required flash rate: thus, for 1 Hz  
flash (on for 0.5 seconds, off for 0.5 seconds) must schedule  
at 2 Hz.  
  
/*-----*/  
void LED_Flash_Update(void)  
{  
    // Change the LED from OFF to ON (or vice versa)  
    if (LED_state_G == 1)  
    {  
        LED_state_G = 0;  
        LED_pin = 0;  
    }  
    else  
    {  
        LED_state_G = 1;  
        LED_pin = 1;  
    }  
}
```

```
/* -----
----- END OF FILE -----
*/
```

Listing 14.16 Part of the library for a generic scheduler for the 8051/52 with 16-bit timing

```
/* -----
2_01_12g.C (v1.00)

-----
*** THIS IS A SCHEDULER FOR STANDARD 8051 / 8052 ***
*** Uses T2 for timing, 16-bit auto reload ***
*** 12 MHz oscillator -> 1 ms (precise) tick interval ***

*/
#include "2_01_12g.h"

// ----- Public variable declarations -----
// The array of tasks (see Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable
//
// See Port.H for port on which error codes are displayed
// and for details of error codes
extern tByte Error_code_G;

/* -----
SCH_Init_T2()

Scheduler initialization function. Prepares scheduler
data structures and sets up timer interrupts at required rate.
Must call this function before using the scheduler.

*/
void SCH_Init_T2(void)
{
    tByte i;

    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }
}
```

```

// Reset the global error variable
// - SCH_Delete_Task() will generate an error code,
// (because the task array is empty)
Error_code_G = 0;

// Now set up Timer 2
// 16-bit timer function with automatic reload

// Crystal is assumed to be 12 MHz
// The Timer 2 resolution is 0.000001 seconds (1 µs)
// The required Timer 2 overflow is 0.001 seconds (1 ms)
// - this takes 1000 timer ticks
// Reload value is 65536 - 1000 = 64536 (dec) = 0xFC18

T2CON = 0x04;      // load Timer 2 control register
T2MOD = 0x00;      // load Timer 2 mode register

TH2     = 0xFC;    // load Timer 2 high byte
RCAP2H  = 0xFC;    // load Timer 2 reload capture reg, high byte
TL2     = 0x18;    // load Timer 2 low byte
RCAP2L  = 0x18;    // load Timer 2 reload capture reg, low byte

ET2     = 1;       // Timer 2 interrupt is enabled
TR2     = 1;       // Start Timer 2
}

/*-----*-
SCH_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

-----*/
void SCH_Start(void)
{
    EA = 1;
}

/*-----*-
SCH_Update()

This is the scheduler ISR. It is called at a rate
determined by the timer settings in the 'init' function.

```

```

This version is triggered by Timer 2 interrupts:
timer is automatically reloaded.

*-----*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;

    TF2 = 0; // Have to manually clear this.

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Inc. the 'RunMe' flag

                if (SCH_tasks_G[Index].Repeat)
                {
                    // Schedule regular tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Repeat;
                }
            }
            else
            {
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }

    /*-----*
     *----- END OF FILE -----*
     *-----*/
}

```

Listing 14.17 Part of the library for a generic scheduler for the 8051/52 with 16-bit timing

```
/*-----*  
2_01_12g.C (v1.00)  
-----*  
*** THIS IS A SCHEDULER FOR STANDARD 8051 / 8052 ***  
*** Uses T2 for timing, 16-bit auto reload ***  
*** 12 MHz oscillator -> 1 ms (precise) tick interval ***  
-*-----*/  
  
#include "2_01_12g.h"  
  
// ----- Public variable declarations -----  
  
// The array of tasks (see Sch51.C)  
extern sTask SCH_tasks_G[SCH_MAX_TASKS];  
  
// The error code variable  
//  
// See Port.H for port on which error codes are displayed  
// and for details of error codes  
extern tByte Error_code_G;  
  
/*-----*/  
  
SCH_Init_T2()  
  
Scheduler initialization function. Prepares scheduler  
data structures and sets up timer interrupts at required rate.  
Must call this function before using the scheduler.  
-*-----*/  
  
void SCH_Init_T2(void)  
{  
    tByte i;  
  
    for (i = 0; i < SCH_MAX_TASKS; i++)  
    {  
        SCH_Delete_Task(i);  
    }  
  
    // Reset the global error variable  
    // - SCH_Delete_Task() will generate an error code,  
    // (because the task array is empty)  
    Error_code_G = 0;  
  
    // Now set up Timer 2  
    // 16-bit timer function with automatic reload
```

```

// Crystal is assumed to be 12 MHz
// The Timer 2 resolution is 0.000001 seconds (1 µs)
// The required Timer 2 overflow is 0.001 seconds (1 ms)
// - this takes 1000 timer ticks
// Reload value is 65536 - 1000 = 64536 (dec) = 0xFC18

T2CON = 0x04;      // load Timer 2 control register
T2MOD = 0x00;      // load Timer 2 mode register

TH2     = 0xFC;    // load Timer 2 high byte
RCAP2H = 0xFC;    // load Timer 2 reload capture reg, high byte
TL2     = 0x18;    // load Timer 2 low byte
RCAP2L = 0x18;    // load Timer 2 reload capture reg, low byte

ET2     = 1;      // Timer 2 interrupt is enabled

TR2     = 1;      // Start Timer 2
}

/*-----*/
SCH_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

/*-----*/
void SCH_Start(void)
{
    EA = 1;
}
/*-----*/
SCH_Update()

This is the scheduler ISR. It is called at a rate
determined by the timer settings in the 'init' function.

This version is triggered by Timer 2 interrupts:
timer is automatically reloaded.

/*-----*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
}

```

```

    TF2 = 0; // Have to manually clear this.

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        //
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            //
            if (SCH_tasks_G[Index].Delay == 0)
            {
                //
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Inc. the 'RunMe' flag

                if (SCH_tasks_G[Index].Period)
                {
                    //
                    // Schedule regular tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                //
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }

    /*-----*
     *----- END OF FILE -----*
     *-----*/

```

Listing 14.18 Part of the library for a generic scheduler for the 8051/52 with 16-bit timing

Example: Further 8051 Schedulers

See CD for full details of a wide range of different 8051 schedulers.

Further reading

Please refer to p. 253 for a number of suitable suggestions for further reading in this area.

chapter 15

Learning to think co-operatively

Introduction

Using a co-operative scheduler in your application has a number of benefits, one of which being that the development process is simplified. However, to get the maximum benefit from the scheduler you need to learn to think ‘co-operatively’.

For example, one key difference between scheduled and desktop applications is the need to think carefully about issues of timing and task duration. More specifically, as we saw in Chapter 14, a key requirement in applications using a co-operative scheduler is that – for all tasks, under all circumstances – the task duration, $Duration_{Task}$, must satisfy the following condition:

$$Duration_{Task} < Tick\ Interval$$

The patterns in this chapter are intended to help you meet this condition by ensuring that tasks will abort if they cannot complete within a specified period of time. Specifically, two timeout patterns are presented here:

- **LOOP TIMEOUT** [page 298]
- **HARDWARE TIMEOUT** [page 305]

LOOP TIMEOUT

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you ensure that your system will not ‘hang’ while waiting for a hardware operation (such as an AD conversion or serial data transfer) to complete?

Background

The Philips 8XC552 is an Extended 8051 device with a number of on-chip peripherals, including an 8-channel, 10-bit ADC. Philips provide an application note (AN93017) that describes how to use this microcontroller. This application note includes the following code:

```
// Wait until AD conversion finishes (checking ADCI)
while ((ADCON & ADCI) == 0);
```

Such code is potentially unreliable, because there are circumstances under which our application may ‘hang’. This might occur for one or more of the following reasons:

- If the ADC has been incorrectly initialized, we cannot be sure that a data conversion will be carried out.
- If the ADC has been subjected to an excessive input voltage, then it may not operate at all.
- If the variables ADCON or ADCI were not correctly initialized, they may not operate as required.

Such problems are not, of course, unique to this particular microcontroller or even to ADCs. Such code is common in embedded applications.

If your application is to be reliable, you need to be able to guarantee that no function will hang in this way. Loop timeouts offer a simple but effective means of providing such a guarantee.

Solution

A loop timeout may be easily created. The basis of the code structure is a software delay, created as follows:

```
unsigned integer Timeout_loop = 0;
...
while (++Timeout_loop);
```

This loop will keep running until the variable `Timeout_loop` reaches its maximum value (assuming 16-bit integers) of 65,535 and then overflows. When this happens, the program will continue. Note that, without some simulation studies or prototyping, we cannot easily determine how long this delay will be. However, we do know that the loop will, eventually, time out.

Such a loop is not terribly useful. However, if we consider again the ADC example given in ‘Background’, we can easily extend this idea. Recall that the original code was as follows:

```
// Wait until AD conversion finishes (checking ADCI)
while ((ADCON & ADCI) == 0);
```

Here is a modified version of this code, this time incorporating a loop timeout:

```
tWord Timeout_loop = 0;

// Take sample from ADC
// Wait until conversion finishes (checking ADCI)
// - simple loop timeout
while (((ADCON & ADCI) == 0) && (++Timeout_loop != 0));
```

Note that this alternative implementation is also useful:

```
tWord Timeout_loop = 1;

// Take sample from ADC
// Wait until conversion finishes (checking ADCI)
// - simple loop timeout
while (((ADCON & ADCI) == 0) && (Timeout_loop != 0))
{
    Timeout_loop++; // Disable for use in hardware simulator...
}
```

The advantage of this second technique is that the loop timeout may be easily commented out, if required, when executing the code on a hardware simulator.

In both cases, we now know that the loop cannot go on ‘for ever’.

Note that we can vary the duration of the loop timeout by changing the initial value loaded into loop variable. The file `TimeoutL.H`, reproduced in Listing 15.1 and included on the CD in the directory associated with this chapter, includes a set of constants that give, very approximately, the specified timeout values.

```

/*-----*
TimeoutL.H (v1.00)

-----
Simple loop timeout delays for the 8051 family based.

* THESE VALUES ARE NOT PRECISE - YOU MUST ADAPT TO YOUR SYSTEM *

-*-----*/
// ----- Public constants -----
// Vary this value to change the loop duration
// THESE ARE APPROX VALUES FOR VARIOUS TIMEOUT DELAYS
// ON 8051, 12 MHz, 12 Osc / cycle
// *** MUST BE FINE TUNED FOR YOUR APPLICATION ***
// *** Timings vary with compiler optimization settings ***
#define LOOP_TIMEOUT_INIT_001ms 65435
#define LOOP_TIMEOUT_INIT_010ms 64535
#define LOOP_TIMEOUT_INIT_500ms 14535

/*-----*
--- END OF FILE
-*-----*/

```

Listing 15.1 The file TimeoutL.H.

We give an example of how to use this file in the following sections.

Hardware resource implications

LOOP TIMEOUT does not use a timer and imposes an almost negligible CPU and memory load.

Reliability and safety implications

Using a **LOOP TIMEOUT** can result in a huge reliability and safety improvement at minimal cost. However, if practical, **HARDWARE TIMEOUT** [page 305] is usually an even better solution.

Portability

Loop timeouts will work in any environment. However, the timings obtained will vary dramatically between microcontrollers and compilers.

Overall strengths and weaknesses

- ☺ Much better than executing code without any form of timeout protection.
- ☺ Many applications use a timer for RS232 baud rate generation, and another timer to run the scheduler. In many 8051 devices, this leaves no further timers available to implement a **HARDWARE TIMEOUT** [page 305]. In these circumstances, use of a loop is the only practical way of implementing effective timeout behaviour.
- ⌚ Timings are difficult to calculate and timer values are not portable. **HARDWARE TIMEOUT** is always a better solution, if you have a spare timer available.

Related patterns and alternative solutions

As mentioned under 'Reliability and safety implications', **HARDWARE TIMEOUT** [page 305] is often a better alternative to **LOOP TIMEOUT**.

In addition, **HARDWARE WATCHDOG** [page 217] provides an alternative; however, it is rather crude by comparison and detects errors at the application (rather than task) level.

Example: Test program for loop timeout code

As noted, loop timeouts must be carefully hand-tuned to give accurate delay values.

The program in Listing 15.2 can be used to test such timeout code.

```
/* -----
Main.C
-----
Testing timeout loops.

*/
#include "Main.H"
#include "TimeoutL.H"

// Function prototypes
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;

    // Simple loop timeout...
    while (++Timeout_loop != 0);
}

/* -----
void Test_10ms(void)
{
    tWord Timeout_loop = LOOP_TIMEOUT_INIT_010ms;

    // Simple loop timeout...
    while (++Timeout_loop != 0);
}

/* -----
void Test_500ms(void)
{
    tWord Timeout_loop = LOOP_TIMEOUT_INIT_500ms;

    // Simple loop timeout...
    while (++Timeout_loop != 0);
}

/* -----
--- END OF FILE ---
*/

```

Listing 15.2 Testing the timeout code

The program is run in the Keil hardware simulator to check the timings (Figure 15.1).

Remember: Changes in compiler optimization settings – and even apparently unconnected changes to the rest of the program – can change these timings, because they alter the way in which the compiler makes use of the available memory areas.

For a final test in the pre-production code, set a port pin high at start of the timeout and clear it at the end. Use an oscilloscope to measure the resulting delay.

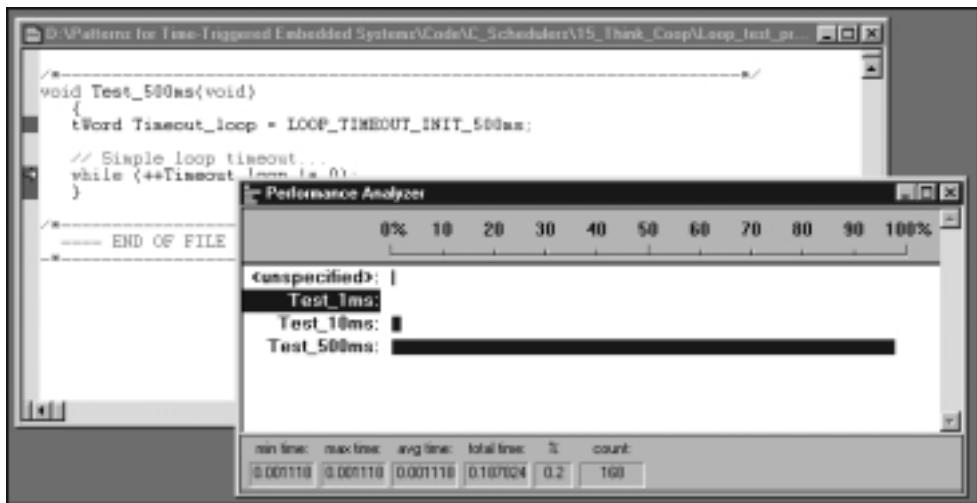


FIGURE 15.1 Testing the timeout code using the Keil hardware simulator

Example: Loop timeouts in an I²C library

We discuss the I²C bus in detail in Chapter 23. Very briefly, I²C is a two-wire serial bus. The two wires are referred to as the serial data (SDA) and serial clock (SCL) lines (Figure 15.2). When the bus is free, both SCL and SDA lines are HIGH.

Here we consider how loop timeouts are used in a version of the I²C library.

At certain stages in the data transmission, we need to 'synchronize the clocks'. This means waiting for the 'clock' line to be pulled high (by a slave device). Some I²C code libraries include fragments of code similar to the following to achieve this:

```
// Synchronize the clock
while (_I2C_SCL == 0);
```

Of course, for all of the reasons discussed in this pattern, this is a dangerous approach.

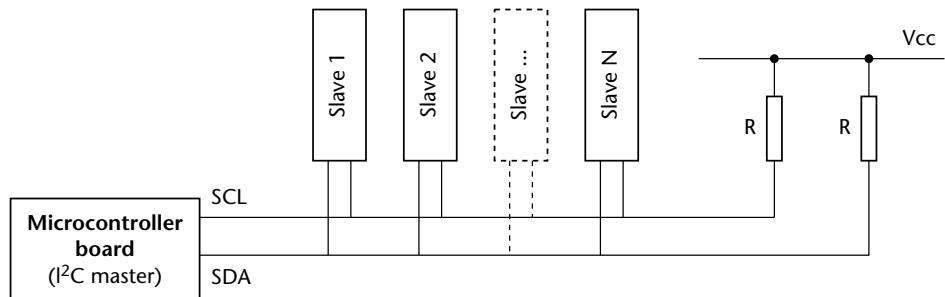


FIGURE 15.2 The I²C bus

The following code fragment uses a loop timeout to improve this code with a 1 ms timeout:

```
#define LOOP_TIMEOUT_INIT_001ms 65435  
...  
tLong Timeout_loop = LOOP_TIMEOUT_INIT_001ms;  
...  
// Try to synchronize the clock  
while ((I2C_SCL == 0) && (++Timeout_loop));  
  
if (!Timeout_loop)  
{  
    return 1; // Error - Timeout condition failed  
}
```

Please refer to Chapter 23 for further details of the I²C bus and this library.

Further reading

HARDWARE TIMEOUT

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you produce well-defined timeout behaviour so that, for example, you can respond within exactly 0.5 ms if an expected event does not occur?

Background

See **LOOP TIMEOUT** [page 298] for relevant background material.

Solution

As we saw in **HARDWARE DELAY** [page 194], we can create portable and easy to use delay code for the 8051 family as follows:

```
// Define Timer 0 / Timer 1 reload values for ~1 msec delay
#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;

    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    for(ms = 0; ms < MS; ms++)
    {
        // Note - delay value is *approximately* 1 ms per loop
        // - see Delay_T0.H for details of PRELOAD_values.
```

```

TH0 = PRELOAD_01ms_H;
TL0 = PRELOAD_01ms_L;

TF0 = 0;           // clear overflow flag
TR0 = 1;           // start timer 0

while (TF0 == 0);    // Loop until Timer 0 overflows (TF0 == 1)

TR0 = 0;           // Stop Timer 0
}
}

```

HARDWARE TIMEOUT involves a simple variation on this technique and allows precise timeout delays to be easily generated.

For example, in **LOOP TIMEOUT** [page 298] we considered the process of reading from an ADC in a Philips 8XC552 microcontroller.

This was the original, potentially dangerous, code:

```
// Wait until AD conversion finishes (checking ADCI)
while ((ADCON & ADCI) == 0);
```

This was a possible application of **LOOP TIMEOUT** to this problem:

```
tWord Timeout_loop = 0;

// Take sample from A-D
// Wait until AD conversion finishes (checking ADCI)
// - simple loop timeout
while (((ADCON & ADCI) == 0) && (++Timeout_loop));
```

Applying **LOOP TIMEOUT** can result in a significant improvement in the reliability of this code, but determining the duration of the timeout is not a trivial process.

Here is an alternative solution, providing a delay of 10 ms which will, with reasonable accuracy, apply across the whole 8051 family (without code modifications):

```
// Configure Timer 0 as a 16-bit timer
TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

ET0 = 0; // No interrupts

// Simple timeout feature - approx 10 ms
TH0 = PRELOAD_10ms_H; // See Timeout.H for PRELOAD details
TL0 = PRELOAD_10ms_L;
TF0 = 0; // Clear flag
TR0 = 1; // Start timer

while (((ADCON & ADCI) == 0) && !TF0);
```

Various portable **Ppreload_** macros are given in the file **Timeout.H** reproduced in Listing 15.3 and included on the CD.

Note that the same PRELOAD_ values may be used with either Timer 0 or Timer 1, as required.

```
/*-----*
 * Timeout.H (v1.00)
 *
-----*

Simple timeout delays for the 8051 family based on T0/T1.

Timer must be correctly configured to use these values:
See Chapter 11 for details.

*-----*/
// ----- Public constants -----
// Timer T_ values for use in simple (hardware) timeouts
// - Timers are 16-bit, manual reload ('one shot').
//
// NOTE: These macros are portable but timings are *approximate*
//       and *must* be checked by hand if accurate timing is required.
//
// Define initial Timer 0 / Timer 1 values for ~50 µs delay
#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)

// Define initial Timer 0 / Timer 1 values for ~500 µs delay
#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)

// Define initial Timer 0 / Timer 1 values for ~1 msec delay
#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)
//

// Define initial Timer 0 / Timer 1 values for ~10 msec delay
#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)
//

// Define initial Timer 0 / Timer 1 values for ~30 msec delay
#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)
```

```
/*-----*  
----- END OF FILE -----  
-----*/
```

Listing 15.3 The file Timeout.H

Hardware resource implications

HARDWARE TIMEOUT requires the use of a timer.

Reliability and safety implications

HARDWARE TIMEOUT is the most reliable form of timeout structure we consider in the book.

Portability

Like all timer-based patterns, this code may be easily ported to other members of the 8051 family. It may also be ported to other microcontrollers.

Overall strengths and weaknesses

- 😊 Accurate timeout delays may be obtained using **HARDWARE TIMEOUT**.
- 😢 The number of timers available is very limited: however, when using a co-operative scheduler, the tasks are running co-operatively and the same timer may be used in several tasks at the same time.

Related patterns and alternative solutions

See **LOOP TIMEOUT** [page 298] for an alternative that does not require the use of any timer hardware.

In addition, **HARDWARE WATCHDOG** [page 217] provides an alternative; however, it is rather crude by comparison and detects errors at the application (rather than task) level.

Example: Testing hardware timeouts

Listing 15.4 illustrates the delays obtained with some hardware timeouts using the Keil hardware simulator (see also Figure 15.3).

```
/*-----*  
----- Main.C -----*
```

```
Testing timeout loops.

*-----*/
#include "Main.H"
#include "TimeoutH.H"

// Function prototypes
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 code variable & TIMEOUT code (dummy here)
#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)
{
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts
}
```

```
// Simple timeout feature - approx 50 µs
TH0 = T_50micros_H; // See TimeoutH.H for T_ details
TL0 = T_50micros_L;
TF0 = 0; // Clear flag
TR0 = 1; // Start timer

while (!TF0);

TR0 = 0;

// Normally need to report timeout TIMEOUTs
// (this test is for demo purposes here)
if (TF0 == 1)
{
    //
    // Operation timed out
    Error_code_G = TIMEOUT;
}
}

/*-----*/
void Test_500micros(void)
{
    //
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 500 µs
    TH0 = T_500micros_H; // See TimeoutH.H for T_ details
    TL0 = T_500micros_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    while (!TF0);

    TR0 = 0;

    // Normally need to report timeout TIMEOUTs
    // (this test is for demo purposes here)
    if (TF0 == 1)
    {
        //
        // Operation timed out
        Error_code_G = TIMEOUT;
    }
}
```

```
/*-----*/
void Test_1ms(void)
{
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 1 ms
    TH0 = T_01ms_H; // See TimeoutH.H for T_ details
    TL0 = T_01ms_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    while (!TF0);

    TR0 = 0;

    // Normally need to report timeout TIMEOUTs
    // (this test is for demo purposes here)
    if (TF0 == 1)
    {
        // Operation timed out
        Error_code_G = TIMEOUT;
    }
}

/*-----*/
void Test_5ms(void)
{
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 10 ms
    TH0 = T_05ms_H; // See TimeoutH.H for T_ details
    TL0 = T_05ms_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    while (!TF0);

    TR0 = 0;

    // Normally need to report timeout TIMEOUTs
```

```
// (this test is for demo purposes here)
if (TF0 == 1)
{
    //
    // Operation timed out
    Error_code_G = TIMEOUT;
}
}

/*-----*/
void Test_10ms(void)
{
    //
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 10 ms
    TH0 = T_10ms_H; // See TimeoutH.H for T_ details
    TL0 = T_10ms_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    while (!TF0);

    TR0 = 0;

    // Normally need to report timeout TIMEOUTs
    // (this test is for demo purposes here)
    if (TF0 == 1)
    {
        //
        // Operation timed out
        Error_code_G = TIMEOUT;
    }
}

/*-----*/
void Test_15ms(void)
{
    //
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 10 ms
    TH0 = T_15ms_H; // See TimeoutH.H for T_ details
```

```
TL0 = T_15ms_L;
TF0 = 0; // Clear flag
TR0 = 1; // Start timer

while (!TF0);

TR0 = 0;

// Normally need to report timeout TIMEOUTs
// (this test is for demo purposes here)
if (TF0 == 1)
{
    //
    // Operation timed out
    Error_code_G = TIMEOUT;
}
}

/*-----*/
void Test_20ms(void)
{
    //
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 10 ms
    TH0 = T_20ms_H; // See TimeoutH.H for T_ details
    TL0 = T_20ms_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    while (!TF0);

    TR0 = 0;

    // Normally need to report timeout TIMEOUTs
    // (this test is for demo purposes here)
    if (TF0 == 1)
    {
        //
        // Operation timed out
        Error_code_G = TIMEOUT;
    }
}

/*-----*/
void Test_50ms(void)
{
```

```

// Configure Timer 0 as a 16-bit timer
TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

ET0 = 0; // No interrupts

// Simple timeout feature - approx 10 ms
TH0 = T_50ms_H; // See TimeoutH.H for T_ details
TL0 = T_50ms_L;
TF0 = 0; // Clear flag
TR0 = 1; // Start timer

while (!TF0);

TR0 = 0;

// Normally need to report timeout TIMEOUTs
// (this test is for demo purposes here)
if (TF0 == 1)
{
    //
    // Operation timed out
    Error_code_G = TIMEOUT;
}
}

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 15.4 Testing the hardware timeouts

Example: Generating timeout-based delays

The ‘timeout’ technique may be readily extended to create an alternative implementation of **HARDWARE DELAY** [page 194]: see Listing 15.5.

```

void Delay_50micros(void)
{
    //
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 50 µs
    TH0 = T_50micros_H; // See TimeoutH.H for T_ details
}

```

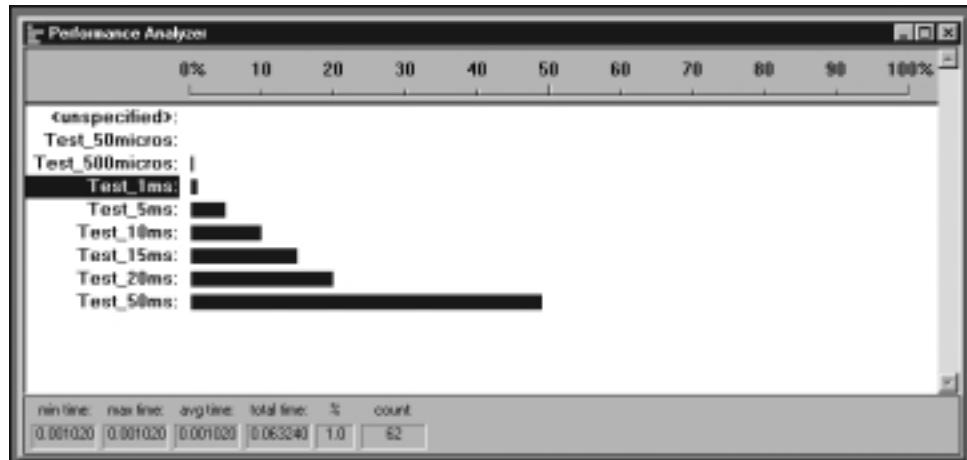


FIGURE 15.3 Testing the hardware timeouts in the Keil hardware simulator

```
TL0 = T_50micros_L;  
TF0 = 0; // Clear flag  
TR0 = 1; // Start timer  
  
while (!TF0);  
  
TR0 = 0;  
}
```

Listing 15.5 Using hardware timeouts to implement delays

Further reading

Task-oriented design

Introduction

Two patterns in this chapter encapsulate key design characteristics that underlie many successful co-operatively scheduled applications:

- **MULTI-STAGE TASK** [page 317] considers techniques that may be used to convert long tasks (scheduled at infrequent intervals) into much shorter tasks (scheduled at frequent intervals)
- **MULTI-STATE TASK** [Page 322] considers techniques that may be used to replace multiple tasks with a single task that performs different activities depending on the current state of the system.

MULTI-STAGE TASK

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you make sure that a long task does not interfere with the normal operation of the scheduler?

Background

See **CO-OPERATIVE SCHEDULER** [page 255] for relevant background information.

Solution

As we have seen, co-operative schedulers have many advantages when compared to pre-emptive or even hybrid schedulers and we generally wish to use a co-operative approach when it is feasible to do so. However, a key design challenge when using this approach is to ensure that each task is completed before the next scheduler tick occurs. One approach that can help us to achieve this goal is to make use of timeouts (see Chapter 15).

Another approach is to use of multi-stage tasks. To understand the need for multi-stage tasks, consider an example (adapted from Pont 1996, Chapter 13). Suppose that as part of a 'backup' temperature monitoring system for a metal furnace we are required to send temperature samples at regular, 5-second, intervals to a desktop PC (Figure 16.1).

The required PC output takes the form shown in Figure 16.2.

At first inspection, the obvious way of sending this information to the PC at the required 5-second interval is to create a function (we will call it `Send_Temp_Data()`), which will be run by the scheduler every five seconds.

However, this is probably a rather bad solution. The problem is that we need to send at least 43 characters to the PC using `Send_Temp_Data()`. At a common baud rate of 9,600 (see Chapter 18 for details), each of the characters will take approximately 1 millisecond to send: as a result, the task duration will be around 40 ms. If we use this (long) task, we will need to set the system tick interval at a value greater than 40 ms, which may have a detrimental impact on the overall responsiveness of the application.

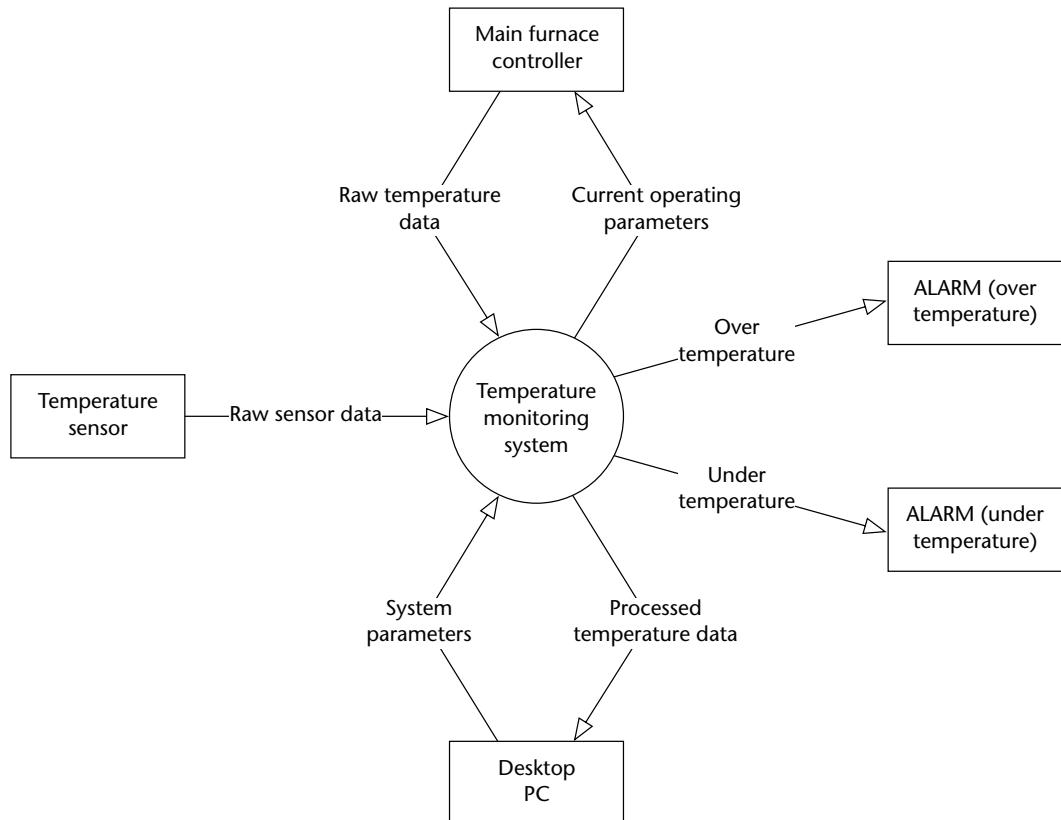


FIGURE 16.1 Part of the design for a temperature-monitoring system

```

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
  
```

FIGURE 16.2 Output from a temperature-monitoring system

The multi-stage alternative solution avoids this problem. Rather than sending all the data at once, we can simply store the data we want to send to the PC in a buffer. Every ten milliseconds (say) we schedule a task to check the buffer and send the next character (if there is one ready to send). In this way, all the required 43 characters of data will be sent to the PC within 0.5 seconds: we can reduce this time if necessary (it rarely is) by scheduling the task to check the buffer every millisecond. Note that

because we do not have to wait for each character to be sent, the process of sending data from the buffer will be very fast (typically a fraction of a millisecond).

Overall, the use of multi-stage tasks in this application allows us the opportunity to use a much shorter tick interval and therefore make more efficient use of the microcontroller's processing power.

The 'RS232' library used as an example here is discussed in detail in Chapter 18.

Hardware resource implications

In general, the use of multi-stage tasks allows much more efficient use of the available microcontroller processing power.

Reliability and safety implications

Use of multi-stage tasks can make your system more responsive, by allowing the user of a shorter tick interval.

Portability

This technique can be (and is) applied in a wide range of different embedded systems.

Overall strengths and weaknesses

- ☺ The use of multi-stage tasks allows the use of shorter tick intervals and, hence, can help to make the system more responsive.
- ☺ The use of multi-stage tasks allows much more efficient use of the available microcontroller processing power.

Related patterns and alternative solutions

This basic architecture is applied in various patterns in this collection, including **PC LINK (RS-232)** [page 362], **LCD CHARACTER PANEL** [page 467] and **SWITCH INTERFACE (SOFTWARE)** [page 399].

Example: Measuring rotational speed

Suppose we wish to measure the speed of a rotating shaft and display the results on a (multiplexed) LED display. This could be part of an automotive or industrial application.

As we will see in Chapter 30, an effective way of measuring the speed is to attach an appropriate rotary encoder to the shaft (Figure 16.3), and count the number of pulses that occur over a fixed period of time (say 100 ms or 1 second). From the count, and having details of the rotary encoder, we can calculate the average speed of rotation.

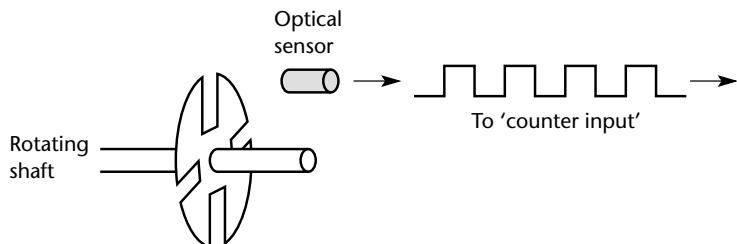


FIGURE 16.3 Measuring rotational speed

We have seen various people try to apply this basic approach in a scheduled environment as follows:

- Create a task of (say) 100 ms duration
- In this task, count the pulses
- Set the value of a (global) `Pulse_count_G` variable at the end of the task
- Use the global variable to update the LED display

The problem with this approach is that 100 ms taken to measure the speed is a **long** task duration and can be difficult to support in many embedded applications. Here, for example, we suggested that we wished to display the speed on a multiplexed LED display. We might well have to update the LEDs every 5 ms to avoid flickering (see **MX LED DISPLAY** [page 450] for details).

Of course, a co-operative scheduler cannot support tasks called every 5 ms **and** tasks of 100 ms duration.

To solve this problem, consider a multi-stage approach to this problem. In the first approach, we waited in the task to count the pulses from the encoder. However, this is not necessary. Timer 0 or Timer 1 (or Timer 2 where available) will count pulses (strictly, falling edges of pulses) on external pins without user intervention, without generating interrupts and without interfering with any other processing. We can use this fact to make this speed measurement system into a multi-stage task as follows:

- Create a very short (<0.1 ms task)
- Schedule this task (say every 100 ms, to be compatible with the first example)
- In this task, read the current pulse count. Store the result in a global variable (for use elsewhere in the program)
- Reset the pulse count to 0
- Repeat 100 ms later etc.

The details of the technique will be considered in Chapter 30, where we will see that it is possible for similar solutions to be created without the need for the timer hardware.

Example: LCD library

Suppose we wish to update an LCD display. As we will see in [LCD CHARACTER PANEL](#) [page 467], updating each character in a typical LCD display can take around 0.5 ms: to update the whole of a 40-character display can therefore take around 20 ms. This is often prohibitively long.

However, suppose we schedule a `LCD_Update()` function every 20 ms and, each time, update only a single display position. At worst, it will take us a total of 800 ms to update the whole display – and we will be able to complete numerous other tasks at the same time. In addition, in most circumstances, only some of the display will have changed: if we keep track of characters that need to be updated, we can usually keep the display fully up to date with a multi-stage task of 0.5 ms duration called every 100 ms. Overall, the multi-stage approach to this single task can make a major difference to the architecture of the whole application.

Full details of the multi-stage LCD library are given in [LCD CHARACTER PANEL](#) [page 467].

Further reading

MULTI-STATE TASK

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you replace multiple tasks in an application with a single task that performs different activities depending on the current state of the system (and why is it, sometimes, a good idea to do so)?

Background

See **CO-OPERATIVE SCHEDULER** [page 255] and **MULTI-STATE TASK** [page 317] for relevant background information.

Solution

MULTI-STATE TASK encapsulates a system architecture that is apparent in many well-designed embedded applications.

To understand the need for this architecture, consider a simple washing machine control system (Figure 16.4).

Here is a brief description of the way in which we expect the system to operate:

- 1 The user selects a wash program (e.g. 'Wool', 'Cotton') on the selector dial.
- 2 The user presses the 'Start' switch.
- 3 The door lock is engaged.
- 4 The water valve is opened to allow water into the wash drum.
- 5 If the wash program involves detergent, the detergent hatch is opened. When the detergent has been released, the detergent hatch is closed.
- 6 When the 'full water level' is sensed, the water valve is closed.
- 7 If the wash program involves warm water, the water heater is switched on. When the water reaches the correct temperature, the water heater is switched off.
- 8 The washer motor is turned on to rotate the drum. The motor then goes through a series of movements, both forward and reverse (at various speeds) to wash the clothes. (The precise set of movements carried out depends on the wash program that the user has selected.) At the end of the wash cycle, the motor is stopped.
- 9 The pump is switched on to drain the drum. When the drum is empty, the pump is switched off.

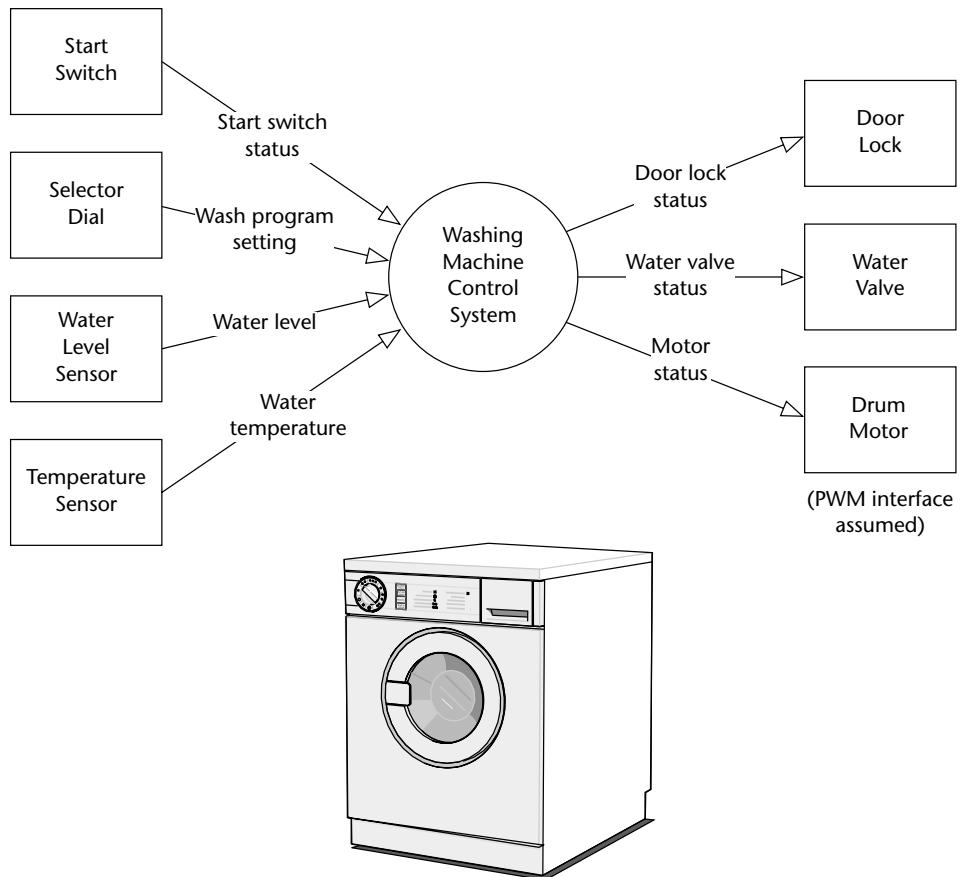


FIGURE 16.4 A possible context diagram for the control system in a domestic washing machine

The description is simplified for the purposes of this example, but it will be adequate for our purposes here.

Based on this description we will try to identify some of the functions that will be required to implement this system. A provisional list might be as shown in Figure 16.5.

- | | |
|---|--|
| <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_Valve() |
|---|--|

FIGURE 16.5 A provisional list of functions that could be used to develop a washing-machine control system

Now, suppose we wish to identify the **tasks** to be scheduled (co-operatively) in order to implement this application. Based on our list, it may be tempting to conclude that each of the functions listed in Figure 16.5 should become a task in the system. While it would be possible to work in this way it would be likely to lead to a complex and cumbersome system implementation.

To see why this is so, take one example: the function `Control_Water_Heater()`. We want to heat the water only at particular times during the wash cycle. Therefore, if we want to treat this as a task and schedule it, say every 100 ms, we need to creation an implementation something like the following:

```
void TASK_Control_Water_Heater(void)
{
    if (Switch_on_water_heater_G == 1)
    {
        Water_heater = ON;
        return;
    }

    // Switch off heater
    Water_pin = OFF;
}
```

What this task does when it is executed is to check a flag: if it is necessary to heat the water, it starts to do so: otherwise, it stops the heating process.

There are two problems with creating the program in this way:

- We are going to end up with large numbers of tasks (very large numbers in a more substantial application), most of which – like this task – actually do very little. In applications without external memory this is a particular problem, because each task will consume some of the limited memory (RAM) resources.
- It is not at all clear which, if any, of these tasks will actually set the flag (`Switch_on_water_heater_G`), or the other similar flags that will be required in the other tasks in this application.

In practice, what we require in this and many similar applications is a single ‘System Update’ task: this, as we will see, is a task that will be regularly scheduled and will, where necessary, call functions such as `Control_Water_Heater()` as and when required.

In the washing machine, this system update task will look something like the code in Listing 16.1.

```
/* -----
void WASHER_Update(void)
{
    static tWord Time_in_state;

    switch (System_state_G)
    {
```

```
case START:  
{  
// For demo purposes only  
P1 = (tByte) System_state_G;  
  
// Lock the door  
WASHER_Control_Door_Lock(ON);  
  
// Start filling the drum  
WASHER_Control_Water_Valve(ON);  
  
// Release the detergent (if any)  
if (Detergent_G[Program_G] == 1)  
{  
    WASHER_Control_Detergent_Hatch(ON);  
}  
  
// Ready to go to next state  
System_state_G = FILL_DRUM;  
Time_in_state_G = 0;  
  
break;  
}  
  
case FILL_DRUM:  
{  
// For demo purposes only  
P1 = (tByte) System_state_G;  
  
// Remain in this state until drum is full  
// NOTE: Timeout facility included here  
if (++Time_in_state_G >= MAX_FILL_DURATION)  
{  
    // Should have filled the drum by now...  
    System_state_G = ERROR;  
}  
  
// Check the water level  
if (WASHER_Read_Water_Level() == 1)  
{  
    // Drum is full  
  
    // Does the program require hot water?  
    if (Hot_Water_G[Program_G] == 1)  
    {  
        WASHER_Control_Water_Heater(ON);  
    }  
}
```

```
// Ready to go to next state
System_state_G = HEAT_WATER;
Time_in_state_G = 0;
}
else
{
// Using cold water only
// Ready to go to next state
System_state_G = WASH_01;
Time_in_state_G = 0;
}
break;
}

case HEAT_WATER:
{
// For demo purposes only
P1 = (tByte) System_state_G;

// Remain in this state until water is hot
// NOTE: Timeout facility included here
if (++Time_in_state_G >= MAX_WATER_HEAT_DURATION)
{
// Should have warmed the water by now...
System_state_G = ERROR;
}

// Check the water temperature
if (WASHER_Read_Water_Temperature() == 1)
{
// Water is at required temperature
// Ready to go to next state
System_state_G = WASH_01;
Time_in_state_G = 0;
}

break;
}

case WASH_01:
{
// For demo purposes only
P1 = (tByte) System_state_G;
```

```
// All wash program involve WASH_01
// Drum is slowly rotated to ensure clothes are fully wet
WASHER_Control_Motor(ON);

if (++Time_in_state >= WASH_01_DURATION)
{
    System_state_G = WASH_02;
    Time_in_state = 0;
}

break;
}

// REMAINING WASH PHASES OMITTED HERE ...

case WASH_02:
{
    // For demo purposes only
    P1 = (tByte) System_state_G;

    break;
}

case ERROR:
{
    // For demo purposes only
    P1 = (tByte) System_state_G;

    break;
}
}
```

Listing 16.1 A possible implementation of the single task used to implement a washing-machine control system

Listing 16.1 is a representative example of a **MULTI-STATE TASK**.

We can describe the simplest form of this architecture as follows:

- The system involves the use of a number of different functions.
- The functions are always called in the same sequence.
- The functions are called from a single task, as required.

Note that variations on this theme are also common: for example, the functions may not always be called in the same sequence: the precise sequence followed (and

the particular set of functions called) will frequently depend on user preferences or on some other system inputs.

Hardware resource implications

This architecture makes very efficient use of system resources.

Reliability and safety implications

There are no specific reliability or safety implications.

Portability

This high-level pattern is highly portable.

Overall strengths and weaknesses

- ☺ **MULTI-STAGE TASK** encapsulates a simple architecture that matches the needs of many embedded applications

Related patterns and alternative solutions

MULTI-STAGE TASK combined with **ONE-TASK SCHEDULER** [page 911] – and / or with **ONE-YEAR SCHEDULER** [page 919] provides a very simple and efficient system architecture with minimal CPU, memory and power requirements.

Example: Traffic lights

Suppose we wish to create a system for driving three traffic light bulbs. The conventional 'red', 'amber' and 'green' bulbs will be used, with the usual sequencing (Figure 16.6).

Listing 16.2 shows how we can create a multi-state task to achieve this. The 'Update' function is intended to be scheduled every second.

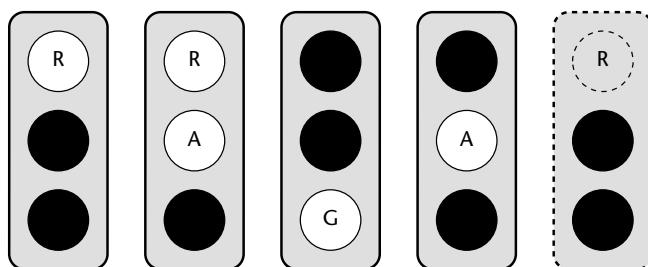


FIGURE 16.6 The required light sequence from red, amber and green bulbs in a traffic-light application

```
/*-----*  
T_lights.C (v1.00)  
-----  
Traffic light control program (Test Version 1.0)  
*-----*/  
#include "Main.h"  
#include "Port.h"  
  
#include "T_lights.h"  
  
// ----- Private constants -----  
  
// Easy to change logic here  
#define ON 0  
#define OFF 1  
  
// Times in each of the (four) possible light states  
// (Times are in seconds - must call the update task once per second)  
//  
#define RED_DURATION (10)  
#define RED_AND_AMBER_DURATION (10)  
  
// NOTE:  
// GREEN_DURATION must equal RED_DURATION  
// AMBER_DURATION must equal RED_AND_AMBER_DURATION  
#define GREEN_DURATION RED_DURATION  
#define AMBER_DURATION RED_AND_AMBER_DURATION  
  
// ----- Private variables -----  
  
// The state of the system  
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; // Slave is Green; Master is Red  
}  
/*-----*  
-----*
```

```
TRAFFIC_LIGHTS_Update()
```

Must be called once per second.

```
- *-----*/  
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;
}
}

/*
----- END OF FILE -----
*/

```

Listing 16.2 Implementing a traffic-light control system using a single multi-state task

Further reading

Hybrid schedulers

Introduction

As we discussed in Chapter 13, co-operative schedulers provide a predictable platform for a wide range of embedded applications.

On some occasions, it can be necessary to incorporate some of the features of pre-emptive schedulers into a co-operative scheduler framework, in a carefully controlled manner. A hybrid scheduler seeks to achieve this.

HYBRID SCHEDULER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application is to have a time-triggered architecture, constructed using a scheduler.

Problem

How do you create and use a hybrid scheduler?

Background

As we have seen, co-operative, time-triggered architectures have many advantages when we wish to develop embedded applications; indeed, throughout this book we have argued that we would generally wish to use such an architecture when it is feasible to do so. However, we have also taken a realistic view of the limitations of the co-operative scheduler; in particular we have acknowledged that in some circumstances there can be a need to run both long tasks (e.g. 100 ms duration every 1,000 ms) and one or more short frequent task (e.g. 0.1 ms duration every 1 ms); these two requirements can conflict in a co-operative system, where – for all tasks, under all circumstances – the task duration, $Duration_{Task}$, must satisfy the condition:

$$Duration_{Task} < Tick\ Interval$$

In this pattern collection we have already considered various ways of meeting the need for both ‘frequent’ and ‘long’ tasks. For example, by using a faster processor (see Chapter 3) or a faster oscillator (Chapter 4) we can shorten the task durations. Alternatively, we can make use of timeouts (see Chapter 15) or develop one or more multi-stage tasks (see Chapter 16).

However, these solutions cannot always match our requirements and we will consider two further, very powerful, solutions in this book:

- In Part F, we will discuss the use of multiprocessor architectures. As we will see, using more than one microcontroller can provide a true multi-tasking capability.
- In this chapter, we will discuss the creation and use of a hybrid scheduler.

We briefly introduced hybrid schedulers in Chapter 13. This architecture combines the features of the pre-emptive and co-operative scheduler and allows (long) tasks to be pre-empted, as we require. However, it does this in a controlled manner, and without the need to resort to complex context-switching code, or to complex mechanisms for inter-task communication (see Figure 17.1)

The hybrid scheduler

- A hybrid scheduler provides limited **multi-tasking** capabilities

Operation:

- Supports any number of co-operatively scheduled tasks
- Supports a single pre-emptive task (which can interrupt the co-operative tasks)

Implementation:

- The scheduler is simple and can be implemented in a small amount of code
- The scheduler must allocate memory for two tasks at a time
- The scheduler will generally be written entirely in a high-level language (such as 'C')
- The scheduler is not a separate application; it becomes part of the developer's code

Performance:

- Rapid responses to external events can be obtained

Reliability and safety:

- With **careful design**, can be as reliable as a (pure) co-operative scheduler

FIGURE 17.1 Features of hybrid schedulers

Solution

WARNING!

When you have created a co-operative scheduler (see Chapter 14), modifying it to produce a hybrid scheduler involves only very minor code changes. We have seen many instances where designers have used hybrid designs 'because it was easy to do'. **This is almost always a mistake.**

In a hybrid scheduler the (purely) co-operative nature of the scheduling is lost. This can have **far-reaching implications** for both the design process and, ultimately, the system reliability.

Refer to 'Reliability and safety issues' before attempting to use this scheduler!

What we are trying to achieve

The form of hybrid scheduler described here differs from the (pure) co-operative scheduler as follows:

- The assumption that all tasks will complete between tick intervals is relaxed: one (or more) co-operative tasks may have a duration greater than the tick interval.
- Like the co-operative schedulers we have discussed any number of co-operative tasks may be scheduled; however, in addition, **one** pre-emptive task may also be scheduled.

- The pre-emptive task can pre-empt (interrupt) the co-operative tasks.
- The pre-emptive task runs to completion once executed: that is, the pre-emptive task cannot itself be interrupted by any of the co-operative tasks. The pre-emptive task is best viewed as ‘top priority’ task.

Also note the following:

- The fact that there is only one pre-emptive task, and that it runs to completion, greatly simplifies the system architecture compared to a fully pre-emptive solution. In particular, we do not need to implement a context switch mechanism. This means (a) that the architecture is still very simple, and (b) that the operating environment can still be implemented, entirely, in ‘C’.
- The fact that the pre-emptive task runs to completion also simplifies the inter-task communication compared with a fully pre-emptive environment: we provide further details later.
- Only a short task (with a maximum duration of around 50% of the tick interval – preferably much less) should be run pre-emptively, otherwise overall system performance will be impaired.

How we achieve it

To see how we can create a hybrid scheduler, we begin by reviewing the implementation of the co-operative scheduler.

Listing 17.1 shows the ‘update’ function from a co-operative scheduler.

```
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;

    TF2 = 0; // Have to manually clear this.

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].Task_p)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Incr. the run flag

                if (SCH_tasks_G[Index].Period)
                {
                    // Schedule regular tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
        }
    }
}
```

```
        }
    }
}
else
{
    // Not yet ready to run: just decrement the delay
    SCH_tasks_G[Index].Delay -= 1;
}
}
}
```

Listing 17.1 The ‘update’ function from a co-operative scheduler

The co-operative version assumes a scheduler data type of the type illustrated in Listing 17.2.

```
// Store in DATA area, if possible, for rapid access
// Total memory per task is 7 bytes
typedef data struct
{
    // Pointer to the task (must be a 'void (void)' function)
    void (code * Task_p)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tWord Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tWord Period;

    // Incremented (by scheduler) when task is due to execute
    tByte RunMe;
} sTask;
```

Listing 17.2 The user-defined data type used in the co-operative scheduler

Listing 17.3 shows the ‘update’ function for a hybrid scheduler.

```
void hSCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;

    TF2 = 0; // Have to manually clear this.

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < hSCH_MAX_TASKS; Index++)
```

```

{
// Check if there is a task at this location
if (hSCH_tasks_G[Index].pTask)
{
    if (hSCH_task_G[Index].Delay == 0)
    {
        // The task is due to run
        //
        if (hSCH_tasks_G[Index].Co_op)
        {
            // If it is a co-operative task, increment the RunMe flag
            hSCH_tasks_G[Index].RunMe += 1;
        }
    }
    else
    {
        // If it is a pre-emptive task, run it IMMEDIATELY
        (*hSCH_tasks_G[Index].pTask) (); // Run the task

        hSCH_task_G[Index].RunMe -= 1; // Reset / reduce
        RunMe flag

        // Periodic tasks will automatically run again
        // - if this is a 'one shot' task, remove it from the array
        if (hSCH_tasks_G[Index].Period == 0)
        {
            hSCH_tasks_G[Index].pTask = 0;
        }
    }
}

if (hSCH_tasks_G[Index].Period)
{
    // Schedule regular tasks to run again
    hSCH_tasks_G[Index].Delay = hSCH_tasks_G[Index].Period;
}
else
{
    // Not yet ready to run: just decrement the delay
    hSCH_tasks_G[Index].Delay -= 1;
}
}
}

```

Listing 17.3 The ‘update’ function from a hybrid scheduler

The hybrid version assumes a scheduler data type as shown in Listing 17.4.

```
// Store in DATA area, if possible, for rapid access
// Total memory per task is 8 bytes
typedef data struct
{
    // Pointer to the task (must be a 'void (void)' function)
    void (code * Task_p)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tWord Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tWord Period;

    // Incremented (by scheduler) when task is due to execute
    tByte RunMe;

    // Set to 1 if task is co-operative
    // Set to 0 if task is pre-emptive
    tByte Co_op;
} sTask;
```

Listing 17.4 The user-defined data type used in the hybrid scheduler

To set the Co_op field to an appropriate value an additional parameter is required in the SCH_Add_Task() function call (Figure 17.2).

Reliability and safety issues

Please read this section carefully before attempting to use this scheduler!

Problems with multi-tasking (pre-emptive) systems

In Chapter 13 we discussed the problems with pre-emptive schedulers, including the issues associated with critical sections of code. As we saw, to deal with critical sections of code in a fully pre-emptive system, we have two main possibilities:

- ‘Pause’ the scheduling by disabling the scheduler interrupt before beginning the critical section; re-enable the scheduler interrupt when we leave the critical section.
- Use a ‘lock’ (or some other form of ‘semaphore mechanism’) to achieve a similar result.

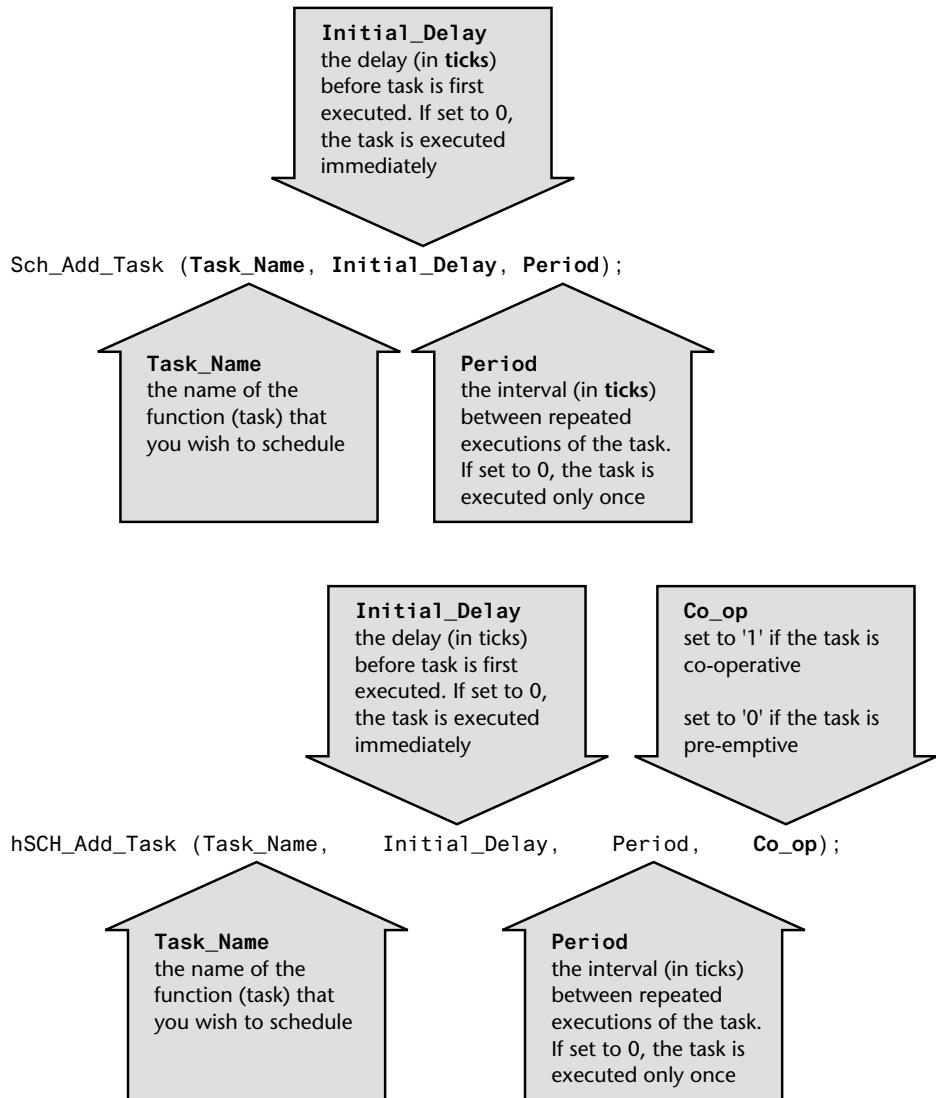


FIGURE 17.2 The parameters of the 'Add Task' function. [Top] The co-operative version. [Bottom] The hybrid version

In Chapter 13, we saw that both of these solutions had drawbacks. In particular, we saw that problems occur with the second solution if a task is interrupted after it reads the lock flag (and finds it unlocked) and before it sets the flag (to indicate that the resource is in use)(see Figure 17.3).

The problem does not occur in a hybrid scheduler, for the following reasons:

- In the case of pre-emptive tasks – because they cannot be interrupted – the 'interrupt between check and lock' situation cannot arise.

```

// ...
// Ready to enter critical section
// - Check lock is clear
if (Lock == LOCKED)
{
    return;
}

// Lock is clear
// Enter critical section

// Set the lock
Lock = LOCKED;

// CRITICAL CODE HERE //

```



**Problems arise if we have a context switch here
(between 'check' and 'set')**

FIGURE 17.3 The problems caused in a pre-emptive environment if a context switch occurs between 'check and 'set'

- In the case of co-operative tasks (which can be interrupted), the problem again cannot occur, for slightly different reasons.

Co-operative tasks can be interrupted 'between check and lock', but only by a pre-emptive task. If the pre-emptive task interrupts and finds that a critical section is unlocked, it will set the lock,³ use the resource, then clear the lock: that is, it will run to completion. The co-operative task will then resume and will **find the system in the same state that it was in before the pre-emptive task interrupted**: as a result, there can be no breach of the mutual exclusion rule.

Note that the hybrid scheduler solves the problem of access to critical sections of code in a simple way: unlike the complete pre-emptive scheduler, we do not require the creation of complex code 'lock' or 'semaphore' structures.

The safest way to use the hybrid scheduler

The most reliable way to use the hybrid scheduler is as follows:

- Create as many co-operative tasks as you require. It is likely that you will be using a hybrid scheduler because one or more of these tasks may have a duration greater than the tick interval; this can be done safely with a hybrid scheduler, but you **must** ensure that the tasks do not overlap.
- Implement **one** pre-emptive task; typically (but not necessarily) this will be called at every tick interval. A good use of this task is, for example, to check for errors or emergency conditions: this task can thereby be used to ensure that your system is able to respond within (say) 10 ms to an external event, even if its main purpose is to run (say) a 1,000 ms co-operative task.

3. Strictly, setting the lock flag is not necessary, as no interruption is possible.

- Remember that the pre-emptive task(s) can interrupt the co-operative tasks. If there are critical code sections, **you need to implement a simple lock mechanism** (the following example will illustrate this in detail).
- The pre-emptive task must be **short** (with a maximum duration of around 50% of the tick interval – preferably much less), otherwise overall system performance will be greatly impaired.
- Test the application carefully, under a full range of operating conditions, and monitor for errors.

Portability

The **HYBRID SCHEDULER** is as portable as the **CO-OPERATIVE SCHEDULER** [page 255]. It does not require the use of assembly code or other non-standard language features.

Overall strengths and weaknesses

The overall strengths and weaknesses of **HYBRID SCHEDULER** may be summarized as follows:

- ☺ Has the ability to deal with both ‘long infrequent tasks’ and a single ‘short frequent task’ that cannot be provided by a pure **CO-OPERATIVE SCHEDULER** [page 255].
- ☺ Is safe and predictable, if used according to the guidelines.
- ☹ It must be handled with caution.

Related patterns and alternative solutions

- See **CO-OPERATIVE SCHEDULER** [page 255]
- See **SCU SCHEDULER (LOCAL)** [page 609]
- See **scc SCHEDULER** [page 677]

Example: Hybrid scheduler using T2 (1 ms ticks)

A complete example of a hybrid scheduler is given in this section (Listings 17.5 to 17.12).

Note the use of a simple locking mechanism to ensure that only one task accesses the LED port at any particular time.

```
/*-----*-
Port.H (v1.00)
-----*
'Port Header' (see Chapter 10) for the project 2_01_12h
```

```

-----*/  

// ----- Sch51.C -----  

// Comment this line out if error reporting is NOT required  

// #define SCH_REPORT_ERRORS  

#ifndef SCH_REPORT_ERRORS  

// The port on which error codes will be displayed  

// ONLY USED IF ERRORS ARE REPORTED  

#define Error_port P1  

#endif  

// ----- LED_hyb.C -----  

// Two tasks share this port (P2)  

// We will treat it as a shared resource, and 'lock' it  

sbit LED_short_pin = P2^2;  

#define LED_long_port P2  

-----*/  

----- END OF FILE -----  

-----*/

```

Listing 17.5 Part of a hybrid scheduler example

```

/*-----*  

Main.c (v1.00)  

-----*  

Demonstration program for:  

/// HYBRID SCHEDULER ///  

Generic 16-bit auto-reload hybrid scheduler (using T2).  

Assumes 12 MHz oscillator (-> 01 ms tick interval).  

*** All timing is in TICKS (not milliseconds) ***  

Required linker options (see Chapter 14 for details):  

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)
{
    // Set up the scheduler
    hSCH_Init_T2();

    LED_Short_Init();

    // Add the 'short' task (on for ~1000 ms, off for ~1000 ms)
    // THIS IS A PRE-EMPTIVE TASK
    hSCH_Add_Task(LED_Short_Update, 0, 1000, 0);

    // Add the 'long' task (duration 10 seconds)
    // THIS IS A CO-OPERATIVE TASK
    hSCH_Add_Task(LED_Long_Update, 0, 20000, 1);

    // Start the scheduler
    hSCH_Start();

    while(1)
    {
        hSCH_Dispatch_Tasks();
    }
}

/*-----*
----- END OF FILE -----*
-----*/

```

Listing 17.6 Part of a hybrid scheduler example

```

/*-----*
----- 2_01_12h.h (v1.00)
-----*
----- - see 2_01_12h.C for details
-----*/
#include "Main.h"
#include "hSCH51.H"

// ----- Public function prototypes -----

```

```

void hSCH_Init_T2(void);
void hSCH_Start(void);

/* -----
--- END OF FILE
--- */

```

Listing 17.7 Part of a hybrid scheduler example

```

/* -----
2_01_12h.C (v1.00)

-----
*** THIS IS A *HYBRID* SCHEDULER FOR STANDARD 8051 / 8052 ***
*** Uses T2 for timing, 16-bit auto reload ***
*** 12 MHz oscillator -> 1 ms (precise) tick interval ***
--- */

#include "2_01_12h.h"

// ----- Public variable declarations -----
// The array of tasks (see Sch51.C)
extern sTaskH hSCH_tasks_G[hSCH_MAX_TASKS];

// The error code variable
//
// See Main.H for port on which error codes are displayed
// and for details of error codes
extern tByte Error_code_G;

/* -----
hSCH_Init_T2()

Scheduler initialization function. Prepares scheduler
data structures and sets up timer interrupts at required rate.

Must call this function before using the scheduler.
--- */

void hSCH_Init_T2(void)
{
    tByte i;

    for (i = 0; i < hSCH_MAX_TASKS; i++)
    {

```

```

        hSCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - hSCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // Now set up Timer 2
    // 16-bit timer function with automatic reload

    // Crystal is assumed to be 12 MHz
    // The Timer 2 resolution is 0.000001 seconds (1 µs)
    // The required Timer 2 overflow is 0.001 seconds (1 ms)
    // - this takes 1000 timer ticks
    // Reload value is 65536 - 1000 = 64536 (dec) = 0xFC18

    T2CON = 0x04;      // load Timer 2 control register
    T2MOD = 0x00;      // load Timer 2 mode register

    TH2     = 0xFC;    // load Timer 2 high byte
    RCAP2H = 0xFC;    // load Timer 2 reload capture reg, high byte
    TL2     = 0x18;    // load Timer 2 low byte
    RCAP2L = 0x18;    // load Timer 2 reload capture reg, low byte

    ET2 = 1; // Timer 2 interrupt is enabled

    TR2 = 1; // Start Timer 2
}

/*-----*/
hSCH_Start()
Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

/*-----*/
void hSCH_Start(void)
{
    EA = 1;
}

/*-----*/
hSCH_Update

```

This is the scheduler ISR. It is called at a rate determined by the timer settings in hSCH_Init(). This version is triggered by Timer 2 interrupts: timer is automatically reloaded.

```
-----*/
void hSCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;

    TF2 = 0; // Have to manually clear this.

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < hSCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (hSCH_tasks_G[Index].pTask)
        {
            if (hSCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                //
                if (hSCH_tasks_G[Index].Co_op)
                {
                    // If it is a co-operative task, increment the RunMe flag
                    hSCH_tasks_G[Index].RunMe += 1;
                }
            }
            else
            {
                // If it is a pre-emptive task, run it IMMEDIATELY
                (*hSCH_tasks_G[Index].pTask)();
                // Run the task

                hSCH_tasks_G[Index].RunMe -= 1; // Reset / reduce
                RunMe flag

                // Periodic tasks will automatically run again
                // - if this is a 'one shot' task, remove it from the
                array
                if (hSCH_tasks_G[Index].Period == 0)
                {
                    hSCH_tasks_G[Index].pTask = 0;
                }
            }
        }

        if (hSCH_tasks_G[Index].Period)
        {
            // Schedule regular tasks to run again
        }
    }
}
```

```

        hSCH_tasks_G[Index].Delay = hSCH_tasks_G[Index].Period;
    }
}
else
{
    // Not yet ready to run: just decrement the delay
    hSCH_tasks_G[Index].Delay -= 1;
}
}

/*
----- END OF FILE -----
*/

```

Listing 17.8 Part of a hybrid scheduler example

```

/*
----- *
hSCH51.h (v1.00)

-----
- see hSCH51.C for details
-*----- */

#ifndef _hSCH51_H
#define _hSCH51_H

#include "Main.h"

// ----- Public data type declarations -----
// Store in DATA area, if possible, for rapid access
// Total memory per task is 8 bytes
typedef data struct
{
    // Pointer to the task (must be a 'void (void)' function)
    void (code * pTask)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tWord Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tWord Period;
}

```

```

    // Incremented (by scheduler) when task is due to execute
    tByte RunMe;

    // Set to 1 if task is co-operative
    // Set to 0 if task is pre-emptive
    tByte Co_op;
} sTaskH;

// ----- Public function prototypes -----
// Core scheduler functions
void hSCH_Dispatch_Tasks(void);
tByte hSCH_Add_Task(void (code *) (void), tWord, tWord, bit);
bit hSCH_Delete_Task(tByte);
void hSCH_Report_Status(void);

// ----- Public constants -----
// The maximum number of tasks required at any one time
// during the execution of the program
//
// MUST BE ADJUSTED FOR EACH NEW PROJECT
#define hSCH_MAX_TASKS (2)

#endif

/*-----*
--- END OF FILE ---
-*-----*/

```

Listing 17.9 Part of a hybrid scheduler example

```

/*-----*
hSCH51.C (v1.00)
-----*

/// HYBRID SCHEDULER CORE ///

*** THESE ARE THE CORE SCHEDULER FUNCTIONS ***
--- These functions may be used with all 8051 devices ---

*** hSCH_MAX_TASKS *must* be set by the user ***
--- see "Sch51.h" ---

*** Includes power-saving mode ***
--- You *MUST* confirm that the power-down mode is adapted ---
--- to match your chosen device (usually only necessary with
--- Extended 8051s, such as c515c, c509, etc ---
```

```

-----*/  

#include "Main.h"  

#include "Port.h"  

#include "hSch51.h"  

// ----- Public variable definitions -----  

// The array of tasks  

sTaskH hSCH_tasks_G[hSCH_MAX_TASKS];  

// Used to display the error code  

// See Main.H for details of error codes  

// See 'Private Constants' (below) for details of the error port  

// - refer to book for further details (Chapter 14)  

tByte Error_code_G = 0;  

// ----- Private function prototypes -----  

static void hSCH_Go_To_Sleep(void);  

// ----- Private variables -----  

// Keeps track of time since last error was recorded (see below)  

static tWord Error_tick_count_G;  

// The code of the last error (reset after ~1 minute)  

static tByte Last_error_code_G;  

/*-----*/  

hSCH_Dispatch_Tasks()  

This is the 'dispatcher' function. When a task (function)  

is due to run, hSCH_Dispatch_Tasks() will run it.  

This function must be called (repeatedly) from the main loop.  

/*-----*/  

void hSCH_Dispatch_Tasks(void)  

{
    tByte Index;  

    // Dispatches (runs) the next task (if one is ready)
    for (Index = 0; Index < hSCH_MAX_TASKS; Index++)
    {
        // Only dispatching co-operative tasks
        if ((hSCH_tasks_G[Index].Co_op) && (hSCH_tasks_G[Index].RunMe > 0))
        {
            (*hSCH_tasks_G[Index].pTask)(); // Run the task
        }
    }
}

```

```

        hSCH_tasks_G[Index].RunMe -= 1;    // Reset / reduce RunMe flag

        // Periodic tasks will automatically run again
        // - if this is a 'one shot' task, remove it from the array
        if (hSCH_tasks_G[Index].Period == 0)
        {
            // Faster than call to delete task
            hSCH_tasks_G[Index].pTask = 0;
        }
    }

    // Report system status
    hSCH_Report_Status();

    // The scheduler enters idle mode at this point
    hSCH_Go_To_Sleep();
}

/*-----*/
hSCH_Add_Task()

Causes a task (function) to be executed at regular intervals
or after a user-defined delay

Fn_P - The name of the function which is to be scheduled.
       NOTE: All scheduled functions must be 'void, void' -
              that is, they must take no parameters, and have
              a void return type.

Del - The interval (TICKS) before the task is first executed

Per - If 'Per' is 0, the function is only called once,
      at the time determined by 'Del'. If Per is non-zero,
      then the function is called repeatedly at an interval
      determined by the value of Per (see below for examples
      that should help clarify this).

Co-op - Set to 1 if it a co-op task; 0 if pre-emptive

RETN: The position in the task array at which the task has been added.
      If the return value is hSCH_MAX_TASKS then the task could not be
      added to the array (there was insufficient space). If the
      return value is < hSCH_MAX_TASKS, then the task was added
      successfully.

Note: this return value may be required, if a task is
      to be subsequently deleted - see hSCH_Delete_Task().

```

EXAMPLES:

```

Task_ID = hSCH_Add_Task(Do_X,1000,0,0);
Causes the function Do_X() to be executed once after 1000 ticks.
(Pre-emptive task)

Task_ID = hSCH_Add_Task(Do_X,0,1000,1);
Causes the function Do_X() to be executed regularly, every 1000 ticks.
(co-operative task)

Task_ID = hSCH_Add_Task(Do_X,300,1000,0);
Causes the function Do_X() to be executed regularly, every 1000 ticks.
Task will be first executed at T = 300 ticks, then 1300, 2300, etc.
(Pre-emptive task)

*-----*/
tByte hSCH_Add_Task(void (code* Fn_p)(), // Task function pointer
                     tWord   Del,    // Num ticks 'til task first runs
                     tWord   Per,    // Num ticks between repeat runs
                     bit     Co_op) // Co_op / pre_emp
{
    tByte Index = 0;

    // First find a gap in the array (if there is one)
    while ((hSCH_tasks_G[Index].pTask != 0) && (Index < hSCH_MAX_TASKS))
    {
        Index++;
    }

    // Have we reached the end of the list?
    if (Index == hSCH_MAX_TASKS)
    {
        // Task list is full
        //

        // Set the global error variable
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;

        // Also return an error code
        return hSCH_MAX_TASKS;
    }

    // If we're here, there is a space in the task array
    hSCH_tasks_G[Index].pTask = Fn_p;

    hSCH_tasks_G[Index].Delay = Del;
    hSCH_tasks_G[Index].Period = Per;

    hSCH_tasks_G[Index].Co_op = Co_op;
}

```

```

    hSCH_tasks_G[Index].RunMe = 0;

    return Index; // return position of task (to allow later deletion)
}

/*-----*
hSCH_Delete_Task()

Removes a task from the scheduler. Note that this does
*not* delete the associated function from memory:
it simply means that it is no longer called by the scheduler.

PARAMS: Task_index - The task index. Provided by hSCH_Add_Task().

RETURNS: RETURN_ERROR or RETURN_NORMAL

-*-----*/
bit hSCH_Delete_Task(tByte Task_index)
{
    bit Return_code;

    if (hSCH_tasks_G[Task_index].pTask == 0)
    {
        // No task at this location...
        //
        // Set the global error variable
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;

        // ...also return an error code
        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; // return status
}
/*-----*
hSCH_Report_Status()

Simple function to display error codes.

```

This version displays code on a port with attached LEDs:
adapt, if required, to report errors over serial link, etc.

Errors are only displayed for a limited period
(60000 ticks = 1 minute at 1 ms tick interval).

After this the error code is reset to 0.

This code may be easily adapted to display the last
error 'for ever': this may be appropriate in your
application.

See Chapter 14 for further information.

```
-----*/  
void hSCH_Report_Status(void)  
{  
#ifdef SCH_REPORT_ERRORS  
    // ONLY APPLIES IF WE ARE REPORTING ERRORS  
    // Check for a new error code  
    if (Error_code_G != Last_error_code_G)  
    {  
        // Negative logic on LEDs assumed  
        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; // Reset error code  
            }  
        }  
    }  
}
```

```

#endif
}

/*-----*/
hSCH_Go_To_Sleep()

This scheduler enters 'idle mode' between clock ticks
to save power. The next clock tick will return the processor
to the normal operating state.

Note: a slight performance improvement is possible if this
function is implemented as a macro, or if the code here is simply
pasted into the 'dispatch' function.

However, by making this a function call, it becomes easier
- during development - to assess the performance of the
scheduler, using the 'performance analyser' in the Keil
hardware simulator. See Chapter 14 for examples for this.

*** May wish to disable this if using a watchdog ***

*** ADAPT AS REQUIRED FOR YOUR HARDWARE ***

/*-----*/
void hSCH_Go_To_Sleep()
{
    PCON |= 0x01;      // Enter idle mode (generic 8051 version)

    // Entering idle mode requires TWO consecutive instructions
    // on 80c515 / 80c505 - to avoid accidental triggering
    // PCON |= 0x01;      // Enter idle mode (#1)
    // PCON |= 0x20;      // Enter idle mode (#2)
}
/*-----*/
---- END OF FILE
/*-----*/

```

Listing 17.10 Part of a hybrid scheduler example

```

/*-----*/
LED_Hyb.C (v1.00)

-----
- See LED_Hyb.C for details.
/*-----*/

```

```
// ----- Pic function prototypes -----
void LED_Short_Init(void);
void LED_Short_Update(void);
void LED_Long_Update(void);

/*-----*
--- END OF FILE -----
*-----*/
```

Listing 17.11 Part of a hybrid scheduler example

```
/*-----*
LED_Hyb.C (v1.0)

-----
Simple 'Flash LED' test function for HYBRID scheduler.

*** With lock mechanism ***

*-----*/
#include "Main.h"
#include "Port.H"

#include "LED_Hyb.h"
#include "Delay_T0.h"

// ----- Private constants -----
// For the lock mechanism
#define LOCKED 1
#define UNLOCKED 0

// ----- Private variable definitions -----
static bit LED_short_state_G;
// The lock flag
static bit LED_lock_G = UNLOCKED;

/*-----*
LED_Flash_Init()
- Prepare to flash LEDs.

*-----*/
void LED_Short_Init(void)
{
```

```
LED_short_state_G = 0;
LED_long_port = 0xAA;
LED_lock_G = UNLOCKED;
}

/*-----*/
LED_Short_Update()
Flashes an LED (or pulses a buzzer, etc) on a specified port pin.
Must schedule at twice the required flash rate: thus, for 1 Hz
flash (on for 0.5 seconds, off for 0.5 seconds) must schedule
at 2 Hz.

/*-----*/
void LED_Short_Update(void)
{
// The port has a lock
// If it is locked, we simply return
if (LED_lock_G == LOCKED)
{
return;
}

// Port is free - lock it
LED_lock_G = LOCKED;

// Change the LED from OFF to ON (or vice versa)
if (LED_short_state_G == 1)
{
LED_short_state_G = 0;
LED_short_pin = 0;
}
else
{
LED_short_state_G = 1;
LED_short_pin = 1;
}

// Unlock the port
LED_lock_G = UNLOCKED;
}

/*-----*/
LED_Long_Update()
```

```
Demo 'long' task (10 second duration).  
-----*/  
void LED_Long_Update(void)  
{  
    tByte i;  
  
    // The port has a lock  
    // If it is locked, we simply return  
    if (LED_lock_G == LOCKED)  
    {  
        return;  
    }  
  
    // Port is free - lock it  
    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);  
    }  
    // Unlock the port  
    LED_lock_G = UNLOCKED;  
}  
/*-----*  
--- END OF FILE -----  
-----*/
```

Listing 17.12 Part of a hybrid scheduler example

Example: Further hybrid schedulers

Please see Chapter 34 for examples of the use of hybrid schedulers in speech-processing applications.

Further reading

Please refer to page 253 for a number of suitable suggestions for further reading in this area.

The user interface

In this part, we will consider some key patterns suitable for assisting in the creation of user interfaces for a wide range of embedded applications.

In Chapter 18, we present **PC LINK (RS-232)** [page 364]. This pattern considers how, using the universal 'RS-232' standard, we can transfer data between an embedded device and a desktop, notebook or similar PC. In addition to being a useful pattern in its own right, this pattern illustrates many of the features required in software developed for a co-operatively scheduled environment.

In Chapter 19, we will explore techniques for reading inputs from switches: here we will be concerned with both software-only and hardware-based interfaces. In Chapter 20, we will see how the same basic techniques may be extended to work efficiently with keypads.

In Chapter 21, we turn our attention to the creation of LED displays. In particular, we will be concerned with multiplexed LED interfaces, which are the only cost-effective solution in most applications.

Finally, in Chapter 22, we will consider how to control LCD (text) displays. In this chapter we will be specifically concerned with the interaction with displays based on the ubiquitous Hitachi HD44780 LCD controller chip.

chapter 18

Communicating with PCs via RS-232

Introduction

In this chapter we present **PC LINK (RS-232)**. This pattern describes how, using the universal 'RS-232' standard, we can transfer data between an embedded device and a desktop, notebook or similar PC.

Note: We are **not** concerned here with the process of transferring data between two (or more) microcontrollers: this is discussed in Part F.

PC LINK (RS-232)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you link your embedded application to a desktop (or notebook or hand-held) PC using 'RS-232'?

Background

This pattern is concerned with the use of what is commonly referred to as the RS-232 communication protocol, to transfer data between an 8051 microcontroller and some form of personal computer (PC, notebook or similar).

We provide some important background information on RS-232 in this section. We begin, however, by considering some important terminology used in the communications area.

What are 'simplex', 'half-duplex' and 'duplex' serial communications?

In a **simplex** serial communication system, we typically require at least two wires ('signal' and 'ground'). Data transfer is only in one direction: for example, we might transfer data from a simple data-monitoring system to a PC using simplex data transfer.

In a **half-duplex** serial communication system we again (typically) require at least two wires. Here, data transfer can be in both directions. However, transmission can only be in one direction at a time. In a variation of the data-monitoring example, we might use a half-duplex communication strategy to allow us to send information to the embedded module (to set parameters, such as sampling rate). We might also, at other times, use the same link to download the data from the embedded board to a PC.

In a **full-duplex** serial communication system, we typically require at least three wires (Signal A, Signal B, Ground). The line Signal A will carry data in one direction **at the same time that** Signal B carries data in the other direction.

What is 'RS-232'?

In 1997 the Telecommunications Industry Association released what is formally known as TIA-232 Version F, a serial communication protocol which has been

universally referred to as 'RS-232' since its first 'Recommended Standard' appeared in the 1960s. Similar standards (V.28) are published by the International Telecommunications Union (ITU) and by CCITT (Consultative Committee International Telegraph and Telephone).

The 'RS-232' standard includes details of:

- The protocol to be used for data transmission.
- The voltages to be used on the signal lines.
- The connectors to be used to link equipment together.

Overall, the standard is comprehensive and widely used, at data transfer rates of up to around 115 or 330 kbytes / second (115 / 330 k baud). Data transfer can be over distances of 15 metres or more.

Note that RS-232 is a peer-to-peer communication standard. Unlike the multi-drop RS-485 standard (which we consider in Chapter 27), RS-232 is intended to link only two devices together.

Basic RS-232 protocol

RS-232 is a character-oriented protocol. That is, it is intended to be used to send single 8-bit blocks of data. To transmit a byte of data over an RS-232 link, we generally encode the information as follows:

- We send a 'Start' bit.
- We send the data (8 bits).
- We send a 'Stop' bit (or bits).

We consider each of these stages here.

Quiescent state

When no data are being sent on an RS-232 'transmit' line, the line is held at a Logic 1 level.

Start bit

To indicate the start of a data transmission we pull the 'transmit' line low.

Data

Data are often encoded in ASCII (American Standard Code for Information Interchange), in 7-bit form. The bits are sent least significant bit first. If we are sending 7-bit data, the eighth data bit is often used as a simple parity check bit and is transmitted in order to provide a rudimentary error detection facility on a character-by-character basis.

Note that none of the code presented here uses parity bits: we use all 8 bits for data transfer.

Stop bit(s)

The stop bits consist of a Logic 1 output. These can be 1 or, less commonly, 1½ or 2 pulses wide.

Note that we will use a single stop bit in all code example.

Asynchronous data transmission and baud rates

RS-232 uses an asynchronous protocol. This means that no clock signal is sent with the data. Instead, both ends of the communication link have an internal clock, running at the same rate. The data (in the case of RS-232, the 'Start' bit) is then used to synchronize the clocks, if necessary, to ensure successful data transfer.

RS-232 generally operates at one of a (restricted) range of baud rates. Typically these are: 75, 110, 300, 1,200, 2,400, 4,800, 9,600, 14,400, 19,200, 28,800, 33,600, 56,000, 115,000 and (rarely) 330,000 baud. Of these, 9,600 baud is a very 'safe' choice, as it is very widely supported.

RS-232 voltage levels

The threshold levels used by the receiver are +3V and -3V and the lines are inverted. The maximum voltage allowed is +/- 15V.

Note that these voltages cannot be obtained directly from the naked microcontroller port pins: some form of interface hardware is required. For example, the Maxim Max232 and Max233 are popular and widely used line driver chips; we consider how to use such chips in 'Solution'.

Flow control

RS-232 is often used with some form of flow control. This is a protocol, implemented through software or hardware, that allows a receiver of data to tell the transmitter to pause the dataflow. This might be necessary, for example, if we were sending data to a PC and the PC had filled a RAM buffer: the PC would then tell our embedded application to pause the data transfer until the buffer contents had been stored on disk.

Although hardware handshaking can be used, this requires extra signal lines. The most common flow control technique is 'Xon / Xoff' control. This requires a half- or full-duplex communication link, and can operate as follows:

- 1** Transmitter sends a byte of data.
- 2** The receiver is able to receive more data: it does nothing.
- 3** The transmitter sends another byte of data.
- 4** Steps 1–3 continue until the receiver cannot accept any more data: it then sends a 'Control s' (Xoff) character back to the transmitter.
- 5** The transmitter receives the 'Xoff' command and pauses the data transmission.

- 6 When the receiver node is ready for more data, it sends a ‘Control q’ (Xon) character to the transmitter.
- 7 The transmitter resumes the data transmission.
- 8 The process continues from Step 1.

Solution

In this section we consider how to implement an RS-232 link from an embedded 8051 microcontroller to a PC.

Transceiver hardware

As noted in ‘Background’, the voltages used in RS-232 communications are incompatible with the voltages used on the microcontroller itself. Therefore, you will require some form of voltage level conversion circuitry between the microcontroller board and the PC cable.

Usually the most cost-effective way of achieving this is to use a ‘transceiver’ (transmitter-receiver) chip created for this purpose. Of these, the Max232 (from Maxim) is frequently used; however, in new designs the more recent Max233 is a better choice as it requires fewer external components.

The required link to a Max233 is illustrated in Figure 18.1, which also shows the standard connections (9-pin D-type socket: ‘DB-9’) used for most recent designs.

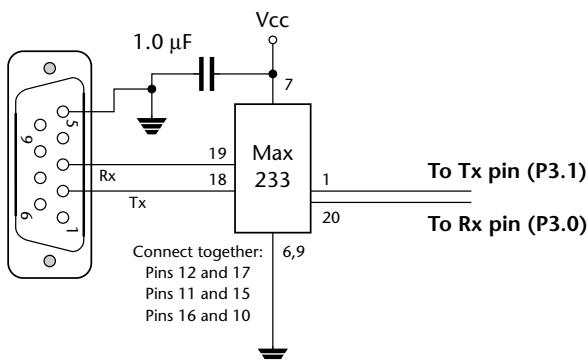


FIGURE 18.1 Using a Max233 as an RS-232 transceiver

Cable connections

To connect to the PC using the software presented here, you need a 3-wire cable.

Use DB-9 socket (‘female’) connector at the PC end and DB-9 plug (‘male’) connector at the microcontroller end.

The required cable connections are a ‘straight through’ cable as described in Table 18.1.

TABLE 18.1 Connections required for a ‘straight through’ cable needed to link the microcontroller to the desktop PC (or similar terminal)

PC (COM1, COM2) DB-9 connector		Microcontroller DB-9 connector
RxD – Pin 2	to	TxD – Pin 2
TxD – Pin 3	to	RxD – Pin 3
Ground – Pin 5	to	Ground – Pin 5

The software architecture

Suppose we wish to transfer data to a PC at a standard 9,600 baud rate; that is, 9,600 bits per second. As we discussed in ‘Background’, transmitting each byte of data, plus stop and start bits, involves the transmission of ten bits of information (assuming a single stop bit is used). As a result, each byte takes approximately 1 ms to transmit.

This has important implications in all applications, not least those using a scheduler. If, for example, we wish to send this information to the PC:

Current core temperature is 36.678 degrees

then the task sending these 42 characters will take more than 40 milliseconds to complete. This will frequently be an unacceptably long duration.

The most obvious way of solving this problem is to increase the baud rate; however, this is not always possible and it does not solve the underlying problem.

A better solution is to write all data to a buffer in the microcontroller. The contents of this buffer will then be sent – usually one byte at a time – to the PC, using a regular, scheduled task.

This solution is used in the code libraries presented in the following sections and included on the accompanying CD. The architecture is a good example of a **MULTI-STAGE TASK** [page 317], and is frequently used in user-interface libraries: **LCD CHARACTER PANEL** [page 467], for example, uses a very similar architecture.

Using the on-chip U(S)ART for RS-232 communications

Having decided on the basic architecture for the RS-232 library, we need to consider in more detail how the on-chip serial port is used.

This port is full duplex, meaning it can transmit and receive simultaneously. It is also receive buffered, meaning it can commence reception of a second byte before a previously received byte has been read from the receive register. (However, if the first byte still has not been read by the time reception of the second byte is complete, one of the bytes will be lost.)

The serial port can operate in four modes (one synchronous mode, three asynchronous modes). In this pattern, we are primarily interested in Mode 1. In this mode, 10 bits are transmitted (through TxD) or received (through RxD): a start bit (0), 8 data bits (lsb first), and a stop bit (1).

Note that the serial interface may also provide interrupt requests when transmission or reception of a byte has been completed. However, for reasons discussed in Chapter 1, none of the code used in this pattern will generate interrupts.

Serial port registers

The serial port control and status register is the special function register SCON. This register contains the mode selection bits (and the serial port interrupt bits, TI and RI).

SBUF is the receive and transmit buffer of serial interface. Writing to SBUF loads the transmit register and initiates transmission. Reading out SBUF accesses a physically separate receive register.

Baud rate generation

There are several different ways to generate the baud rate clock for the serial port depending on the mode in which it is operating.

As already noted, we are primarily concerned here with the use of the serial port in Mode 1. In this mode the baud rate is determined by the overflow rate of Timer 1 or Timer 2. For reasons discussed in Chapter 14, we assume that, if Timer 2 is available, it will usually be used to drive the scheduler. Therefore, we focus on the use of Timer 1 for baud rate generation.

The baud rate is determined by the Timer 1 overflow rate and the value of SMOD follows:

$$\text{Baud rate (Mode 1)} = \frac{2^{\text{SMOD}} \times \text{Frequency}_{\text{oscillator}}}{32 \times \text{Instructions}_{\text{cycle}} \times (256 - TH1)}$$

where:

SMOD is the ‘double baud rate’ bit in the PCON register

Frequency_{oscillator} is the oscillator / resonator frequency

Instructions_{cycle} is the number of machine instructions per oscillator cycle (e.g. 12 or 6)

TH1 is the reload value for Timer 1

Note that Timer 1 is used in 8-bit auto-reload mode and that interrupt generation should be disabled.

It is very important to appreciate that it is not generally possible to produce standard baud rates (e.g. 9600) using Timer 1, unless you use an 11.0592 MHz crystal oscillator.

To see why this is so, we will assume SMOD = 0 (it works equally well with SMOD = 1), that there are 12 instructions per cycle and that we require a baud rate of 9,600. The equation now becomes:

$$9600 = \frac{11059200}{32 \times 12 \times (256 - TH1)}$$

This becomes:

$$\frac{11059200}{9600 \times 384} = 256 - TH1$$

Or:

$$256 - TH1 = 3$$

Thus, if we set TH1 to 253 (0xFD) we get a *precise* 9,600 baud rate. If, for example, we repeat this with a 12 MHz oscillator, we get:

$$\frac{12000000}{9600 \times 384} = 256 - TH1$$

Or:

$$256 - TH1 = 3.255208333333$$

Thus, the required value of TH1 is 252.7447916667.

The nearest integer value is 253. This means that our actual baud rate will be approximately 10,417 baud – more than 8% higher than the required (9,600) rate.

Remember: This is an asynchronous protocol and **relies for correct operation on the fact that both ends of the connection are working at the same baud rate**. In practice, you can generally work with a difference in baud rates at both ends of the connection **by up to 5%**, but no more.

Despite the possible 5% margin, it is always good policy to get the baud rate as close as possible to the standard value because, **in the field**, there may be significant temperature variations between the oscillator in the PC and that in the embedded system. This will lead to a ‘drift’ in baud rates on PC and microcontroller, even if they were precisely the same to start with: as a result, if the baud rates were initially mismatched, then communication with the PC may fail completely during normal use. This type of ‘inexplicable fault’ has caused many developers sleepless nights (‘I don’t understand it! It works fine in all the tests in the lab!’).

Note also that it is generally **essential** to use some form of crystal oscillator (rather than a ceramic resonator) when working with asynchronous serial links (such as RS-232, RS-485 or CAN): the ceramic resonator is not sufficiently stable for this purpose: see **CRYSTAL OSCILLATOR** [page 54] and **CERAMIC RESONATOR** [page 64] for details.

Dealing with the 11.0592 MHz problem

In a scheduled application, using an 11.0592 MHz crystal oscillator is not ideal. 11.0592 MHz may translate into precise baud rates, but it is not ideal as a source of ‘ticks’ in the scheduler: in particular, it is impossible to produce precise 1 ms ticks from an 8051 device driven by an 11.0592 MHz crystal.

There are at least four solutions to this problem:

- The first solution is to use a 11.0592 MHz crystal and work with a 5 ms tick interval. Note that a 5 ms interval can be generated precisely at this crystal frequency, as demonstrated in the example on the CD.
- The second solution is to use Timer 2 to generate the baud rates. Timer 2 allows precise baud-rate generation at a wider range of oscillator frequencies than Timer 1; of course – in a single-processor system – this means that Timer 2 cannot simultaneously be used to generate the scheduler ticks, and Timer 0 or Timer 1 will need to be used for this purpose.
- The third solution is to use the dedicated baud rate generator included on recent 8051 devices. These allow very standard baud rates to be generated even with – say – 12 MHz crystals. In addition, use of a dedicated generator frees up the other timers for general-purpose use. Again, one of the libraries on the CD make use of internal baud rate generator.
- The fourth solution is to use two-microcontroller solution, and a shared-clock scheduler. Figure 18.2 illustrates this approach.

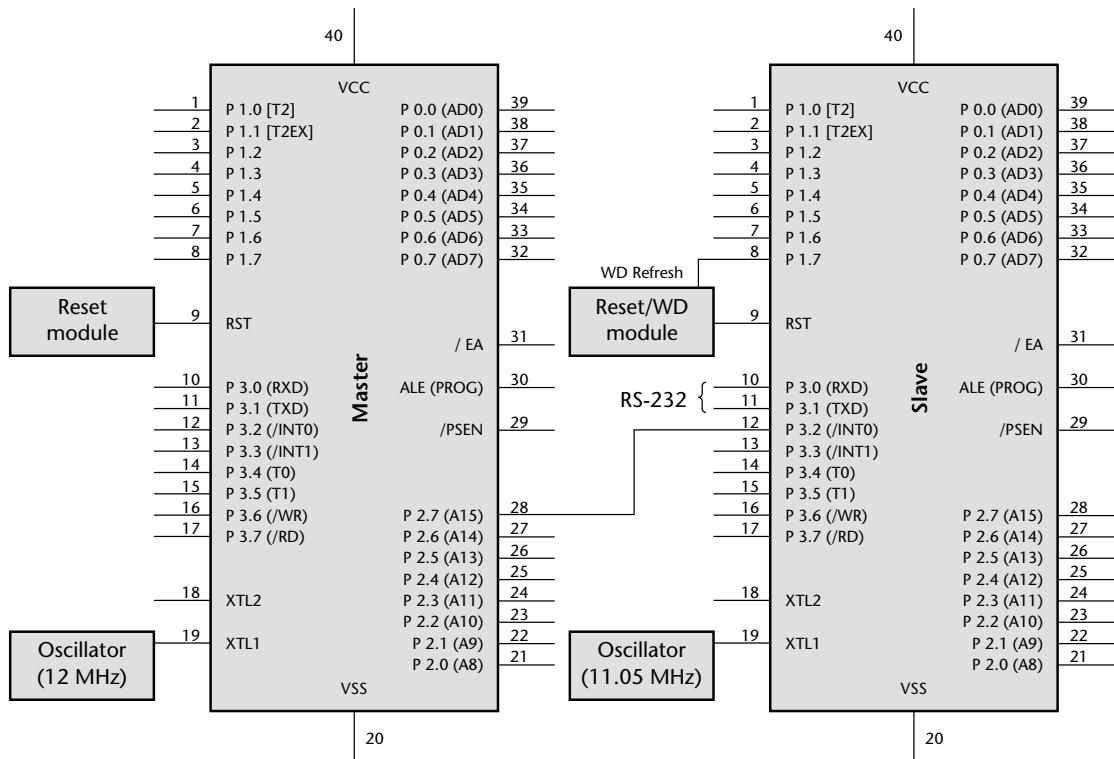


FIGURE 18.2 Using two processors and a shared-clock (interrupt) scheduler to provide precise 1 ms tick intervals (via the 12 MHz crystal on the Master), and precise baud rates (via the 11.0592 MHz crystal on the Slave).

PC software

We focus in this pattern on the software required on the microcontroller in order to transfer data to a PC. Clearly, however, we also need software on the PC itself.

If your desktop computer is running Windows (95, 98, NT, 2000), then a simple but effective option is the 'Hyperterminal' application which is included with all of these operating systems. Figure 18.3 shows this application running one of the example applications.

While Hyperterminal (or similar terminal-emulator programs in other environments) are useful, they are not suitable for all purposes.

If you need more specialized PC code for storing or analyzing data from your embedded application, you will probably need to write your own. Such programs lie beyond the scope of this book but Axelson (1998) is a good source of further information and useful code samples.

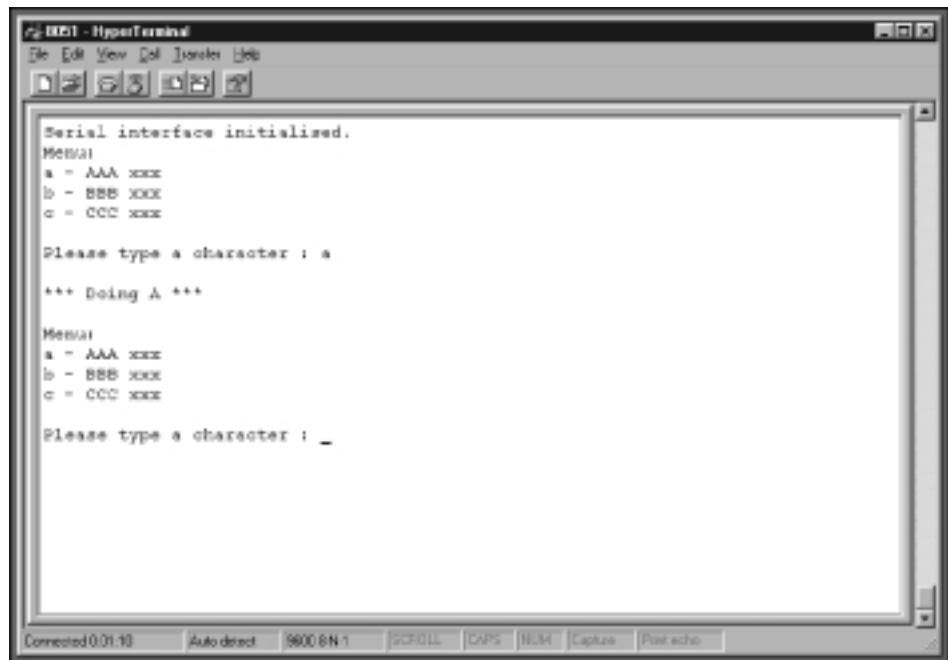


FIGURE 18.3 Using 'Hyperterminal' to display the information from an embedded microcontroller

Hardware resource implications

The use of the PC link library described in this pattern has the following hardware resource implications:

- It uses the UART.
- It uses two port pins (P3.0, P3.1).
- It uses either a general-purpose timer or a specialized baud rate generator circuit.

Overall, however, it is the memory load that causes greatest difficulty, particularly in situations where only internal memory is available. The root of this difficulty is that fact that, as we discussed in ‘Solution’, the architecture used (to transmit data to the PC) is based on the use of a buffer to which the user writes as required; this buffer is then emptied, one character at a time, by a scheduled task.¹

The buffer itself is very simple:

```
// The transmit buffer length
#define PC_LINK_TRAN_BUFFER_LENGTH 100
static tByte Tran_buffer[PC_LINK_TRAN_BUFFER_LENGTH];
```

If you run out of memory, there are several options:

- You can reduce the buffer size. This may mean that tasks in your applications must break down the data they send into smaller blocks. Typically, the main implication is that shorter strings must be used.
- You can increase the baud rate and adapt the code so that the scheduler sends more than one byte of data in every time the ‘update’ function is run.
- If using on-chip memory only, you can choose an 8051 device with additional on-chip RAM: for example, Dallas and Infineon produce a number of such devices. We discuss how to make use of this memory in **OFF-CHIP DATA MEMORY** [page 94].

Reliability and safety issues

We consider the reliability and safety implications of RS-232 communications in this section.

What about `printf()`?

We do not generally recommend the use of standard library functions, such as `printf()`, because:

- This function sends data immediately to the UART. As a result, the duration of the transmission is usually too long to be safely handled in a co-operatively scheduled application.
- Most implementations of `printf()` do not incorporate timeouts, making it possible that use of this functions can ‘hang’ the whole application if errors occur.

1. Note that while both the ‘transmit’ and (where available) ‘receive’ channels have associated buffers, the transmit buffer is usually larger than the receive buffer, since – in most cases – the direction of dataflow is from the microcontroller to the PC. As a result, we focus on the transmit buffer here. However, similar concerns – and solutions – apply to the receive buffer in circumstances where the main direction of information flow is from the PC to the microcontroller.

If you are determined to use `printf()`, refer to Chapter 10 where a simple library using this function is presented.

General comments

The RS-232 protocol does not provide any adequate form of error checking: you need to carry this out manually. A simple approach is to send important data to the PC two (or more) times and compare the two (or more) versions: if there is a mismatch, have the data sent again. Clearly, this approach greatly increases the required communication bandwidth.

Portability

The UART is part of the 8051 core: therefore if your application has a UART-based RS-232 interface it will generally be portable. However, use of non-core features (like dedicated baud rate generators or more than one serial port) will dramatically reduce the porting opportunities.

Overall strengths and weaknesses

- ☺ RS-232 support is part of the 8051 core: applications based on RS-232 are very portable.
- ☺ At the PC end too, RS-232 is ubiquitous: every PC has one or more RS-232 ports.
- ☺ Links can – with modern transceiver chips – be up to 30 m (100 ft) in length.
- ☺ Because of the hardware support, RS-232 generally imposes a low software load.
- ☹ RS-232 is a peer-to-peer protocol (unlike, for example, RS-485: see Chapter 27): you can only connect one microcontroller directly (simultaneously) to each PC.
- ☹ RS-232 has little or no error checking at the hardware level (unlike, for example, CAN: see Chapter 28): if you want to be sure that the data you received at the PC are valid, you need to carry out checks in software.

Related patterns and alternative solutions

There are several related patterns and some alternative solutions. We discuss these here.

Selecting an oscillator or resonator

As already noted, it is generally essential to use some form of crystal oscillator (rather than a ceramic resonator) when working with asynchronous serial links: the ceramic resonator is not sufficiently stable for this purpose: see **CRYSTAL OSCILLATOR** [page 54] and **CERAMIC RESONATOR** [page 64] for details.

Multi-drop communications

If you wish to connect more than one microcontroller to a single PC, this is readily possible. Use a shared-clock scheduler (see **SCI SCHEDULER (DATA)** [page 593] and **SCC SCHEDULER** [page 677]) to link the controllers together, and use a serial port on one of the boards to send the data to the PC, over an RS-232 link. If required, you can send data to multiple PCs in this way.

USB

RS-232 has been around since the 1960s. It is popular and effective, but has its limitations. Increasingly, PCs are being released with (additional) USB (universal serial bus) ports.

It is now becoming possible to use USB to transfer data between PCs and 8051 microcontrollers. For example, the Infineon C541 is an 8051 device with on-chip USB support. Note that this chip includes not just the USB controller, but also the driver as well: unlike RS-232, no external transceiver hardware is required (Figure 18.4).

Other USB-compatible 8051 devices are the EZ-USB range from Cypress Semiconductor,² and the TUSB3200 from Texas Instruments.³

Note that for 8051 microcontrollers without USB support, an alternative you may wish to consider is the FT8U232AM USB UART chip from FTDI.⁴ As far as the 8051 is concerned, the USB UART appears to be an ordinary RS-232 transceiver.

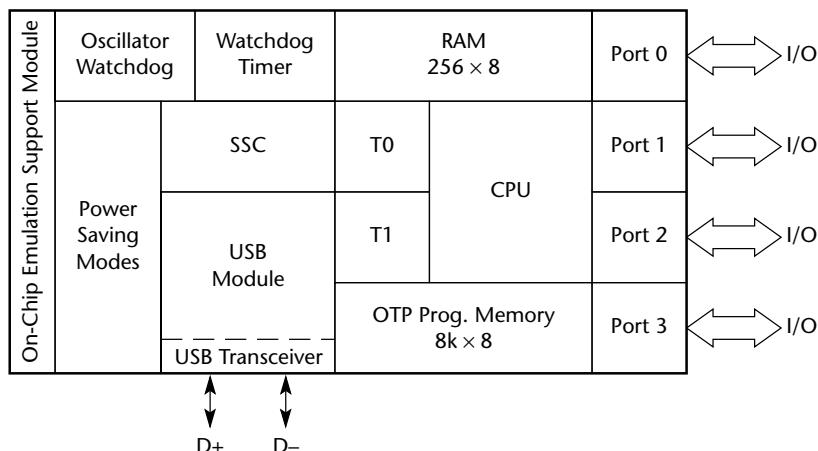


FIGURE 18.4 The organization of the Infineon C541 microcontroller, which has on-chip USB support, (Reproduced courtesy of Infineon).

2. www.cypress.com

3. www.ti.com

4. www.ftdi.co.uk

Example: A PC Link library for the 8051/8052 (generic)

This example illustrates how to link a Standard 8051 device to a PC (listings 18.1 to 18.5). The software is menu driven in this example: the user is (via a terminal emulator on the PC) given the option of running one of three different tasks on the microcontroller (see Figure 18.5).

In this library, the scheduler is driven by Timer 2, and Timer 1 is used for baud rate generation.

As usual, all files for this example are included on the CD, in the directory associated with this chapter.

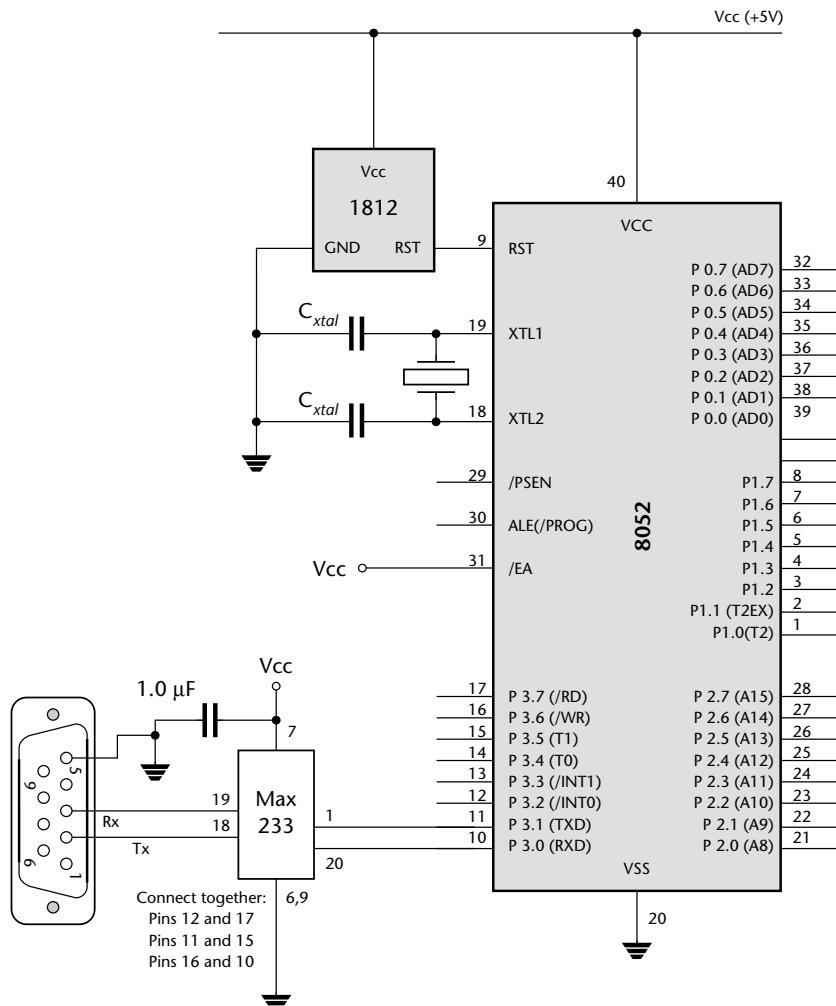


FIGURE 18.5 Connecting an 8051 microcontroller to a Max233 (RS-232) transceiver.

```

/*-----*-
Port.H (v1.00)

-----*
'Port Header' (see Chapter 10) for the project IO_T2_T1

// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif

// ----- Lnk_IO.C -----
// Pins 3.0 and 3.1 used for RS-232 interface

/*-----*-
----- END OF FILE -----
-*-----*/
```

Listing 18.1 Part of a generic PC link library (menu driven)

```

/*-----*-
Main.c (v1.00)

-----*
Demo program for PC Link (RS-232) pattern.

Linker options:

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)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Set baud rate to 9600: generic 8051 version
    PC_LINK_Init_T1(9600);

    // We have to schedule this task (10x - 100x a second)
    //
    // TIMING IS IN TICKS NOT MILLISECONDS (5 ms tick interval)
    SCH_Add_Task(MENU_Command_Processor,10,2);

    SCH_Start();

    while(1)
    {
        // Displays error codes on P4 (see Sch51.C)
        SCH_Dispatch_Tasks();
    }
}

/*-----*-
----- END OF FILE -----
-*-----*/
```

Listing 18.2 Part of a generic PC link library (menu driven)

```

/*-----*-
Menu_A.C (v1.00)

-----
Simple framework for interactive operation (e.g. data downloads)
to / from desktop / notebook PC (etc) via serial (UART) link.

Use 'Hyperterminal' (under Windows 95, 98, 2000) or similar
terminal emulator program on other operating systems.

Terminal options:

- Data bits      = 8
- Parity         = None
- Stop bits      = 1
- Flow control   = Xon / Xoff

-*-----*/
```

```
#include "Main.h"
#include "2_05_11g.h"
#include "Menu_A.h"
#include "Lnk_IO_A.h"

/*-----*
 * MENU_Command_Processor()
 *
 This function is the main menu 'command processor' function.
 Schedule this once every 10 ms (approx.).
 *-----*/
void MENU_Command_Processor(void)
{
    static bit First_time_only;
    char Ch;

    if (First_time_only == 0)
    {
        First_time_only = 1;
        MENU_Show_Menu();
    }

    // Check for user inputs
    PC_LINK_Update();

    Ch = PC_LINK_Get_Char_From_Buffer();

    if (Ch != PC_LINK_NO_CHAR)
    {
        MENU_Perform_Task(Ch);
        MENU_Show_Menu();
    }
}

/*-----*
 * MENU_Show_Menu()
 *
 Display menu options on PC screen (via serial link)
 - edit as required to meet the needs of your application.
 *-----*/
void MENU_Show_Menu(void)
{
    PC_LINK_Write_String_To_Buffer("Menu:\n");
    PC_LINK_Write_String_To_Buffer("a - AAA xxx\n");
    PC_LINK_Write_String_To_Buffer("b - BBB xxx\n");
}
```

```
    PC_LINK_Write_String_To_Buffer("c - CCC xxx\n\n");
    PC_LINK_Write_String_To_Buffer("Please type a character : ");
}

/*-----*/
MENU_Perform_Task()

Perform the required user task
- edit as required to match the needs of your application.

-*-----*/
void MENU_Perform_Task(char c)
{
// Echo the menu option
PC_LINK_Write_Char_To_Buffer(c);
PC_LINK_Write_Char_To_Buffer('\n');

// Perform the task
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_Write_String_To_Buffer("\n*** Doing C ***\n\n");
    P1 = 'C';
}

/* -----
----- END OF FILE -----
----- */

```

Listing 18.3 Part of a generic PC link library (menu driven)

```

/* -----
Lnk_IO_A.C (v1.00)

-----
Simple PC link library
Uses the USART, and Pins 3.1 (Tx) and 3.0 (Rx)
See text for details

*/
#include "Main.h"
#include "Lnk_IO_A.h"

// ----- Public variable declarations -----
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)
{
    PCON &= 0x7F; // Set SMOD bit to 0 (don't double baud rates)

    // Receiver enabled.
    // 8-bit data, 1 start bit, 1 stop bit,
    // Variable baud rate (asynchronous)
    // Receive flag will only be set if a valid stop bit is received
    // Set TI (transmit buffer is empty)
    SCON = 0x72;

    TMOD |= 0x20; // T1 in mode 2, 8-bit auto reload

    TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
        / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));

    TL1 = TH1;
    TR1 = 1; // Run the timer
    TI = 1; // Send first character (dummy)

    // Set up the buffers for reading and writing
    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");

    // Interrupt *NOT* enabled
    ES = 0;
}

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 18.4 Part of a generic PC link library (menu driven)

```
/*-----*
 * Lnk_IO.C (v1.00)
 *
 *-----*
 * Core files for simple PC link library for 8051 family
 * Uses the USART, and Pins 3.1 (Tx) and 3.0 (Rx)
 * See text for details.
 *
 *-----*/
#include "Main.h"
#include "Lnk_io.h"

// ----- Public variable definitions -----
tByte In_read_index_G;      // Data in buffer that has been read
tByte In_waiting_index_G;   // Data in buffer not yet read

tByte Out_written_index_G;  // Data in buffer that has been written
tByte Out_waiting_index_G;  // Data in buffer not yet written

// ----- Public variable declarations -----
// The error code variable
//
// See Main.H for port on which error codes are displayed
// and for details of error codes
extern tByte Error_code_G;

// ----- Private function prototypes -----
void PC_LINK_Send_Char(const char);

// ----- Private constants -----
// The receive buffer length
#define PC_LINK_RECV_BUFFER_LENGTH 8

// The transmit buffer length
#define PC_LINK_TRAN_BUFFER_LENGTH 100

// Value returned by PC_LINK_Get_Char if no character is
// available in buffer
#define PC_LINK_NO_CHAR 127

#define XON 0x11
#define XOFF 0x13

// ----- Private variables -----
```

```

static tByte Rcv_buffer[PC_LINK_RECV_BUFFER_LENGTH];
static tByte Tran_buffer[PC_LINK_TRAN_BUFFER_LENGTH];

/*-----*/
PC_LINK_Update()
Checks for character in the UART (hardware) receive buffer
Sends next character from the software transmit buffer
/*-----*/
void PC_LINK_Update(void)
{
    //
    // Deal with transmit bytes here

    // Is there any data ready to send?
    if (Out_written_index_G < Out_waiting_index_G)
    {
        PC_LINK_Send_Char(Tran_buffer[Out_written_index_G]);
        Out_written_index_G++;
    }
    else
    {
        //
        // No data to send - just reset the buffer index
        Out_waiting_index_G = 0;
        Out_written_index_G = 0;
    }

    // Only dealing with received bytes here
    // -> Just check the RI flag
    if (RI == 1)
    {
        //
        // Flag only set when a valid stop bit is received,
        // -> data ready to be read into the received buffer

        // Want to read into index 0, if old data have been read
        // (simple ~circular buffer)
        if (In_waiting_index_G == In_read_index_G)
        {
            In_waiting_index_G = 0;
            In_read_index_G = 0;
        }

        // Read the data from USART buffer
        Rcv_buffer[In_waiting_index_G] = SBUF;

        if (In_waiting_index_G < PC_LINK_RECV_BUFFER_LENGTH)
        {

```

```

        // Increment without overflowing buffer
        In_waiting_index_G++;
    }

    RI = 0; // Clear RT flag
}
}

/*-----*/
PC_LINK_Write_Char_To_Buffer()

Stores a character in the 'write' buffer, ready for
later transmission

/*-----*/
void PC_LINK_Write_Char_To_Buffer(const char CHARACTER)
{
    // Write to the buffer *only* if there is space
    if (Out_waiting_index_G < PC_LINK_TRAN_BUFFER_LENGTH)
    {
        Tran_buffer[Out_waiting_index_G] = CHARACTER;
        Out_waiting_index_G++;
    }
    else
    {
        // Write buffer is full
        // Increase the size of PC_LINK_TRAN_BUFFER_LENGTH
        // or increase the rate at which UART 'update' function is called
        // or reduce the amount of data sent to PC
        Error_code_G = ERROR_USART_WRITE_CHAR;
    }
}

/*-----*/
PC_LINK_Write_String_To_Buffer()

Copies a (null terminated) string to the character buffer.

/*-----*/
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++;
    }
}

```

```

        }

    }

/*-----*/
PC_LINK_Get_Char_From_Buffer()

Retrieves a character from the (input) buffer, if available

/*-----*/
char PC_LINK_Get_Char_From_Buffer(void)
{
    char Ch = PC_LINK_NO_CHAR;

    // If there is new data in the buffer
    if (In_read_index_G < In_waiting_index_G)
    {
        Ch = Recv_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()

Based on Keil sample code, with added (loop) timeouts.
Implements Xon / Off control.

/*-----*/
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;

    // Wait for uart (with simple timeout)
    Timeout1 = 0;
    while ((++Timeout1) && (RI == 0));

    if (Timeout1 == 0)
    {
        // USART did not respond - error
        Error_code_G = ERROR_USART_TI;
        return;
    }

} while ((++Timeout2) && (SBUF != XON));
if (Timeout2 == 0)
{
    // uart did not respond - error
    Error_code_G = ERROR_USART_TI;
    return;
}

RI = 0;
}
}

Timeout1 = 0;
while ((++Timeout1) && (TI == 0));

if (Timeout1 == 0)
{
    // uart did not respond - error
    Error_code_G = ERROR_USART_TI;
    return;
}

TI = 0;
SBUF = 0x0d; // output CR
if}(RI)
{
if (SBUF == XOFF)
{
    Timeout2 = 0;

    do {
        RI = 0;
```

```

        // Wait for USART (with simple timeout)
        Timeout1 = 0;
        while ((++Timeout1) && (RI == 0));

        if (Timeout1 == 0)
        {
            // USART did not respond - error
            Error_code_G = ERROR_USART_TI;
            return;
        }

    } while ((++Timeout2) && (SBUF != XON));

    RI = 0;
}
}

Timeout1 = 0;
while ((++Timeout1) && (TI == 0));

if (Timeout1 == 0)
{
    // USART did not respond - error
    Error_code_G = ERROR_USART_TI;
    return;
}

TI = 0;
SBUF = CHARACTER;
}

/*
-----*-
---- END OF FILE -----
-*-----*-
*/

```

Listing 18.5 Part of a generic PC link library (menu driven)

Example: Using a dedicated baud rate generator

As we noted in ‘Solution’, some 8051 microcontrollers have a timer dedicated to baud rate generation. This can be a useful facility, since it leaves us (usually) with three free timers for use in the scheduler, creation of delays and so on.

In this example, we present an ‘output only’ library for the Infineon C515c (listings 18.6 to 18.11). This uses the C515C baud rate generator. In the example, we display elapsed time on the screen of a PC or similar terminal.

There are two main points to note in this example:

- The library is considerably smaller and simpler than the ‘duplex’ library presented earlier.
- The microcontroller is running at 10 MHz; despite this, the internal baud rate generator allows us to create a very accurate baud rate for the serial link. This is a very useful feature.

```
/* -----
Port.H (v1.00)

-----
'Port Header' (see Chapter 10) for the project 0_T2_IN
----- */

// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#ifndef SCH_REPORT_ERRORS

#define SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif

// ----- Lnk_IO.C -----
// Pins 3.0 and 3.1 used for RS-232 interface

/* -----
--- END OF FILE ---
----- */
*/
```

Listing 18.6 Part of an example using a dedicated timer for baud rate generation on the Infineon c515c

```
/* -----
Main.c (v1.00)

-----
Test program for output-only PC link library, based on c515c.
```

Linker options:

```

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_10i.h"
#include "Lnk_0_B.h"
#include "Elap_232.h"

/* ..... */
/* ..... */

void main(void)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Set baud rate to 9600, using internal baud rate generator
    PC_LINK_Init_Internal(9600);

    // Prepare the elapsed time library
    Elapsed_Time_RS232_Init();

    // We have to schedule this task (~100x a second is enough here)
    // - this writes data to PC
    //
    // TIMING IS IN TICKS (1 ms tick interval)
    SCH_Add_Task(PC_LINK_Update, 10, 10);

    // Update the time once per second
    SCH_Add_Task(Elapsed_Time_RS232_Update, 1000, 1000);

    SCH_Start();

    while(1)
    {
        // Displays error codes on P4 (see Sch51.C)
        SCH_Dispatch_Tasks();
    }
}

----- END OF FILE -----
----- */

```

Listing 18.7 Part of an example using a dedicated timer for baud rate generation on the Infineon c515c

```
/*-----*
Elap_232.C (v1.00)

-----
Simple library function for keeping track of elapsed time
Demo version to display time on PC screen via RS232 link.

-----*/
#include "Main.h"
#include "Elap_232.h"
#include "Lnk_0.h"

// ----- Public variable definitions -----
tByte Hou_G = 0;
tByte Min_G = 0;
tByte Sec_G = 0;

// ----- Public variable declarations -----
// See Char_Map.c
extern const char code CHAR_MAP_G[10];
/*-----*/
Elapsed_Time_RS232_Init()

Init function for simple library displaying elapsed time on PC
screen.

-----*/
void Elapsed_Time_RS232_Init(void)
{
    char Time_Str[] = 'Elapsed time';

    PC_LINK_Write_String_To_Buffer(Time_Str);
}

/*-----*/
Elapsed_Time_RS232_Update()

Function for displaying elapsed time on PC Screen.

*** Must be scheduled once per second ***

-----*/
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];

    // We don't display seconds in this version.
    // We simply use the seconds data to turn on and off the colon
    // (between hours and minutes)
    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);
}

/* -----
--- END OF FILE ---
----- */

```

Listing 18.8 Part of an example using a dedicated timer for baud rate generation on the Infineon c515c

```

/* -----
Char_Map.C (v1.00)

-----
This lookup table (stored in ROM) is used to convert 'integer'
values into appropriate character codes for the LCD display
and RS232 link.

----- */
// ----- Public constants -----
const char code CHAR_MAP_G[10]
    = {'0','1','2','3','4','5','6','7','8','9'};

/* -----
----- END OF FILE -----
----- */

```

Listing 18.9 Part of an example using a dedicated timer for baud rate generation on the Infineon c515c

```

/* -----
Lnk_0_B.C (v1.00)

-----
Simple PC link library version B (for c515c, internal baud rate)
Uses the USART, and Pin 3.1 (Tx)
See text for details

----- */
#include "Main.h"
#include "Lnk_0_B.h"

// ----- Public variable declarations -----
extern tByte Out_written_index_G;
extern tByte Out_waiting_index_G;

/* -----
PC_LINK_Init_Internal()

This version uses internal baud rate generator on C5x5 family.

----- */

```

```

void PC_LINK_Init_Internal(const tWord BAUD_RATE)
{
    tWord SRELplus1024;
    tWord SRELx;

    PCON &= 0x7F; // Set SMOD bit to 0 (don't double baud rates)

    // Receiver disabled
    // 8-bit data, 1 start bit, 1 stop bit, variable baud rate (asynchronous)
    SCON = 0x42;

    // Use the internal baudrate generator (80c5x5 family)
    ADCON0 |= 0x80;
    // Set the baud rate (begin)
    SRELplus1024 = (tWord)((((tLong) OSC_FREQ / 100) * 3125)
                           / ((tLong) BAUD_RATE * 1000));

    SRELx = 1024 - SRELplus1024;

    SRELL = (tByte)(SRELx & 0x00FF);
    SRELH = (tByte)((SRELx & 0x0300) >> 8);

    TI = 1;
    // Set the baud rate (end)

    // Set up the buffers for writing
    Out_written_index_G = 0;
    Out_waiting_index_G = 0;

    PC_LINK_Write_String_To_Buffer("Serial interface initialized.\n");

    // Serial interrupt *NOT* enabled
    ES = 0;
}

/*-----*
----- END OF FILE -----*
-----*/
```

Listing 18.10 Part of an example using a dedicated timer for baud rate generation on the Infineon c515c

```

/*-----*
Lnk_0.C (v1.00)

-----*
Core files for simple write-only PC link library for 8051 family
[Sends data to PC - cannot receive data from PC]
```

Uses the USART, and Pin 3.1 (Tx)

See text for details.

```
-----*/  
#include "Main.h"  
#include "Lnk_0.h"  
  
// ----- Public variable definitions -----  
  
tByte Out_written_index_G; // Data in buffer that has been written  
tByte Out_waiting_index_G; // Data in buffer not yet written  
  
// ----- Public variable declarations -----  
  
// The error code variable  
//  
// See Port.H for port on which error codes are displayed  
// and for details of error codes  
extern tByte Error_code_G;  
  
// ----- Private constants -----  
  
// The transmit buffer length  
#define PC_LINK_TRAN_BUFFER_LENGTH 100  
  
// ----- Private variables -----  
  
static tByte Tran_buffer[PC_LINK_TRAN_BUFFER_LENGTH];  
-----*/  
  
PC_LINK_Update()  
  
Sends next character from the software transmit buffer  
  
NOTE: Output-only library (Cannot receive chars)  
-----*/  
void PC_LINK_Update(void)  
{  
    // Deal with transmit bytes here  
  
    // Is there any data ready to send?  
    if (Out_written_index_G < Out_waiting_index_G)  
    {  
        PC_LINK_Send_Char(Tran_buffer[Out_written_index_G]);  
  
        Out_written_index_G++;  
    }  
    else  
    {
```

```

        // No data to send - just reset the buffer index
        Out_waiting_index_G = 0;
        Out_written_index_G = 0;
    }

}

/*-----*/
PC_LINK_Write_Char_To_Buffer()

Stores a character in the 'write' buffer, ready for
later transmission

/*-----*/
void PC_LINK_Write_Char_To_Buffer(const char CHARACTER)
{
    // Write to the buffer *only* if there is space
    if (Out_waiting_index_G < PC_LINK_TRAN_BUFFER_LENGTH)
    {
        Tran_buffer[Out_waiting_index_G] = CHARACTER;
        Out_waiting_index_G++;
    }
    else
    {
        // Write buffer is full
        // Increase the size of PC_LINK_TRAN_BUFFER_LENGTH
        // or increase the rate at which UART 'update' function is called
        // or reduce the amount of data sent to PC
        Error_code_G = ERROR_USART_WRITE_CHAR;
    }
}

/*-----*/
PC_LINK_Send_Char()

Based on Keil sample code, with added (loop) timeouts.
Implements Xon / Off control.

/*-----*/
void PC_LINK_Send_Char(const char CHARACTER)
{
    tLong Timeout1 = 0;

    if (CHARACTER == '\n')
    {
        Timeout1 = 0;

```

```
        while ((++Timeout1) && (TI == 0));
        if (Timeout1 == 0)
        {
            // usart did not respond - error
            Error_code_G = ERROR_USART_TI;
            return;
        }

        TI = 0;
        SBUF = 0x0d; // output CR
    }

Timeout1 = 0;
while ((++Timeout1) && (TI == 0));

if (Timeout1 == 0)
{
    // usart did not respond - error
    Error_code_G = ERROR_USART_TI;
    return;
}

TI = 0;
SBUF = CHARACTER;
}

/*-----*/
PC_LINK_Write_String_To_Buffer()

Copies a (null terminated) string to the character buffer.
(The contents of the buffer are then passed over the serial link.)

/*-----*/
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++;
    }
}
```

```

/* -----
----- END OF FILE -----
*/

```

Listing 18.11 Part of an example using a dedicated timer for baud rate generation on the Infineon c515

Example: An output-only library for the 8051/8052 (generic)

This example again illustrates how to link a Standard 8051 device to a PC.

Unlike the first example, the software is ‘write only’: data are transferred from the microcontroller to the PC but not vice versa. To illustrate this, the software displays elapsed time on the PC via a terminal emulator program.

In Figure 18.6, we illustrate this software running on the Keil hardware simulator. Of course, it may equally well display its outputs using Hyperterminal (or equivalent), as in Figure 18.3: however, the use of the hardware simulator is useful during the early stages of program development.

The hardware shown in Figure 18.2 can again be used here to run this program.

The source code for this example is included on the CD.

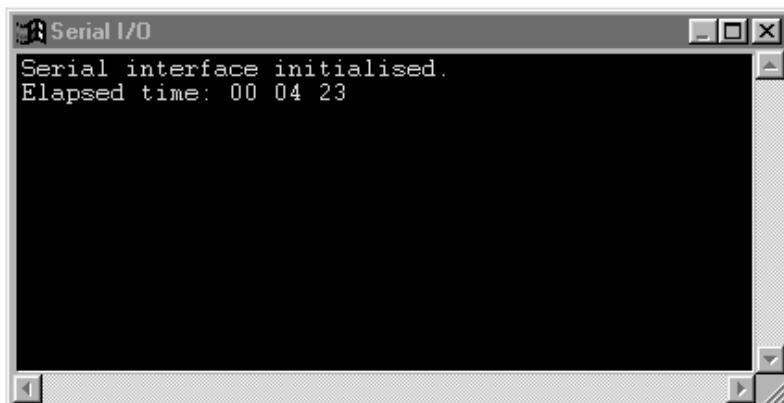


FIGURE 18.6 Output from a simple ‘output only’ PC link library running under the Keil hardware simulator.

Further reading

In a very readable book, Axelson (1998) considers the use of RS-232 communications. Although she discusses 8051 microcontrollers, the main focus is on the PC end of the communication link. Axelson provides some useful code examples that can be adapted for use in many PC-based data acquisition and monitoring systems.

Switch interfaces

Introduction

Reading the status of one or more push-button switches is a very common requirement in embedded applications. The four patterns in this chapter describe different solutions to this problem.

The patterns are as follows:

- **SWITCH INTERFACE (SOFTWARE)** [page 399]
Uses a minimum of external hardware. Reads a single switch, debounces it and reports its status
- **SWITCH INTERFACE (HARDWARE)** [page 410]
Use external hardware to perform switch debouncing. Increased cost (compared with **SWITCH INTERFACE (SOFTWARE)**), but much higher level of protection against ESD, malicious damage etc.
- **ON-OFF SWITCH** [page 414]
Building on **SWITCH INTERFACE (SOFTWARE)** OR **SWITCH INTERFACE (HARDWARE)** this pattern provides, through software, the following behaviour:
Assume that the switch is in the OFF state. It remains in this state until it is pressed. When pressed, the state changes to ON. It remains in this state until it is pressed. When pressed, the state changes to OFF.
This type of behaviour can be used, for example, to allow a single (non-latching) switch to control a piece of machinery.
- **MULTI-STATE SWITCH** [page 423]
Building on **SWITCH INTERFACE (SOFTWARE)** OR **SWITCH INTERFACE (HARDWARE)** this pattern provides, through software, the following behaviour:
Assume that the switch state is in the OFF state. The switch is pressed and briefly held. The switch state changes to 'ON STATE 1'. The user continues to depress the switch. The switch state becomes 'ON STATE 2', etc. Releasing the switch (at any time) puts the switch back in the OFF state.

That is, the switch state depends on the duration of the switch press. Any number of 'ON' states may be supported (although having more than around three is generally very confusing for the user).

This type of behaviour can be useful when, for example, setting time on a clock. Pressing the set switch will initially cause the displayed time to change slowly; sustained depressions will cause the displayed time to increase more rapidly.

Note: For reasons discussed in **SWITCH INTERFACE (SOFTWARE)**, in 'Safety and reliability', we are only concerned with push-button switches in these patterns: 'toggle' or 'latching' switches are not considered (and the use of such switches is not recommended).

SWITCH INTERFACE (SOFTWARE)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you connect the port pin of an 8051 microcontroller to some form of mechanical switch (for example, a simple push-button switch or an electromechanical relay)?

Background

Consider the simple push-button switches illustrated in Figure 19.1.

Depressing this switch will, in either arrangement, cause a voltage change from approximately Vcc to 0V at the input port.

In an ideal world, this change in voltage would take the form illustrated in Figure 19.2 (top). In practice, all mechanical switch contacts *bounce* (that is, turn on and off, repeatedly, for a short period of time) after the switch is closed or opened. As a result, the actual input waveform looks more like that shown in Figure 19.2 (bottom). Usually, switches bounce for less than 20 ms: however large mechanical switches

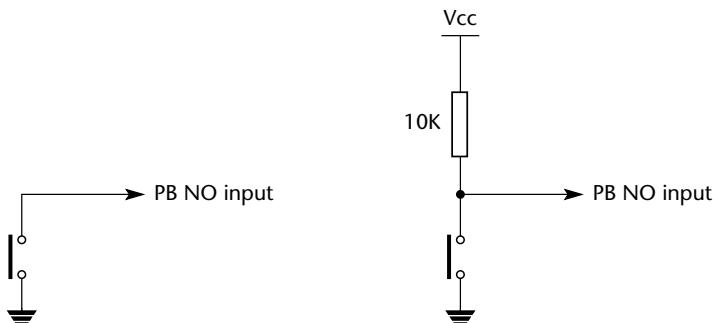


FIGURE 19.1 An example of a push-button ('normally open') switch input

[Note: The resistor-free arrangement [left] is appropriate where port pins have an internal pull resistor: this is usually the case on the 8051 family, with the exception of Port 0. Where there is no internal pull-up, the arrangement on the right must be used.]

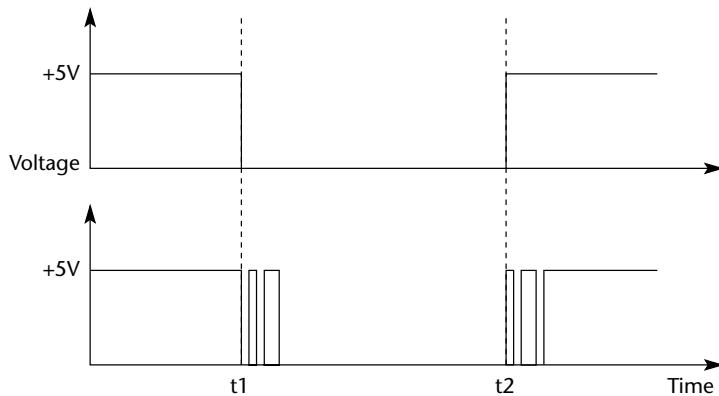


FIGURE 19.2 The voltage signal resulting from the switch shown in Figure 19.1

[Note: [Top] Idealized waveform resulting from a switch depressed at time t_1 and released at time t_2 . [Bottom] Actual waveform showing leading edge bounce following switch depression and trailing edge bounce following switch release.]

exhibit bounce behaviour for 50 ms or more.

This bounce is equivalent to pressing an idealized switch multiple times. This causes various potential problems, not least:

- If we need to distinguish between single and multiple switch presses: for example, rather than reading 'A' from a keypad, we read 'AAAAA'.
- If we wish to count the number of switch presses.
- If we need to distinguish between a switch being depressed and being released: for example, if the switch is depressed once and then released some time later, the bounce will make it appear as if the switch has been pressed again.

Solution

Checking for a switch input is, in essence, straightforward:

- 1 We read the relevant port pin.
- 2 If we think we have detected a switch depression, we read the pin again (say) 20 ms later.
- 3 If the second reading confirms the first reading, then we assume the switch really has been depressed.

Note that the figure of '20 ms' will, of course, depend on the switch used: the data sheet of the switch will provide this information. If you have no data sheet, you can either experiment with different figures or measure directly using an oscilloscope.

We illustrate this basic procedure in the example.

Hardware resource implications

Reading a switch input in software imposes very minor loads on CPU and memory resources.

Reliability and safety issues

We consider some reliability and safety issues associated with switches here.

Latching vs. push-button switches

All the examples in this chapter (and throughout this book) focus on the use of push-button (rather than 'latching') switches.

To illustrate why this is the case, we will consider two possible designs for a switch-based interface (Figure 19.3). The first interface design uses a 'latching' switch: specifically, this is a 5-position, rotary switch. The second design uses two push-button switches and five LEDs to provide similar functionality.

We can immediately see some advantages of the push-button solution when we consider how each system behaves when power is first applied and in an emergency situation.

In the case of the push-button solution, the software designer has control of the initial state. For example, if it is appropriate to always start the system in 'State 2' (as illustrated in Figure 19.3), this can be easily achieved. Similarly, suppose that State 5 involves control of a piece of machinery: if a problem develops with the machine, we might need to change the system to (say) State 4. Again, we can do this under software control **and we can make this change clear to the user simply by lighting the appropriate LED**.

The rotary switch interface behaves very differently. If the user moves the switch to Position 5 before applying power, then, at power up, how do we move the system to State 2? And if we do change the state to State 2, we cannot indicate this fact to the user, since we are unable to move the rotary switch. Similarly, in the emergency situation just discussed, we may be able to shut down the machine under software control, but we cannot show the user that the system is now in State 4.

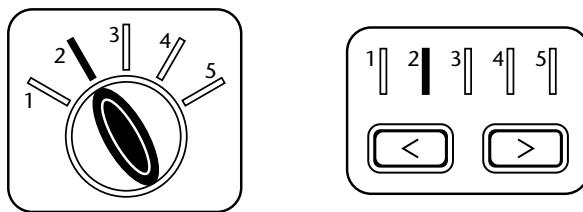


FIGURE 19.3 Two possible switch-based interfaces

[Note: The latching solution (left) is not recommended for reasons discussed in the text.]

These problems arise not just with multi-state rotary switches but with any form of latching switch.

We can summarize the advantage of the push-button solution by saying that it **leaves the software in control of the hardware**, rather than vice versa. This is a good general design guideline in embedded systems.

Switch bounce times

In electrical engineering terms, the switches used as inputs in a typical microcontroller system are ‘dry’. This means that, because they must handle only small currents ($< 20\text{ mA}$) and voltages ($< 10\text{ V}$), they suffer little (electrical) ‘wear and tear’ and can be expected to have a long life.

However, switches also have moving mechanical parts and their performance, inevitably, varies with age. Even a good quality switch will change its performance as it is used: in particular, the period of bounce tends to increase with age. As a result, it is important not to trust to luck or experiments with new switches when determining appropriate debounce times: you should allow a margin (at least 20%) with new switches to allow for the impact of prolonged usage.

Using single-pole switches

We have already discussed the advantages of push-button (PB) switches. Unfortunately, simply using a PB switch is not enough to ensure reliability. For example, consider again the switch in Figure 19.1: this type of switch is often referred to as ‘normally open’ (NO). If this form of NO switch is removed or damaged, leaving the connection permanently open, we cannot detect this fact in software. This may have safety or reliability implications.

In some circumstances, there may be advantages to using a ‘normally closed’ (NC) PB switch (Figure 19.4). Using a PB-NC switch, we can detect the removal of the switch or damage to the wiring, although we cannot generally distinguish this from a normal (sustained) switch depression.

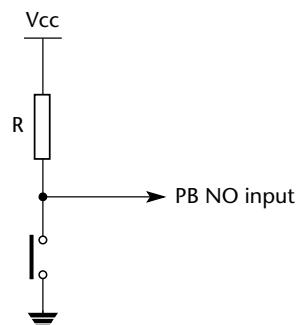


FIGURE 19.4 An example of a push-button (‘normally closed’) switch input

We can provide an even more reliable solution by using a PB ‘double-pole, double-throw’ (PB-DPDT) switch (Figure 19.5). This generates two inputs, which will always have opposite logic if the switch is undamaged and wired correctly. Using such an input device, we can detect various types of switch faults (including switch removal) and wiring faults (including wire cutting).

Note that such a switch requires two input pins and slightly more software than a single-pole switch input.

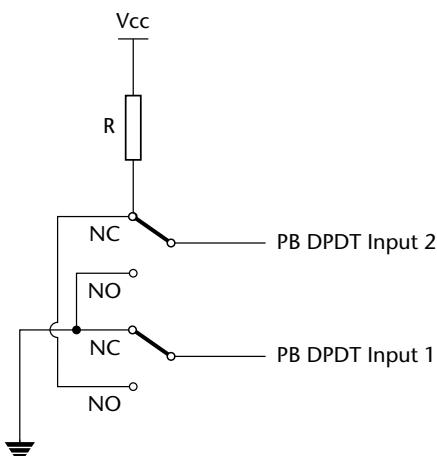


FIGURE 19.5 An example of a push-button DPDT switch input

[Note: With the arrangement shown, Input 1 will have a voltage of 0V and Input 2 Vcc with the switch open: these values will be reversed when the switch is closed.]

Out-of-range inputs and ESD

Like all software-only input techniques, **SWITCH INTERFACE (SOFTWARE)** provides no protection against out-of-range inputs: if someone applies +/-20V to your switch (by mistake or deliberately), they may ‘fry’ the microcontroller.

Similar problems arise in the event of electrostatic discharge (ESD). ESD can present a significant problem in harsh environments (e.g. industrial systems, automotive applications) and compliance with international standards in this area (e.g. IEC 1000-4-2) is a requirement for some applications.

There is little you can do, in software, to guard again either of these situations. See **SWITCH INTERFACE (HARDWARE)** [page 410] for a discussion of this issue and a solution.

Portability

None of these discussions in this pattern is 8051 specific: the pattern may be used with other microcontroller families without difficulty.

Overall strengths and weaknesses

- 😊 **SWITCH INTERFACE (SOFTWARE)** requires a minimum of external hardware.
- 😊 It is very flexible: for example, the programmer can incorporate ‘auto-repeat’ functions with few code changes.
- 😊 It is simple and cheap to implement.
- 😢 Like all software-only input techniques, **SWITCH INTERFACE (SOFTWARE)** provides no protection against out-of-range inputs or electrostatic discharge (ESD). See **SWITCH INTERFACE (HARDWARE)** [page 410] for a discussion of this issue and a solution.

Related patterns and alternative solutions

Reading a switch input is a classic example of a situation where we can trade hardware cost against software complexity. For an alternative approach using hardware, consider **SWITCH INTERFACE (HARDWARE)** [page 410].

Where the switch (or other input device) does not suffer from bounce (for example, where your input is derived from a solid-state relay or some other form of ‘electronic switch’) then much of the complexity of these input strategies can be avoided through the use of **PORT I/O** [page 174].

Example: Controlling a flashing LED

In this example, we use a push-button switch to control the flashing of an LED. Here the LED flashes only when the switch is pressed.

The hardware used is illustrated in Figure 19.6 and the software framework is given in Listings 19.1 to 19.4.

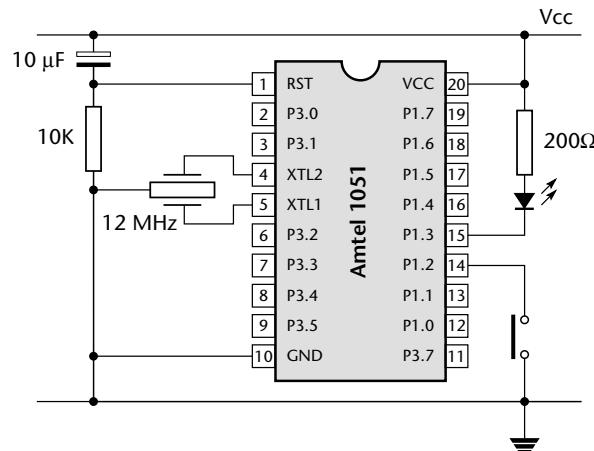


FIGURE 19.6 Some test hardware used to demonstrate a simple, software-based switch interface

```

/*-----*
Port.H (v1.00)

-----'
'Port Header' (see Chapter 10) for the project SWITCH_A
-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif

// -----Swit_A.C -----
// Connect single push-button switch on this pin (to gnd)
// - debounced in software
sbit Sw_pin = P1^2; // The switch pin

// ----- LED_Swit.C -----
// Connect LED from +5V (etc) to this pin, via appropriate resistor
// [see Chapter 7 for details]
sbit LED_pin = P1^3;

/*-----*
---- END OF FILE -----
-----*/

```

Listing 19.1 Part of the software used to demonstrate a simple, software-based switch interface

```

/*-----*
Main.c (v1.00)

-----'

Demo program for SWITCH INTERFACE (SOFTWARE) pattern.

See Chapter 19 for details.

Required linker options (see text for details):
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)
{
    // Set up the scheduler
    SCH_Init_T0();

    // Set up the switch pin
    SWITCH_Init();

    // Prepare for the 'Flash_LED' task
    LED_Flash_Switch_Init();

    // Add a 'SWITCH_Update' task, every ~200 ms.
    // Scheduler timings is in ticks.
    // [1 ms tick interval - see Sch 'init' function]
    SCH_Add_Task(SWITCH_Update, 0, 200);

    // Add LED task
    // Here, LED will only flash while switch is pressed...
    SCH_Add_Task(LED_Flash_Switch_Update, 5, 1000);

    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

----- END OF FILE -----
----- */

```

Listing 19.2 Part of the software used to demonstrate a simple, software-based switch interface

```

----- */
LED_Swit..C (v1.00)
----- */

```

Simple 'Flash LED' test function for scheduler.

(Controlled by switch press)

```

-----*/
#include "Main.h"
#include "Port.h"
#include "LED_Switch.h"

// ----- Public variable declarations -----
extern bit Sw_pressed_G;

// ----- Private variables -----
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()
Flashes an LED (or pulses a buzzer, etc) on a specified port pin.
Must schedule at twice the required flash rate: thus, for 1 Hz
flash (on for 0.5 seconds, off for 0.5 seconds) must schedule
at 2 Hz.

-----*/
void LED_Flash_Switch_Update(void)
{
    // Do nothing if switch is not pressed
    if (!Sw_pressed_G)
    {
        return;
    }

    // Change the LED from OFF to ON (or vice versa)
    if (LED_state_G == 1)

```

```

    {
        LED_state_G = 0;
        LED_pin = 0;
    }
else
{
    LED_state_G = 1;
    LED_pin = 1;
}
}

/* -----
----- END OF FILE -----
----- */

```

Listing 19.3 Part of the software used to demonstrate a simple, software-based switch interface

```

/* -----
----- SWIT_A.C (v1.00)

-----
Simple switch interface code, with software debounce.

----- */
#include "Main.h"
#include "Port.h"

#include "Swit_A.h"

// ----- Public variable definitions -----
bit Sw_pressed_G = 0; // The current switch status

// ----- Private constants -----
// Allows NO or NC switch to be used (or other wiring variations)
#define SW_PRESSED (0)

// SW_THRES must be > 1 for correct debounce behaviour
#define SW_THRES (3)

/* -----
----- SWITCH_Init()

Initialization function for the switch library.
----- */

```

```

void SWITCH_Init(void)
{
    Sw_pin = 1; // Use this pin for input
}

/*
 *-----*
SWITCH_Update()

This is the main switch function.

It should be scheduled every 50 - 500 ms.

*-----*/
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; // Switch is pressed...
            return;
        }

        // Switch pressed, but not yet for long enough
        Sw_pressed_G = 0;
        return;
    }

    // Switch not pressed - reset the count
    Duration = 0;
    Sw_pressed_G = 0; // Switch not pressed...
}

/*
 *-----*
---- END OF FILE -----
*-----*/

```

Listing 19.4 Part of the software used to demonstrate a simple, software-based switch interface

Further reading

SWITCH INTERFACE (HARDWARE)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you create a very robust switch interface for use in a hostile (e.g. industrial, automotive) environment?

Background

Consider the following scenarios:

- Your embedded application is used in an industrial environment where high levels of electrostatic discharge are likely. How can you ensure that your device remains fully operational?
- Your automotive security application may be the subject of deliberate vandalism or damage. Specifically, reports on the WWW have revealed that thieves have found it possible to disable a similar security system by applying 12V from a car battery directly to one of the system switches. How can you ensure that your system is more robust?

In general, **SWITCH INTERFACE (SOFTWARE)** [page 399] describes techniques that are only suitable in ‘safe’ applications: in hostile environments, you need a more robust solution. This must be hardware based.

Solution

As noted in ‘Background’, creating a robust switch interface requires the use of off-chip hardware.

Traditionally, techniques involving J-K flip-flops, high-impedance CMOS gates or R-C integrators have all been used for switch debouncing: Huang (2000), for example, provides details of these techniques. In general, these approaches – while performing the debounce operation – provide only very limited protection, at best, again ESD and similar hazards. Because, as we saw in **SWITCH INTERFACE (SOFTWARE)** [page 399], the process of switch debouncing is almost trivial in a scheduled application, the cost of external hardware for switch debouncing alone cannot generally be justified.

More recently, several specialized ICs for protection and switch debouncing have appeared on the market. Of these, the Maxim 6816/6817/6818 family are a good example (Figure 19.7).

This is how Maxim describes these devices:

- The Max6816/Max6817/Max6818 are single, dual, and octal switch debouncers that provide clean interfacing of mechanical switches to digital systems. They accept one or more bouncing inputs from a mechanical switch and produce a clean digital output after a short, preset qualification delay. Both the switch opening bounce and the switch closing bounce are removed.
- Robust inputs can exceed power supplies by up to $\pm 25V$.
- ESD protection for input pins:
 - $\pm 15\text{ kV}$ Human Body Model
 - $\pm 8\text{ kV}$ IEC 1000-4-2, Contact Discharge
 - $\pm 15\text{ kV}$ IEC 1000-4-2, Air-Gap Discharge
- Single-supply operation from $+2.7\text{V}$ to $+5.5\text{V}$.
- Single (Max6816), dual (Max6817) and octal (Max6818) versions available.
- No external components required.
- $6\text{ }\mu\text{A}$ supply current.

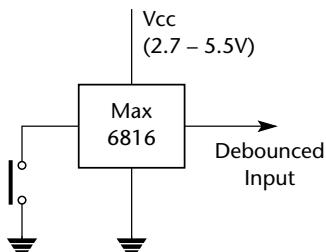


FIGURE 19.7 Typical application circuit for the Max6816

Hardware resource implications

Reading a debounced switch input imposes minimal loads on CPU and memory resources.

Reliability and safety issues

For the reasons discussed in ‘Solution’, this is a highly reliable method for creating a switch interface.

For additional reliability – particularly in the event of malicious damage – see **SWITCH INTERFACE (SOFTWARE)** [page 399] for discussions on the use of multi-pole switches.

Portability

These techniques are inherently portable.

Overall strengths and weaknesses

- 😊 Greatly increased reliability (compared with software-only solutions) in hostile environments.
- 😢 Increased costs and hardware complexity.

Related patterns and alternative solutions

See **SWITCH INTERFACE (SOFTWARE)** [page 399].

Example: Reading 8 switch inputs in a hostile environment

Figure 19.8 illustrates the use of a Max6818 to read the inputs from eight switches connected to Port 1 of an 8051 device. The results are reported on Port 2.

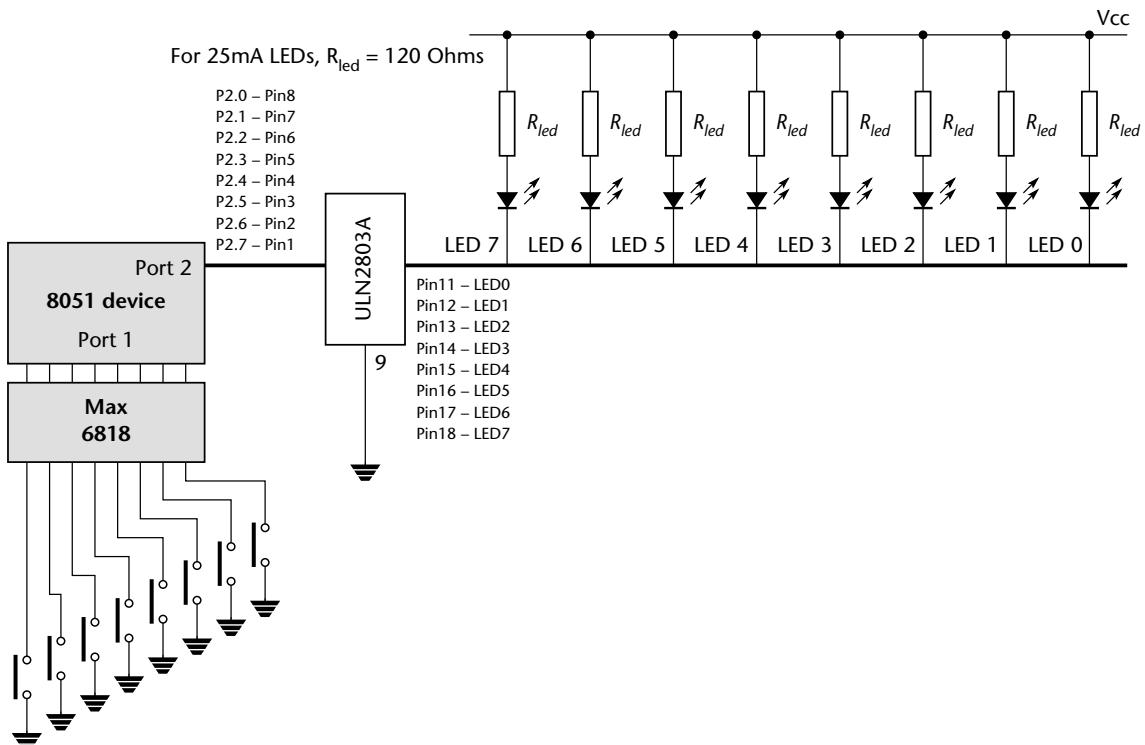


FIGURE 19.8 Using a Max6818 to debounce eight switches

The software requirements can be most easily met using a **SUPER LOOP** [page 162], as shown in Listing 19.5.

```
void main(void)
{
    while(1)
    {
        P2 = P1;
    }
}
```

Listing 19.5 A trivial Super Loop switch interface

Further reading

ON-OFF SWITCH

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you obtain the behaviour illustrated in Figure 19.9 from a single, push-button switch connected to the port pin of a microcontroller?

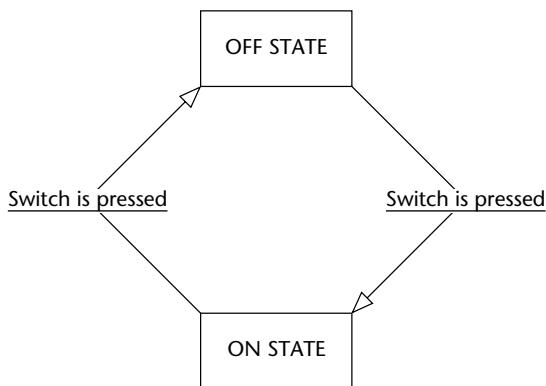


FIGURE 19.9 Creating an 'on-off' (latching) behaviour using a single push-button switch

Background

Consider a problem that can arise when we have a single switch used to turn on and off a piece of equipment (Figure 19.10).

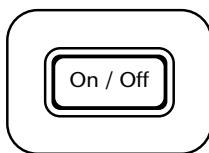


FIGURE 19.10 Illustrating the 'on-off' problem

The switch is intended to operate as follows:

- The user presses the switch to turn on a piece of equipment.
- The equipment operates as required.
- The user presses the switch again to turn off the equipment.

This seems very straightforward. However, suppose we are applying the basic approach to switch reading presented in **SWITCH INTERFACE (SOFTWARE)** [page 399], or **SWITCH INTERFACE (HARDWARE)** [page 410]. This can be summarized as follows:

- 1 We read the relevant port pin.
- 2 If we think we have detected a switch depression, we read the pin again 100 ms later.
- 3 If the second reading confirms the first reading, we assume the switch really has been depressed.

This is what can happen:

- The user presses the switch to turn on the piece of equipment.
- The switch is checked. It is depressed.
- The switch is checked (say) 100 ms later: the second check confirms the first. The equipment is turned on.
- The switch is checked 100 ms later. It is still depressed.
- The switch is checked 100 ms later: the second check confirms the first. The equipment is turned off again.
- And so on.

This behaviour arises because the user will generally wait until the equipment begins to work and will then remove their finger from the switch. In the best case scenario, a switch depression is likely to last about 500 ms. Unless we take action to prevent it, the equipment will ‘flicker’ on and off.

Solution

We can most simply create an on-off switch by adding a ‘switch block’ counter to the existing interface code. This works as follows:

- 1 Every time we find the switch has been pressed, we ‘block’ it for – say – 1 second.
- 2 While the switch is blocked, any changes in switch status are ignored: thus, for example, if the user keeps the switch depressed for half a second, this fact will be ignored.

The ‘on-off’ example that follows illustrates how this is achieved in practice.

Hardware resource implications

Reading a switch input imposes minimal loads on CPU and memory resources.

Reliability and safety issues

In hostile environments, this code should be based on **SWITCH INTERFACE (HARDWARE)** [page 410].

Note that, where safety is a primary concern, the blocking of switch inputs may not be appropriate. As an alternative, you can retain the basic switch-handling approach, but use two switches (Figure 19.11).

The use of two switches can make it easier to react quickly to changes in the inputs and may be safer under some circumstances.

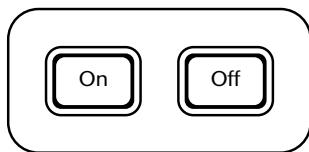


FIGURE 19.11 One approach to solving the 'on-off' problem

Portability

This pattern may be adapted for use with other microcontrollers without difficulty.

Overall strengths and weaknesses

☺ A simple way of achieving 'ON-OFF' behaviour from a single switch.

Related patterns and alternative solutions

This pattern builds on **SWITCH INTERFACE (SOFTWARE)** [page 399] and / or **SWITCH INTERFACE (HARDWARE)** [page 410].

See 'Reliability and safety issues' for an alternative solution.

Example: Controlling a flashing LED

In this example, we use the hardware in Figure 19.12 to illustrate the creation of an on-off switch interface.

The key software files required in this example are presented in Listings 19.6 to 19.9: as usual, a complete set of files for the project are included on the CD.

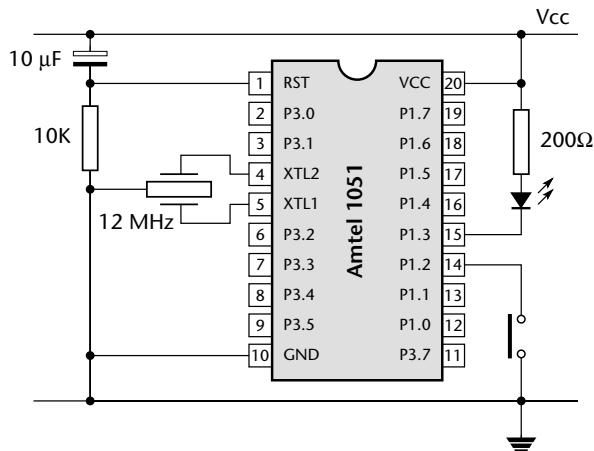


FIGURE 19.12 Hardware for an 'on-off' LED demonstrator

```

/* ----- *
Port.H (v1.00)

----- */

'Port Header' (see Chapter 10) for the project ON_OFF
----- */

// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#ifndef SCH_REPORT_ERRORS

#define SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif

// -----Swit_C.C -----
// Connect single push-button switch on this pin (to gnd)
// - debounced in software
sbit Sw_pin = P1^2; // The switch pin

// ----- LED_Swit.C -----
// Connect LED from +5V (etc) to this pin, via appropriate resistor
// [see Chapter 7 for details]
sbit LED_pin = P1^3;

```

```
/*-----*  
---- END OF FILE -----  
-----*/
```

Listing 19.6 Part of an example demonstrating a software-based switch interface

```
/*-----*  
-----*  
Main.c (v1.00)  
-----*  
  
Demo program for ON-OFF SWITCH pattern.  
  
See Chapter 19 for details.  
  
Required linker options (see text for details):  
  
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)  
{  
  
    // Set up the scheduler  
    SCH_Init_T0();  
  
    // Set up the switch pin  
    SWITCH_ON_OFF_Init();  
  
    // Prepare for the 'Flash_LED' task  
    LED_Flash_Switch_Init();  
  
    // Add a 'SWITCH_ON_OFF_Update' task, every ~200 ms.  
    // Scheduler timings is in ticks.  
    // [1 ms tick interval - see Sch 'init' function]  
    SCH_Add_Task(SWITCH_ON_OFF_Update, 0, 200);  
  
    // Add LED task  
    // Here, LED will only flash while switch is in ON state  
    SCH_Add_Task(LED_Flash_Switch_Update, 0, 1000);
```

```
SCH_Start();  
  
while(1)  
{  
    SCH_Dispatch_Tasks();  
}  
}  
  
/*-----*  
---- END OF FILE -----  
*-----*/
```

Listing 19.7 Part of an example demonstrating a software-based switch interface

```
/*-----*  
LED_Swit.C (v1.00)  
  
-----  
Simple 'Flash LED' test function for scheduler.  
(Controlled by switch press)  
*-----*/  
  
#include "Main.h"  
#include "Port.h"  
  
#include "LED_Swit.h"  
  
// ----- Public variable declarations -----  
extern bit Sw_pressed_G;  
  
// ----- Private variables -----  
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()
Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

Must schedule at twice the required flash rate: thus, for 1 Hz
flash (on for 0.5 seconds, off for 0.5 seconds) must schedule
at 2 Hz.

----- */
void LED_Flash_Switch_Update(void)
{
    // Do nothing if switch is not pressed
    if (!Sw_pressed_G)
    {
        return;
    }

    // Change the LED from OFF to ON (or vice versa)
    if (LED_state_G == 1)
    {
        LED_state_G = 0;
        LED_pin = 0;
    }
    else
    {
        LED_state_G = 1;
        LED_pin = 1;
    }
}

/* -----
----- END OF FILE -----
----- */

```

Listing 19.8 Part of an example demonstrating a software-based switch interface

```

/* -----
SWIT_C.C (v1.00)

-----
On-Off Switch code, with software debounce.

----- */

```

```
#include "Main.h"
#include "Port.h"
#include "Swit_C.h"

// ----- Public variable definitions -----
bit Sw_pressed_G = 0; // The current switch status

// ----- Private constants -----
// Allows NO or NC switch to be used (or other wiring variations)
#define SW_PRESSED (0)

// SW_THRES must be > 1 for correct debounce behaviour
#define SW_THRES (3)

// ----- Private variables -----
static tByte Sw_press_duration_G = 0;
static tByte Sw_blocked_G = 0;

/*-----*
FUNCTION: SWITCH_ON_OFF_Init()
Initialization function for the switch library.
*-----*/
void SWITCH_ON_OFF_Init(void)
{
    Sw_pin = 1;      // Use this pin for input
    Sw_pressed_G = 0; // Switch is initially OFF
    Sw_press_duration_G = 0;
    Sw_blocked_G = 0;
}

/*-----*
FUNCTION: SWITCH_ON_OFF_Update()
This is the main on-off switch function.
It should be scheduled every 50 - 500 ms.
*-----*/
void SWITCH_ON_OFF_Update(void)
{
    // If the switch is blocked, decrement the count and return
    // without checking the switch pin status.
    // This is done to give the user time to remove their finger
```

```

// from the switch - otherwise if they keep their finger on
// the switch for more than 0.4s the light will switch off again.

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;

        // Change switch state
        if (Sw_pressed_G == 1)
        {
            Sw_pressed_G = 0; // Switch state changed to OFF
        }
        else
        {
            Sw_pressed_G = 1; // Switch state changed to ON
        }

        // Allow no other changes for ~1 second
        Sw_blocked_G = 5;
        return;
    }

    // Switch pressed, but not yet for long enough
    return;
}

// Switch not pressed - reset the count
Sw_press_duration_G = 0;
}

/*-----*
 *----- END OF FILE -----*
 *-----*/

```

Listing 19.9 Part of an example demonstrating a software-based switch interface

Further reading

MULTI-STATE SWITCH

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you obtain the behaviour illustrated in Figure 19.13 from a single, push-button switch connected to the port pin of a microcontroller?

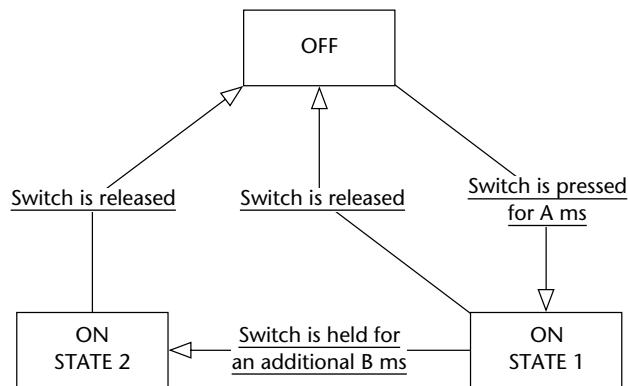


FIGURE 19.13 A three-state switch

Background

Solution

Although – as discussed in **ON-OFF SWITCH** [page 414] – sustained switch depressions can sometimes be problematic, they can also form the basis of more complex switch-based interfaces and can allow us to turn this two-state input device into a three-state (or more) input device.

For example, if we have a real-time clock, we may have just two buttons ('forward' and 'backward') to set the time. To avoid this process becoming unduly tedious, we

might decide that a brief depression of the 'Forward' button should slowly increment the displayed time, while a sustained depression (longer than, say, five seconds), should advance the display more rapidly.

To implement this type of behaviour, we might operate as follows:

- We keep track of the period of time over which the switch has been continually depressed.
- When the depression exceeds a threshold (call it Duration A), we treat this like a normal switch depression.
- When the depression exceeds a larger threshold (call it Duration B), we treat this as a sustained switch depression.

Note that further 'levels' can be added, but more than two (sometimes three) can be confusing to the user.

See the example that follows for complete implementation details.

Hardware resource implications

Creating a multi-state switch imposes very minor loads on CPU and memory resources.

Reliability and safety issues

The use of multi-state switches does not generally have reliability or safety implications.

Refer to **SWITCH INTERFACE (SOFTWARE)** [page 399] and **SWITCH INTERFACE (HARDWARE)** [page 410] for general discussions about the safety of switch interfaces.

Portability

This pattern may be adapted for use with other microcontrollers without difficulty.

Overall strengths and weaknesses

😊 A cost-effective way of improving the usability of many applications.

Related patterns and alternative solutions

See **ON-OFF SWITCH** [page 414].

Example: Counter

In this example, we demonstrate the key features of a multi-state switch by means of a counter that is incremented at a rate which depends on the switch-press duration (Listings 19.10 to 19.14).

The required hardware is illustrated in Figure 19.14.

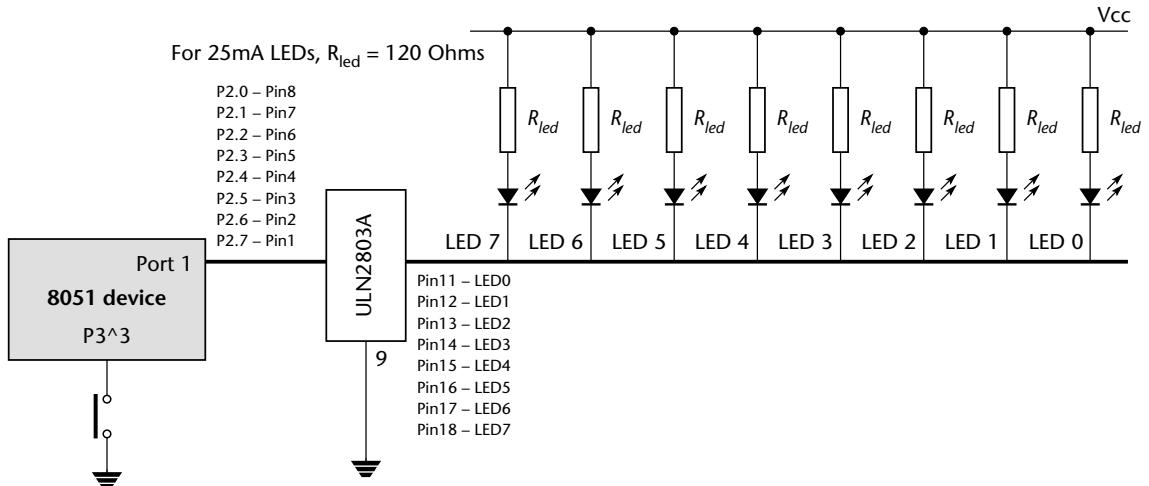


FIGURE 19.14 A collection of LEDs used to demonstrate multi-state switch

The key software files are given in Listings 18.6 to 18.11: a complete set of files for this example is included on the CD.

```
/* ----- *
   Port.H (v1.00)
----- */

'Port Header' (see Chapter 10) for the project MULTI_S
/* ----- */

// ----- Bargraph.C -----
// Connect LED from +5V (etc) to these pins, via appropriate resistor
// [see Chapter 7 for details]
// The 8 port pins may be distributed over several ports if required
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 -----
```

```

// Connect single push-button switch on this pin (to gnd)
// - debounced in software
sbit Sw_pin = P3^3; // Press this to start adjusting the time

/* -----
----- END OF FILE -----
----- */

```

Listing 19.10 Part of an example program demonstrating the use of multi-state switches

```

/* -----
----- Main.c (v1.00)

----- */

Demo program for MULTI-STATE SWITCH pattern.

See Chapter 19 for details.

Required linker options (see text for details):
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)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Set up the display
    BARGRAPH_Init();

    // Set up the switch pin
    SWITCH_MS_Init();

    // Add a 'SWITCH_MS_Update' task, every ~200 ms
    // - timings are in ticks (50 ms tick interval - see Sch 'init' function)
    SCH_Add_Task(SWITCH_MS_Update, 0, 4);

```

```
// Add a 'COUNTER_Update' task every ~1000 ms
SCH_Add_Task(COUNTER_Update, 0, 20);

SCH_Start();

while(1)
{
    SCH_Dispatch_Tasks();
}

/*
----- END OF FILE -----
*/

```

Listing 19.11 Part of an example program demonstrating the use of multi-state switches

```
/*
----- */

Bargraph.c (v1.00)

-----
Simple bargraph library. See Chapter 10.

/*
-----
#include "Main.h"
#include "Port.h"

#include "Bargraph.h"

// ----- Public variable declarations -----
// The data to be displayed
extern tByte Data_G;

// ----- Private constants -----
#define BARGRAPH_ON (1)
#define BARGRAPH_OFF (0)

// ----- Private variables -----
// These variables store the thresholds
// used to update the display
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;

    // Use a linear scale to display data
    // Remember: *9* possible output states
    // - do all calculations ONCE
    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()
 *
 * Update the bargraph display.
 * -----
 */
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);
}

/* -----
--- END OF FILE -----
- */
```

Listing 19.12 Part of an example program demonstrating the use of multi-state switches

```

/* -----
Counter.C (v1.00)

-----
Simple 'counter' function, to illustrate use of multi-state
switches.

- */

#include "Main.h"
#include "Counter.h"
#include "Bargraph.h"

// ----- Public variable definitions -----
tBargraph Data_G;

// ----- Public variable declarations -----
extern tByte Sw_status_G;
/* -----
COUNTER_Update()

Simple counter function (demo purposes).

- */
void COUNTER_Update(void)
{
    Data_G += Sw_status_G;
```

```

    if (Data_G > BARGRAPH_MAX)
    {
        Data_G = 0;
    }
    BARGRAPH_Update();
}

/* -----
   ----- END OF FILE
----- */

```

Listing 19.13 Part of an example program demonstrating the use of multi-state switches

```

/* -----
   ----- SWIT_D.C (v1.00)

----- 4-state switch interface code, with software debounce.

----- */

#include "Main.h"
#include "Port.h"

#include "Swit_D.h"

// ----- Public variables -----
tByte Sw_status_G; // The current switch status

// ----- Private constants -----
// SW_THRES must be > 1 for correct debounce behaviour
#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)

// Allows NO or NC switch to be used (or other wiring variations)
#define SW_PRESSED (0)

// ----- Private variables -----
static tByte Sw_press_duration_G = 0;

/* -----
   ----- SWITCH_MS_Init()
----- */

```

```
Initialization function for the switch library.

*-----
void SWITCH_MS_Init(void)
{
    Sw_pin = 1; // Use this pin for input
    Sw_status_G = 0; // Switch is initially OFF
    Sw_press_duration_G = 0;
}

*-----
SWITCH_MS_Update()

This is the main switch function. It should be scheduled every
50 - 500 ms.

Alters Sw_press_duration_G depending on duration of switch press.

*-----
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; // Switch has been pressed for a long time...
            return;
        }

        if (Sw_press_duration_G > (SW_THRES_X2))
        {
            Sw_status_G = 2; // Switch has been pressed for a medium time...
            return;
        }

        // SW_THRES must be > 1 for software debounce
        if (Sw_press_duration_G > SW_THRES)
        {
            Sw_status_G = 1; // Switch has been pressed for a short time...
            return;
        }
    }
}
```

```
// switch pressed, but not yet for long enough
Sw_status_G = 0;
return;
}

// Switch not pressed - reset the count
Sw_press_duration_G = 0;
Sw_status_G = 0; // Switch not pressed...
}

/*-----*
----- END OF FILE -----
*-----*/
```

Listing 19.14 Part of an example program demonstrating the use of multi-state switches

Further reading

chapter **20**

Keypad interfaces

Introduction

Keypads are a common component in embedded applications.

In this chapter, we consider how you can create reliable keypad-based user interfaces.

KEYPAD INTERFACE

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you connect a small keypad, similar to that illustrated in Figure 20.1, to your application?

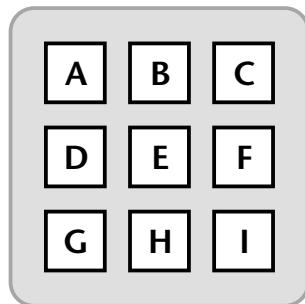


FIGURE 20.1 An example of a three-row x three-column keypad

Background

See Chapter 19 for background information on the reading of single switches.

Solution

Basics

We are concerned here with keypads made up of a matrix of switches, in an arrangement similar to that illustrated in Figure 20.2.

Some key points to note are as follows:

- The matrix arrangement is used to save port pins. If we have R rows and C columns of keys, we need $R + C$ pins if we use a matrix arrangement and $R \times C$ pins if we use individual switches. If you need six or more keys, then the matrix arrangement requires fewer pins.

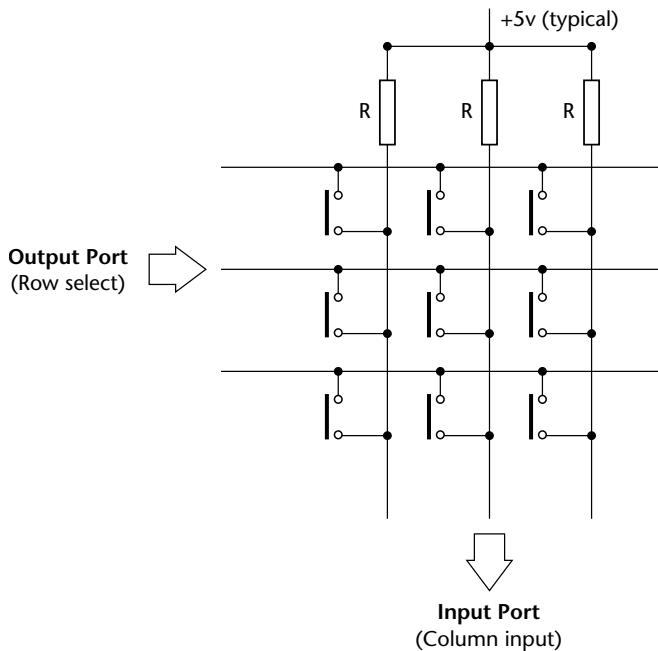


FIGURE 20.2 A schematic of a keypad interface

[Note: The pull-up resistors may be omitted, as usual, if the port has internal pull-ups.]

Many keypads have 12 keys ('0'–'9' plus two function keys – typically '#' and '*'). Using a matrix arrangement, this requires seven port pins.

- The keys may bounce, when pressed and released.
- The duration of the key press will generally be at least 500 ms.
- The keys will not generally be allowed to 'auto repeat': this can be very confusing for users.
- We may wish the user to be able to press one or more 'function keys' in combination with other keys.

Keypad scanning

At the heart of any keypad code is a scanning function: this will typically go through each column in turn and identify if any key in that column has been pressed.

Consider, for example, the keypad shown in Figure 20.3.

In Figure 20.3, the numbers adjacent to the rows and column indicate the port pins to which the keypad should be connected. Pins 0, 1 and 2 (the columns) will be referred to here as the output pins: these are written to during the scanning process. Pins 3, 4, 5 and 6 will be referred to as the input pins: these are read during the scanning process.

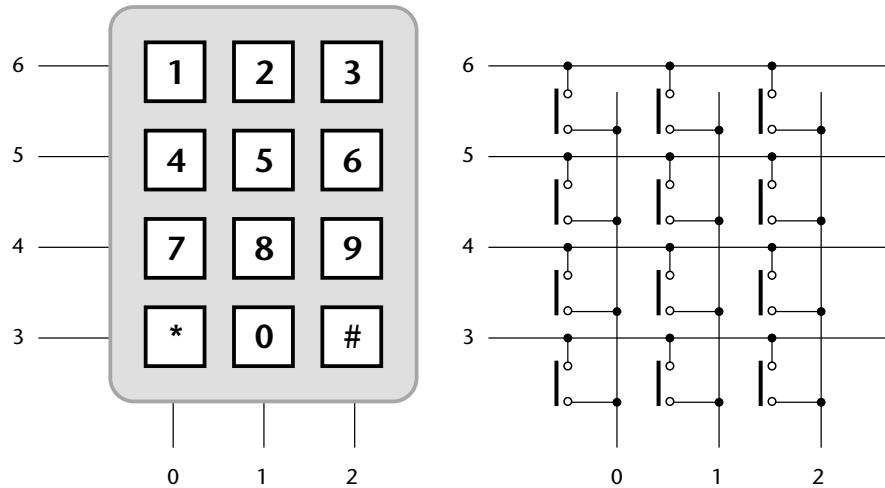


FIGURE 20.3 Typical keypad connections

[Note: The numbers adjacent to the keypad rows and columns represent the pin numbers on the port to which the keypad is connected. Note that no pull-up resistors will generally be needed: refer to Figure 20.2 for details.]

Suppose we wish to see if the key '1' is pressed. We can proceed as follows:

- We set the corresponding output pin (in this case, Pin 6) to a Logic 0 value and the remaining output pins to a Logic 1 value.
- We read the required input pin (in this case, Pin 0).

If this pin is a Logic 1, then the key '1' is not pressed: the Logic 1 value is, instead, obtained via the pull-up resistor on the port pin.

If this pin is at Logic 0, then the key is being pressed (subject to debounce considerations): the Logic 0 voltage reading results from the Logic 0 voltage output on Pin 6.

- We must repeat the reading (say) 200 ms later, to allow for switch bounce.

We need to repeat this process for every key. Listing 20.1 illustrates one way of performing this scanning for the whole keypad.

```
#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; // Scanning column 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; // Scanning column 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; // Scanning column 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)  
    {  
        // No key pressed (or just a function key)  
        Old_Key = KEYPAD_NO_NEW_DATA;  
        Last_valid_key_G = KEYPAD_NO_NEW_DATA;  
  
        return 0; // No new data  
    }  
  
    // A key has been pressed: debounce by checking twice  
    if (Key == Old_Key)  
    {  
        // A valid (debounced) key press has been detected  
  
        // Must be a new key to be valid - no 'auto repeat'  
        if (Key != Last_valid_key_G)  
        {
```

```

    // New key!
    *pKey = Key;
    Last_valid_key_G = Key;

    // Is the function key pressed too?
    if (Fn_key)
    {
        // Function key *is* pressed with another key
        *pFuncKey = Fn_key;
    }
    else
    {
        *pFuncKey = (char) 0x00;
    }

    return 1;
}
}

// No new data
Old_Key = Key;
return 0;
}

```

Listing 20.1 An example of code for keypad scanning

Function keys

Note that more than one key may be pressed at the same time. The ‘function keys’ (#’ and ‘*’) in Listing 20.1 illustrate how we can make use of multiple key depressions.

The main code example presented with this pattern illustrate the use of function keys.

Buffer arrangements

In most scheduled keypad routines, it is useful to have a small buffer, so that key presses are not lost if the system is unable to process them immediately.

Numerous buffer arrangements are possible: the main code example presented with this pattern illustrates one possibility.

Hardware resource implications

While it does not require on-chip facilities (such as timers etc.), the keypad scanning process imposes both a CPU and memory load.

Reliability and safety issues

Keypad scanning is a software-based technique, closely related to **SWITCH INTERFACE (SOFTWARE)** [page 399]. In a hostile environment, use of a keypad may provide a less reliable solution than use of a number of individual switches with a hardware interface (see **SWITCH INTERFACE (HARDWARE)** [page 410]).

However, particularly where a large keypad – such as a QWERTY keypad with around 30 keys – is required, use of separate switches may be impractical. The safest thing to do in such circumstances may be to use a shared-clock scheduler (see Part F), and have a second microcontroller link to the keypad. This ‘keypad microcontroller’ should then have a separate power supply, and should be opto-isolated from the main system board.

Portability

This code is highly portable and this approach to keypad scanning is very widely used.

Overall strengths and weaknesses

- ☺ **Multiplexed keypads are easy to use and inexpensive.**
- ☹ Because the pattern is software based, minimal protection against ESD and malicious damage (for example) is provided: use of a separate, isolated, keypad processor may be necessary if the reliability of the main processor is essential.
- ☹ In harsh environments, it may be safer to avoid multiplexed keypads and use individual switches.

Related patterns and alternative solutions

See **SWITCH INTERFACE (SOFTWARE)** [page 399] and **SWITCH INTERFACE (HARDWARE)** [page 410].

Example: Keypad library with buffer and fn key support

We present here a complete keypad library for the 8051 family. The library has support for function keys (two) and has a buffer facility. The library is intended to work with keypads as shown in Figure 20.3, but is easily adapted to different keypad layouts and sizes.

The demonstration program runs on an Infineon c515c microcontroller. However, none of the keypad code is 515 specific: using appropriate versions of the scheduler and the PC link libraries (included on the CD) it can be used with any 8051 family member.

Figure 20.4 shows a typical program output.

The key library files are given in Listings 20.2 to 20.5. Refer to the CD for the complete set of files for this example.

```
Serial interface initialized.  
Keypad test code - READY  
1234567890  
*1  
*1  
#2  
#5  
0982374122
```

FIGURE 20.4 Typical output from the keypad input program given in this section

```
/*-----*  
Port.H (v1.00)  
-----  
'Port Header' (see Chapter 10) for the project Key_232  
-*-----*/  
// ----- Sch51.C -----  
// Comment this line out if error reporting is NOT required  
// #define SCH_REPORT_ERRORS  
#ifdef SCH_REPORT_ERRORS  
// The port on which error codes will be displayed  
// ONLY USED IF ERRORS ARE REPORTED  
#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_0.C -----  
// Pins 3.0 and 3.1 used for RS-232 interface
```

```
/* -----
   ----- END OF FILE -----
   ----- */
```

Listing 20.2 Part of the software for a keypad interface

```
/* -----
   ----- Main.c (v1.00)
   ----- */

Test program for keypad library, based on c515c
Sends output (to PC) over serial (RS232) link

Linker options:
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 "Lnk_0_B.h"
#include "Keypad.h"
#include "Keyp_232.h"

/* ..... */
/* .....
```

```
void main(void)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Set baud rate to 9600, using internal baud rate generator
    PC_LINK_Init_Internal(9600);

    // Prepare the keypad
    KEYPAD_Init();

    // Prepare the Keypad -> RS232 library
    Keypad_RS232_Init();

    // We have to schedule this task (~100x a second is enough here)
    // - this writes data to PC
    //
    // TIMING IS IN TICKS (1 ms tick interval)
    SCH_Add_Task(PC_LINK_Update, 10, 10);
```

```

    // Read the keypad every ~50 ms
    SCH_Add_Task(Keypad_RS232_Update, 0, 50);

    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

/*-----*
 *----- END OF FILE -----*
 */
```

Listing 20.3 Part of the software for a keypad interface

```

/*-----*
 *----- Keypad.C (v1.00)
 *-----

Simple keypad library, for a 3-column x 4-row keypad.

Key arrangement is: -----
      1  2  3
      4  5  6
      7  8  9
      *  0  #

-----

Supports two function keys ('*' and '#').

See Chapter 19 for details.

/*-----*
```

```

#include "Main.h"
#include "Port.h"

#include "Keypad.h"

// ----- Private function prototypes -----
bit KEYPAD_Scan(char* const, char* const);

// ----- Private constants -----
#define KEYPAD_RECV_BUFFER_LENGTH 6
```

```

// Any valid character will do - must not match anything on keypad
#define KEYPAD_NO_NEW_DATA (char) '-'

// ----- Private variables -----

static char KEYPAD_recv_buffer[KEYPAD_RECV_BUFFER_LENGTH+1][2];

static tByte KEYPAD_in_read_index;      // Data in buffer that has been read
static tByte KEYPAD_in_waiting_index; // Data in buffer not yet read

static char Last_valid_key_G = KEYPAD_NO_NEW_DATA;

/*-----*/
KEYPAD_Init()

Init the keypad.

/*-----*/
void KEYPAD_Init(void)
{
    KEYPAD_in_read_index = 0;
    KEYPAD_in_waiting_index = 0;
}

/*-----*/
KEYPAD_Update()

The main 'update' function for the keypad library.

Must schedule this function (approx every 50 - 200 ms).

/*-----*/
void KEYPAD_Update(void)
{
    char Key, FnKey;

    // Scan keypad here...
    if (KEYPAD_Scan(&Key, &FnKey) == 0)
    {
        // No new key data - just return
        return;
    }

    // Want to read into index 0, if old data has been read
    // (simple ~circular buffer)
    if (KEYPAD_in_waiting_index == KEYPAD_in_read_index)
    {
        KEYPAD_in_waiting_index = 0;
        KEYPAD_in_read_index = 0;
    }
}

```

```

    // Load keypad data into buffer
    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)
    {
        // Increment without overflowing buffer
        KEYPAD_in_waiting_index++;
    }
}

/*-----*
 * KEYPAD_Get_Char_From_Buffer()
 *
 * The Update function copies data into the keypad buffer.
 * This extracts data from the buffer.
 *-----*/
bit KEYPAD_Get_Data_From_Buffer(char* const pKey, char* const pFuncKey)
{
    // If there is new data in the buffer
    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()
 *-----*/

```

This function is called from scheduled keypad function.

Must be edited as required to match your key labels.

Includes two 'function keys' which may be used simultaneously with keys from any other column.

Adapt as required!

```
-----*/
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; // Scanning column 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; // Scanning column 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; // Scanning column 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)
    {
        // No key pressed (or just a function key)
        Old_Key = KEYPAD_NO_NEW_DATA;
        Last_valid_key_G = KEYPAD_NO_NEW_DATA;

        return 0; // No new data
    }

    // A key has been pressed: debounce by checking twice
    if (Key == Old_Key)
```

```

    {
        // A valid (debounced) key press has been detected

        // Must be a new key to be valid - no 'auto repeat'
        if (Key != Last_valid_key_G)
        {
            // New key!
            *pKey = Key;
            Last_valid_key_G = Key;

            // Is the function key pressed too?
            if (Fn_key)
            {
                // Function key *is* pressed with another key
                *pFuncKey = Fn_key;
            }
            else
            {
                *pFuncKey = (char) 0x00;
            }

            return 1;
        }
    }

    // No new data
    Old_Key = Key;
    return 0;
}

/*-----*
----- END OF FILE -----
-*-----*/

```

Listing 20.4 Part of the software for a keypad interface

```

/*-----*
Keyp_232.C (v1.00)

-----
Simple demonstration function for transferring keypad inputs to
PC via serial (RS232) link.

-*-----*/

```

```
#include "Main.h"
#include "Keypad.h"
#include "Keyp_232.h"
#include "Lnk_0.h"

// ----- Private variables -----
tByte Count_G;

/* -----
Keypad_RS232_Init()

Init function for simple library displaying keypad inputs
over serial link.

*/
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()

Function for displaying keypad inputs over serial link.

*/
void Keypad_RS232_Update(void)
{
    char Key, FnKey;

    // Update the keypad buffer
    KEYPAD_Update();

    // Is there any new data in the keypad buffer?
    if (KEYPAD_Get_Data_From_Buffer(&Key, &FnKey) == 0)
    {
        // No new data.
        return;
    }

    // Function key has been pressed (with another key)
    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
{
// An ordinary key (no function key) has been pressed
PC_LINK_Write_Char_To_Buffer(Key);

if (++Count_G == 10)
{
PC_LINK_Write_Char_To_Buffer('\n');
Count_G = 0;
}
}
}

/*-----*
----- END OF FILE -----
*-----*/
```

Listing 20.5 Part of the software for a keypad interface

Example: Keypad and LCD

See Chapter 22 for a complete example of a keypad and LCD interface design.

Further reading

chapter 21

Multiplexed LED displays

Introduction

Many embedded applications contain user interfaces assembled from multi-segment LED displays.

In this chapter, we consider how such displays may be interfaced to the 8051 family of microcontrollers.

MX LED DISPLAY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user-interface for your application.

Problem

How do you display information on one or more multi-segment LED displays?

Background

We consider some essential background material in this section.

What is a multi-segment LED?

Multiple LEDs are often arranged as multi-segment displays: combinations of eight segments (see Figure 21.1) and similar seven-segment displays (without a decimal point) are particularly common.

Such displays are arranged either as ‘common cathode’ or ‘common anode’ packages: the connection of the LEDs within each package is illustrated in Figure 21.2.

In either case, in addition to the common pin, we must provide appropriate signals to each of the LEDs in order to generate the digits we require (see ‘Solution’).

The required current per segment varies from about 2 mA (very small displays) to about 60 mA (very large displays, 100mm or more).

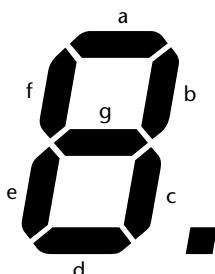


FIGURE 21.1 The segments of what is often referred to as a seven-segment display

[Note: that, as here, a decimal point will frequently also be included, so that this ‘seven-segment’ display actually has eight segments.]

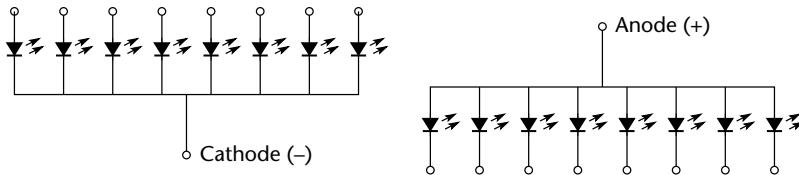


FIGURE 21.2 The internal arrangement of common cathode and common anode LED displays

[Note: that there are eight segments because we assume the presence of a decimal point.]

Basic hardware requirements (driving a single digit)

For reasons that we discussed in **IC BUFFER** [page 118], we cannot generally connect a multi-segment LED directly to a microcontroller port as illustrated in Figure 21.3. As you will recall, this is because the port cannot safely handle the required current (typically around 80 mA per digit).

In most cases, we require some form of buffer or driver IC between the port and the MS LED.

For small displays (around 9 mA per segment), the simple 240 and 241 buffers (see **IC BUFFER** [page 118]) are a cost-effective solution. Figure 21.4 shows a 74x241 (non-inverting buffer) used to drive a single, eight-segment LED display. Note that a 74x240 (inverting) buffer can be used as an alternative.

In most circumstances (such as the multiplexed displays we will focus on here), the basic 240 / 241 buffers have insufficient current capacity and an IC driver chip will generally be used. For example, Figure 21.5 shows a single common cathode LED digit connected to a single port via another octal buffer: the UDN2585A. As we discussed

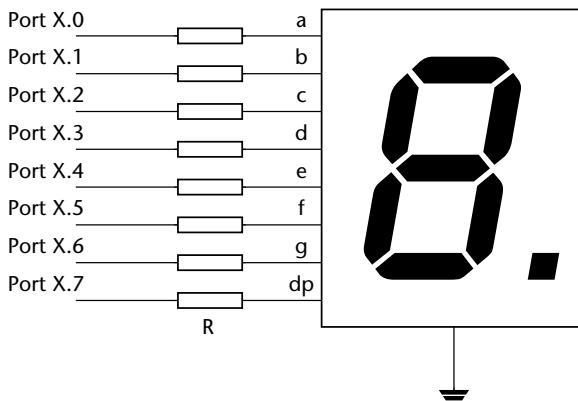
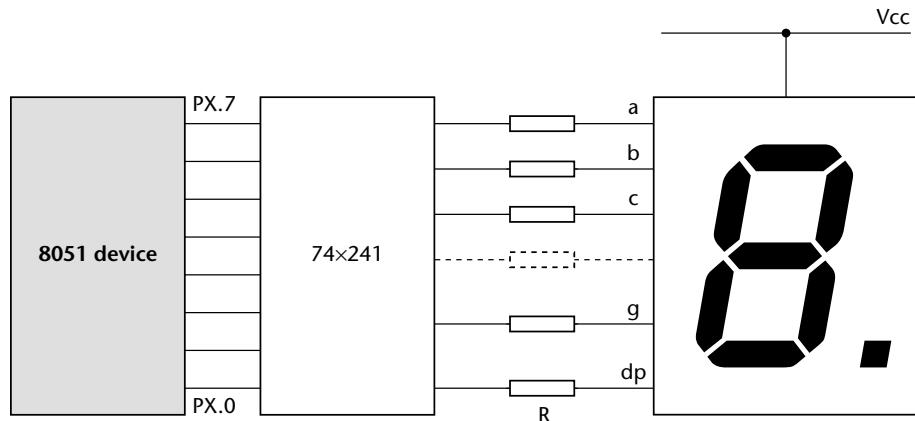


FIGURE 21.3 Attempting to drive a common anode display directly from a microcontroller port

[Note: in most cases, this will not work.]



$$R = \frac{V_{cc} - V_{diode}}{I_{diode}}$$

Note: All resistor values may not be the same
(the decimal point may require a different value)

FIGURE 21.4 Using a 74x241 to drive a multi-segment LED display

[Note: that, in some displays, a different resistor value may be required to drive the decimal point: check your data sheet. Note also that, as we discussed in IC BUFFER [page 118], we do not require pull-up resistors at the buffer output if CMOS logic is used.]

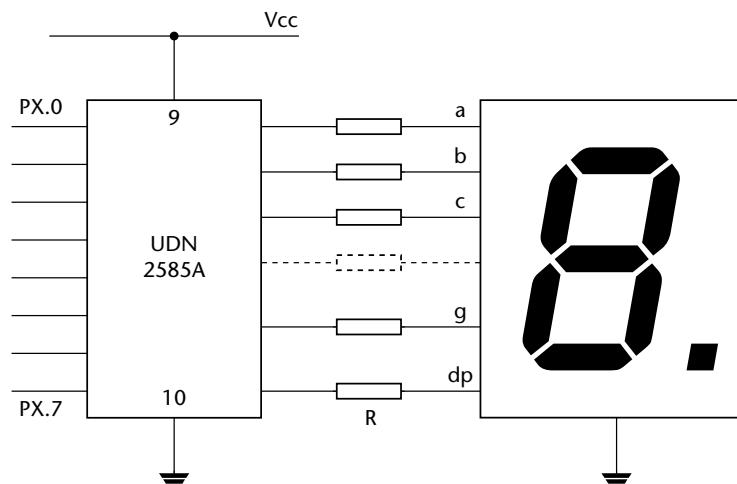


FIGURE 21.5 Using a UDN 2585A to drive an LED display

[Note: that this is an inverting (current source) buffer. Logic 0 on the input line will light the corresponding LED segment.]

in **IC DRIVER** [page 134], each of the (eight) channels in this device can simultaneously source up to 120 mA of current (at up to 25V): this is enough, for example, for even the largest of LED displays.

Here, the output of the 2585 in response to a Logic 0 input will be approximately V_{cc}. As a result, the resistor values are again calculated as shown in Figure 21.4.

Hardware for multiple digits

As discussed in ‘Background’, driving a single multi-segment display is straightforward. Suppose, however, that we need to drive four digits, perhaps for a simple digital clock. To do so using the approach discussed in ‘Background’ would require four spare ports. In many applications, you will not have four ports available. In addition, to use four ports often requires a separate driver (buffer) circuit and resistor pack for each digit: this can significantly increase the system cost.

The most common solution, discussed here, is to multiplex the displays. This means that (in this example) we drive each display for only a quarter of the time: as long as we cycle around all of the displays at between 20 and 50 Hz, the user will not generally notice that the displays are not being simultaneously driven. Even with a simple implementation, this basic approach allows us to drive up to eight LEDs (each with decimal points) using only two ports. More commonly, this allows us to drive four LED digits using 12 port pins. As in many 8051 applications (even those using external memory) Port 1 and most of Port 3 are available, this approach can be used in many applications.

A simple circuit suitable for creating a cost-effective multiplexed display is shown in Figure 21.6. Here, the information required to drive each digit is supplied on Port X, while the selection of the digits is controlled by Port Y.

Note that Port Y must control (sink) the current coming from eight individual LEDs: this may be up to around 140 mA, even for a collection of small displays and

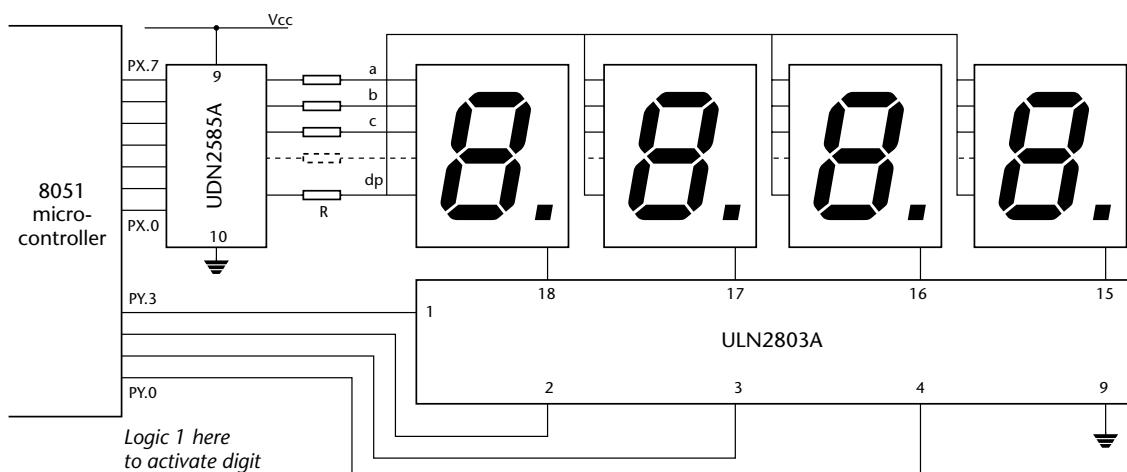


FIGURE 21.6 Multiplexing four (common cathode) LED modules

can approach 500 mA for large (100 mm) displays. As a result, we cannot directly drive the displays from a port, but must use, for example, a ULN 2003 or 2803 as a (sink) buffer. Note that you can replace the 2003 / 2803 (or equivalent) with four NPN transistors (e.g. 2N2222), but this is seldom cost-effective.

You will also need a buffer on Port X to source sufficient current: the UDN 2585A is, again, a good choice here. Use of such a source buffer is often essential, since – to achieve a bright display – we usually supply up to four times the normal display current, for a quarter of the time (thus keeping the average current at the required value). Your microcontroller will have a greatly reduced life if you try to provide this current from the naked chip.

Again, the values of R in Figure 21.6 are calculated as shown in Figure 21.4. In this case, it can be helpful to use a ‘resistor pack’ to implement the circuit: such packs contain seven or eight resistors, in a standard (dual in line: DIL or similar) package.

Solution

We consider here both the software codes required to display digits on the display and the refresh rates needed for flicker-free operation.

Basic software

The software to control the LED digits in arrangements like those shown in Figures 21.4 and 21.5 is easy to write. If, for example, we connect the display to Port 3, we can control it by simply writing:

```
P3 = code;
```

Here, ‘code’ is a numerical representation of the required segment activations. The data required to display the digits 0–9, with or without a following decimal point, are given in Listing 21.1.

```
/*-----*-
Connections
DP  G   F   E   D   C   B   A   =   LED display pins
|   |   |   |   |   |   |   |
x.7 x.6 x.5 x.4 x.3 x.2 x.1 x.0   =   Port pins

LED codes (NB - positive logic assumed here)
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

```

To display decimal point, add 10 (decimal) to the above values

```

-----*/  

// Lookup table - stored in code area (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};  

// -----

```

Listing 21.1 Codes that may be used to display data on multi-segment LED displays

Controlling more than one digit per port (Software issues)

We need to update all the displays at a rate of approximately 50 Hz, if we can: if necessary, rates as low as 20 Hz will do. If we have N LED modules, each will be active for approximately 1/N of the time. Note that, if you choose to drive the displays at high current values, the timing needs to be reasonably accurate, otherwise you may reduce the life of the display and driver components by exposing them to excessively high currents for sustained periods.

Overall, for a typical four-module display, we aim to update one module at least every 5 milliseconds. This is easy to achieve using a suitable scheduler, as we demonstrate in the examples that follow.

Hardware resource implications

All the examples here require the use of a scheduler. The main resource implication is that, to update (say) four digits, you need a scheduler with a tick interval of around 5 ms. This is not usually a limiting factor. Overall, the LED update code will consume only around 1% of the CPU time in a typical application, since – although frequent – it is a simple and fast operation.

The memory requirements are minimal.

A substantial number of port pins are required.

Reliability and safety implications

LEDs are not visible in bright light and, as such, must be used with care if they are displaying safety-related information.

For multiplexed displays, you must ensure that the application (or scheduler) cannot become locked: if it does, the display will very quickly be destroyed by high current values. Thus, if a poorly designed task blocks the scheduler, even for a few seconds, and delays a display update (usually scheduled around 20 to 50 Hz), you will probably destroy the display. Whatever happens, you need to ensure that such an event will not impact on the general operation of the microcontroller and, therefore, completely destroy the application.

Portability

These techniques are easily ported across members of the 8051 family and to any other processor family. No features unique to the 8051 are employed.

Overall strengths and weaknesses

- ☺ Multi-segment LED displays are reliable and cheap.
- ☺ The techniques discussed in this pattern are easy to apply and highly portable.
- ☹ A typical interface requires a substantial number of port pins: up to 12 pins for a multiplexed 4-digit display.
- ☹ Displays are not visible in bright sunlight. Provide an additional audible warning if necessary.
- ☹ Multiplexed displays can suffer from reliability problems if the scheduling is not handled correctly.

Related patterns and alternative solutions

There are numerous ‘intelligent’ LED drivers, such as the Maxim 7219, that can deal with the task of driving and multiplexing LED displays for you. These usually have a serial interface and, as a result, require few port pins. This can be a useful solution if port availability is limited.

Another way of reducing port requirements is to use a version of the 74x247, BCD-to-7-segment decoder IC as an interface to your displays, as illustrated in Figure 21.7. The main problem with this approach is that the 74x247 is no longer widely available. If you can find a good supply and are assured of future supplies, this device is easy to use: consult the data sheet for details.

Another alternative, if lack of ports is a problem, is to use an additional Small 8051 in your design. This chip will then be responsible for the display multiplexing and updates: data can be easily transferred to such a device using a serial interface and a shared-clock scheduler (Figure 21.8). Shared-clock schedulers are discussed in Part F.

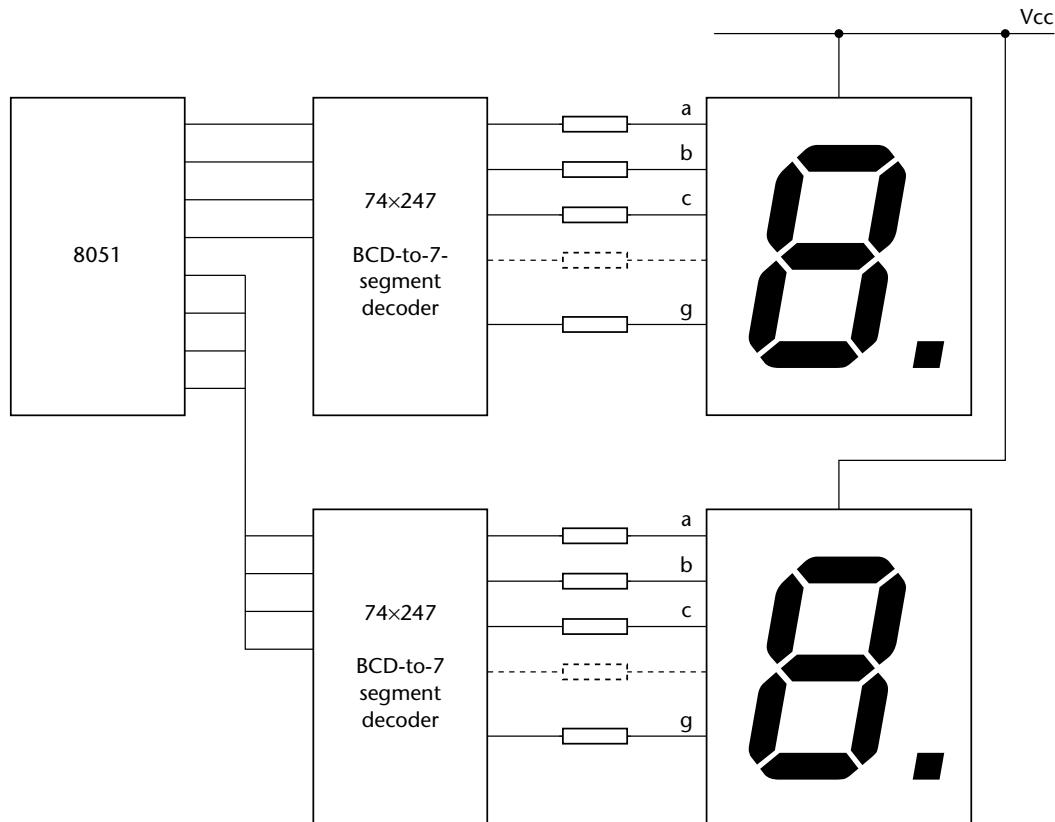


FIGURE 21.7 Driving two multi-segment LED displays from a single port using 74x247 BCD decoders

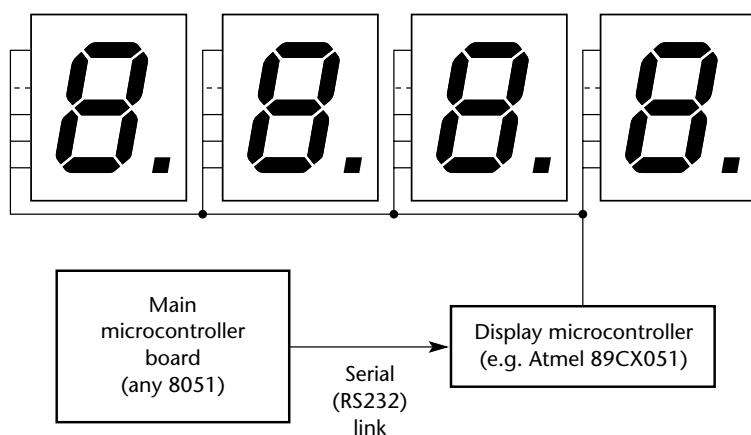


FIGURE 21.8 Using a shared-clock scheduler and two 8051 microcontrollers to provide rapid updates of an LED display

Example: Displaying elapsed time on a multiplexed (4-digit) LED display

A common use for LED displays is to display the time. This simple example demonstrates the display of elapsed time on four multiplexed LEDs. The code is written for a standard 8051 (8052) device.

The required hardware is illustrated in Figure 21.6.

The key source files for this example follow (Listings 21.2 to 21.5): all the files required in this example are included on the CD.

```
/*-----*/
Port.H (v1.00)

-----
'Port Header' (see Chapter 10) for the project LED_TIME

/*-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1
#endif

// ----- LED_MX4.C -----
// LED connection requires 12 port pins
#define LED_DATA_PORT (P2)

/* Connections to LED_DATA_PORT - See Figure 21.6 for details

      DP   G   F   E   D   C   B   A   =   LED display pins
      |   |   |   |   |   |   |   |   |
x.7  x.6  x.5  x.4  x.3  x.2  x.1  x.0   =   Port pins
x.7 == LED_DATA_PORT^7, etc

LED codes (NB - positive logic assumed here)

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 = abcfg   => 01101111 = 0x6F

To display decimal point, add 10 (decimal) to the above values */

// Any combination of (4) pins on any ports may be used here
sbit LED_DIGIT_0 = P3^3;
sbit LED_DIGIT_1 = P3^4;
sbit LED_DIGIT_2 = P3^5;
sbit LED_DIGIT_3 = P3^6;

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 21.2 Part of an example that displays elapsed time on four multiplexed LED displays

```

/*-----*
Main.c (v1.00)

-----*/

Demonstration program for:

Program driving 4 multiplexed multi-segment LED displays
- displays elapsed time.

Required linker options (see Chapter 13 for details):
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)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Add the 'Time Update' task (once per second)
    // - timings are in ticks (1 ms tick interval)
    // (Max interval / delay is 65535 ticks)
    SCH_Add_Task(CLOCK_LED_Time_Update,100,10);

    // Add the 'Display Update' task (once per second)
    // Need to update a 4-segment display every 3 ms (approx)
    // Need to update a 2-segment display every 6 ms (approx)
    SCH_Add_Task(LED_MX4_Display_Update,0,3);

    // Start the scheduler
    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

/*-----*
 *----- END OF FILE -----*
 *-----*/

```

Listing 21.3 Part of an example that displays elapsed time on four multiplexed LED displays

```

/*-----*
 *----- Cloc_Mx4.C (v1.00)
 *-----*/

Simple library function for keeping track of elapsed time

This version for (Mx) LED display

/*-----*/
#include "Main.h"
#include "Port.h"

#include "Cloc_Mx4.h"

// ----- Public variable declarations -----

```

```
extern tByte LED_Mx4_Data_G[4];
extern tByte code LED_Table_G[20];

// ----- Private variable definitions-----

// Time variables
static tByte Hou_G, Min_G, Sec_G;

/*-----*/
CLOCK_LED_Time_Update()
Updates the global time variables.

*** Must be scheduled once per second ***

*/
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)
    {
        // Need to update the minutes data
        // (both digits)
        LED_Mx4_Data_G[1] = LED_Table_G[Min_G / 10];
        LED_Mx4_Data_G[0] = LED_Table_G[Min_G % 10];
    }

    // We don't display seconds in this version.
    // We simply use the seconds data to turn on and off the decimal
```

```

    // point between hours and minutes
    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)
    {
        // Need to update the 'tens of hours' data
        LED_Mx4_Data_G[3] = LED_Table_G[Hou_G / 10];
    }
}

/* -----
----- END OF FILE -----
----- */
```

Listing 21.4 Part of an example that displays elapsed time on four multiplexed LED displays

```

/* -----
----- LED_Mx4.C (v1.00)

-----
Simple library function for displaying data on four multiplexed,
eight-segment LED displays

----- */

#include "Main.h"
#include "Port.h"

#include "LED_Mx4.h"

// ----- Public variable definitions -----
// Lookup table - stored in code area
// See Port.H for connections and code details

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};  
  
// Global data formatted for display (initially 0,0,0,0)  
tByte LED_Mx4_Data_G[4] = {0x3F,0x3F,0x3F,0x3F};  
  
// ----- Private variable definitions-----  
  
static tByte Digit_G;  
/*-----*/  
  
LED_MX4_Display_Update()  
Updates (four) multiplexed 8-segment LED displays.  
Usually aim to scheduled at around 3 ms intervals: typically  
around a 1% CPU load on basic 8051.  
/*-----*/  
void LED_MX4_Display_Update(void)  
{  
    // Increment the digit to be displayed  
    if (++Digit_G == LED_NUM_DIGITS)  
    {  
        Digit_G = 0;  
    }  
  
    // Allows any pins to be used  
    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];  
}  
  
/*-----*  
----- END OF FILE -----  
-----*/
```

Listing 21.5 Part of an example that displays elapsed time on four multiplexed LED displays

Further reading

chapter **22**

Controlling LCD panels

Introduction

We considered the creation of LED-based user interfaces in Chapter 21. Here we are concerned with the use of liquid crystal displays (LCDs) in such interfaces.

Unlike LEDs, LCDs are based on passive display technology: this means that LCDs control the passage of light rather than emitting light. This fact directly contributes to the low power consumption of these devices: large (5V) panels require up to 5 mA: a total power consumption of up to 25 mW, excluding any backlight. Small panels consume around half this power. Since a *single* LED has a very similar power consumption, use of LCD displays in battery-powered embedded systems is particularly popular.

While various types of LCD display are available, they can be divided into two basic groups: graphics displays and text displays. Notebook (and increasingly desktop) PCs use sophisticated graphics displays, made up of many thousands of individual ‘picture elements’ (pixels). Such displays are expensive in their own right and generally require large amounts of memory (typically several megabytes) and powerful processors for efficient operation. As we have seen, the type of embedded devices we are concerned with in this book do not usually justify this level of expense. Instead, we will be concerned here with small displays intended primarily to display text.

Various types of LCD-based character displays are available. These are typically arranged as one, two or four lines of between 16 and 40 characters. Inevitably, the larger displays are more expensive and consume more power. Each LCD character is usually a 5x8 matrix of dots: less commonly, a 5x11 matrix is also used: note that, in each case, the characters themselves are 5x7 and 5x10 pixels in size, with the bottom line being reserved for the cursor. To generate characters on such displays would tend to consume a large percentage of the available CPU time (and most of the ports or address space) on most embedded processors. As a result, most LCD panels include an on-board controller to deal with this: this is generally a variant on the Hitachi

HD44780. Displays based on this popular controller all have very similar hardware interfaces and will display the same mixture of English (or Japanese) characters.

We focus on LCD panels based on this ‘standard’ controller in the pattern **LCD CHARACTER PANEL** presented in this chapter.

Please note that much of the background material presented here is adapted from the Hitachi HD44780 data sheet.

LCD CHARACTER PANEL

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you connect an LCD-based ‘character panel’ display to your embedded processor and control it efficiently using a high-level programming language?

Background

As outlined at the start of this chapter, we are concerned here with LCD character panels based on the HD44780 microcontroller.

Key HD44780 components

The HD44780 contains its own reset circuitry, memory and so forth: it contains several important components (Figure 22.1), which may be accessed and controlled from an attached microcontroller or microprocessor. We will discuss some of the key components in the sections that follow.

As shown in Figure 22.1, the interface between the microcontroller and the HD44780 consists of five sets of signals. A summary of each of these signals is given in Table 22.1.

DD RAM

At the heart of the HD44780 is the DD RAM: the ‘Display Data’ RAM. This stores display data represented in 8-bit character codes. Its capacity is 80 characters; as a result the maximum display sizes are 20 charactersx4 lines or 40 charactersx2 lines.

Use of the HD44780 involves transferring data (via the 4-bit or 8-bit data bus) into the DD RAM. The HD44780 will then refresh the display as required using these data.

Characters that can be stored and displayed via DD RAM include most of the core (displayable) characters from the ASCII table, with only minor changes (Table 22.2): this means that, in general, you can simply send character data to the display and obtain the expected outputs.

Note that *none* of these characters uses the bottom line of the display (the cursor line): as a result, low-case characters with ‘tails’ (g, j, p, q, y) will appear higher than normal on the display.

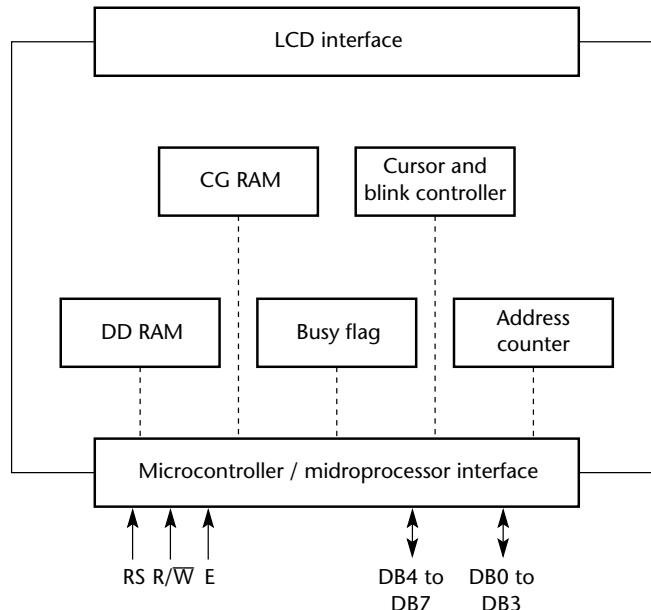


FIGURE 22.1 Key components in the Hitachi HD44780.

TABLE 22.1 The HD44780 interface

Signal	Number of lines	Input and / or output?	Function
RS	1	I	Selects instruction register (0) or data register (1)
R /W	1	I	Selects read (1) or write (0)
E	1	I	Starts data read/write
DB4 to DB7	4	I/O	Used for data transfer between the 8051 and the HD44780. DB7 can be used as a busy flag (not used in the code samples here)
DB0 to DB3	4	I/O	Used for data transfer and receive between the 8051 and the HD44780. These pins are not used during 4-bit operation (not used in the code samples here)

TABLE 22.2 The basic character set in the HD44780

	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	V	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	{		}	→	←

[Most of this follows the ASCII code: however, note the Yen symbol (Character 0x5C) in place of '\', and that 0x7E and 0x7F are left and right arrows, respectively. UK users should also note that there is no '£' symbol (character 0x23 is '#').]

While the character codes are straightforward, knowing which address to write the data to is slightly less straightforward. There are a number of different configurations of display available (including 16 characters x1 line, 16x2, 20x2, 20x4, 24x1, 24x2, 40x1, 40x2). The memory locations for each of the segments on these displays are shown in Table 22.3.

TABLE 22.3 Memory locations for the start of each line in various possible HD44780 LCD displays

#characters	#lines	Top line	Second line	Third line	Fourth line
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 – xx67
24	1	0x00 – 0x17	-	-	-
24	2	0x00 – 0x17	0x40 – 0x58	-	-
40 (20)	1 (2)	0x00 – 0x13	0x40 – 0x53	-	-
40	2	0x00 – 0x27	0x40 – 0x67	-	-

CG RAM

The CG RAM is the ‘Character Generator’ RAM: this can be used to allow the creation and display of user-defined characters. This facility is helpful if you wish to display characters outside the ‘standard’ range, such as a ‘£’ sign.

We demonstrate how to use CG RAM in an example that follows.

Registers

The HD44780 has two 8-bit registers, an instruction register (IR) and a data register (DR):

- The IR stores instruction codes, such as display clear and cursor shift and address information for DDRAM and character generator CGRAM.
- The DR temporarily stores data to be written into DDRAM or CGRAM. Data written into the DR from the 8051 are automatically written into DDRAM or CGRAM by an internal operation.

Busy flag (BF)

When the busy flag is 1, the HD44780 is performing an internal operation and further instructions or data will not be accepted. When RS = 0 and R/W = 1 (Table 22.4), the busy flag is output to DB7.

TABLE 22.4 Register selection

RS	R/W	Operation
0	0	IR write as an internal operation (display clear etc.)
0	1	Read busy flag (DB7) and address counter (DB0 to DB6)
1	0	DR write as an internal operation (DR to DDRAM or CGRAM)
1	1	DR read as an internal operation (DDRAM or CGRAM to DR)

Cursor/blink control circuit

As the name suggests, the cursor/blink control circuit generates the cursor or character blinking. The cursor or the blinking will appear with the digit located at the display data RAM (DDRAM) address set in the address counter (AC). For example, when the address counter is 0x08, the cursor position is displayed at DDRAM address 0x08.

Address counter (AC)

The address counter (AC) assigns addresses to both DDRAM and CGRAM. When an address of an instruction is written into the IR, the address information is sent from the IR to the AC.

Selection of either DDRAM or CGRAM is also determined concurrently by the instruction.

After writing to DDRAM or CGRAM, the AC is automatically incremented by 1. The AC contents are then output to DB0 to DB6 when RS = 0 and R/W = 1 (Table 22.4).

Solution

As discussed in 'Background', we will consider only LCD devices based on the Hitachi HD44780 controller.

Hardware

The HD44780 can send data in either two 4-bit operations or one 8-bit operation. This feature was probably originally intended to allow connection to both 4-bit and 8-bit microcontrollers. However, even with 8-bit microcontrollers (such as the 8051) the 4-bit interface is the most commonly applied, since:

- 1 The 4-bit interface is sufficiently fast for most applications (a single character is transferred in under 0.1 ms).
- 2 It saves four port pins.

We will use a 4-bit interface in our examples.

Despite its name the '4-bit' interface actually requires six port pins: four for the data bus and (usually) two additional port pins for the control lines (Figure 22.2).

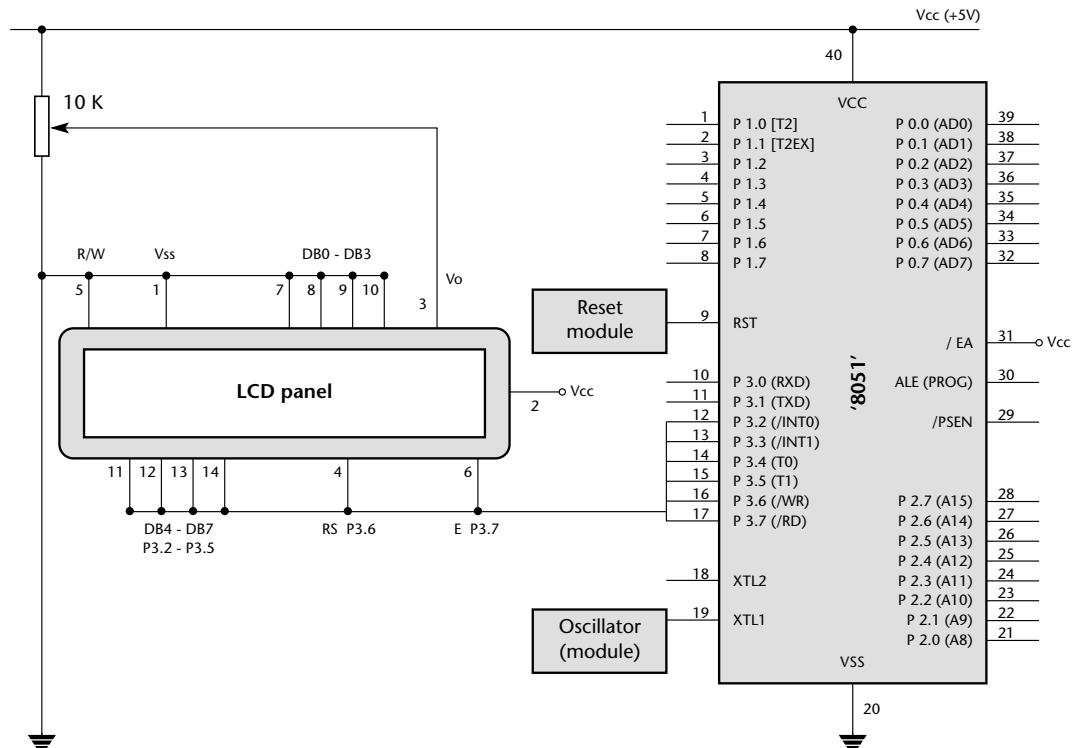


FIGURE 22.2 A 4-bit interface between an HD44780-based LCD panel and an 8051 microcontroller

In addition to the ground and power connections, you also need to connect the contrast adjustment pin (V_o). We have successfully used many LCD display by pulling this pin to ground. However, the recommended contrast adjustment takes the form of a potentiometer, connected as shown in Figure 22.3.

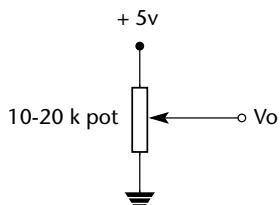


FIGURE 22.3 The recommended contrast adjustment connection

[Note: that, in many cases, a pulling V_o to ground will also work.]

Back lighting

If the module has a backlight, this will greatly increase the power consumption: check your data sheet for details. Before assuming that you require a backlight, try some of the modern displays now available: recent devices have greatly improved contrast and visibility.

The data bus

The data bus (4-bit) needs to be connected to four port pins with internal pull-up resistors. If using a port (e.g. Port 0) without internal pull-ups, add external 10K pull-ups (to V_{cc}).

Software

Complete software libraries are given in the examples that follow.

Note that in these examples – as in the RS-232 library – we have implemented the library as a **MULTI-STAGE TASK**: each character is written to a buffer and updates of the LCD display itself are carried out with a scheduled ‘update’ function.

Note also that these examples illustrate the use of user-defined characters.

Hardware resource implications

Uses a number of port pins, plus one timer.

Reliability and safety issues

LCD displays are reliable and have a long life.

Portability

This pattern can be applied to any microcontroller family.

Overall strengths and weaknesses

- ☺ LCDs provide the basis of a professional and flexible user interface.
- ☹ The panels are not cheap.
- ☹ Even a '4-bit' interface requires six pins.

Related patterns and alternative solutions

- See **MX LED DISPLAY** [page 450]
- See **PC LINK (RS-232)** [page 362]

Example: Displaying elapsed time on an LCD

This example displays elapsed time on an LCD character panel (Listings 22.1 to 22.3).

Hardware

The required hardware is shown in Figure 22.2.

Software

```
/*-----*  
Port.H (v1.00)  
-----*  
'Port Header' (see Chapter 10) for the project LCD_TIME  
-----*/  
// ----- Sch51.C -----  
// Comment this line out if error reporting is NOT required  
//#define SCH_REPORT_ERRORS  
  
#ifdef SCH_REPORT_ERRORS  
// The port on which error codes will be displayed  
// ONLY USED IF ERRORS ARE REPORTED  
#define Error_port P1  
  
#endif  
// ----- LCD_A.C -----  
  
// NOTE: Any combination of 6 pins may be used (any ports, any order)  
// NOTE: Number in [] are pin numbers on *MANY* LCDs
```

```

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; // Display register select output [4]
sbit LCD_EN = P3^7; // Display enable output [6]

// Connect Vss [1] on LCD to Gnd
// Connect Vcc [2] on LCD to +5V
// Connect Vo [3] on LCD to Gnd
// Connect RW [5] on LCD to Gnd

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 22.1 Part of a demonstration program that displays elapsed time on an LCD character panel

```

/*-----*
Elap_LCD.C (v1.00)

-----
Simple library function for keeping track of elapsed time
This version for LCD display.

*-----*/
#include "Main.h"
#include "Port.h"

#include "Elap_LCD.h"
#include "LCD_A.h"

// ----- Public variable definitions -----
tByte Hou_G = 0;
tByte Min_G = 0;
tByte Sec_G = 0;

// ----- Public variable declarations -----
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()

```

Init function for simple library displaying elapsed time on LCD character panel.

```
-----*/
void Elapsed_Time_LCD_Init(void)
{
    // Set up the initial data to be sent to the LCD
    // ASSUMES 20 CHARACTER DISPLAY
    char* pTime = "Elapsed time:-      ";
    tByte c;

    for (c = 0; c < LCD_CHARACTERS; c++)
    {
        LCD_data_G[0][c] = pTime[c];

        // For demo purposes only
        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()

Function for displaying elapsed time on LCD character panel.

*** Must be scheduled once per second ***

```
-----*/
void Elapsed_Time_LCD_Update(void)
{
    bit Min_update = 0;
    bit Hou_update = 0;

    // Set (for example) to 2 to test - otherwise 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)
    {
        // Need to update the 'hours' data
        LCD_data_G[0][15] = CHAR_MAP_G[Hou_G / 10];
        LCD_data_G[0][16] = CHAR_MAP_G[Hou_G % 10];
        Hou_update = 0;
    }

    if (Min_update)
    {
        // Need to update the minutes data
        LCD_data_G[0][18] = CHAR_MAP_G[Min_G / 10];
        LCD_data_G[0][19] = CHAR_MAP_G[Min_G % 10];
        Min_update = 0;
    }

    // We don't display seconds in this version.
    // We simply use the seconds data to turn on and off the colon
    // (between hours and minutes)
    if ((Sec_G % 2) == 0)
    {
        LCD_data_G[0][17] = ' ';
    }
    else
    {
        LCD_data_G[0][17] = ':';
    }
}

/* -----
   ---- END OF FILE -----
*/
```

Listing 22.2 Part of a demonstration program that displays elapsed time on an LCD character panel

```
/*-----*  
LCD_A.C (v1.00)  
-----  
LCD LIBRARY CODE  
Designed for scheduled operation,  
in this case for a 2-line x 20-character display  
'4-BIT' INTERFACE (uses 6 pins) to standard HD44780-based LCD  
-*-----*/  
  
// Hardware resources:  
// Uses T0 (for delays) plus six I/O pins  
  
#include "Main.h"  
#include "Port.h"  
  
#include "LCD_A.h"  
#include "Delay_T0.h"  
  
// ----- Public variable definitions -----  
  
// The LCD data  
char LCD_data_G[LCD_LINES][LCD_CHARACTERS+1]  
= {" PLEASE WAIT      , " ...           "};  
  
// ----- Private function prototypes -----  
  
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);  
  
// ----- Private constants -----  
  
// Bitmaps for user-defined characters [for demonstration purposes]  
  
// This is a UK Pound (currency) sign  
// 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)
// This is 'Degrees Celsius' (as in temp. of boiling water = 100oC)
// 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)

#define LCD_INC_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

// Define Timer 0 / Timer 1 reload values for ~50 us delay
#define PRELOAD50micros (65536 - (tWord)((OSC_FREQ /
20000)/(OSC_PER_INST)))
#define PRELOAD50microsH (PRELOAD50micros / 256)
#define PRELOAD50microsL (PRELOAD50micros % 256)

/*-----*/
LCD_Init()

RATHER SLOW, BUT MANAGES TO INITIALIZE ALL TESTED DISPLAYS
(and is only called at the start of the program)

NOTE: I suggest you call this function THREE TIMES.

/*-----*/
void LCD_Init(void)
{
    tByte loop;
    tByte l,c;

    // Set up the LCD port
    LCD_D4 = 1;
    LCD_D5 = 1;
    LCD_D6 = 1;
    LCD_D7 = 1;
}

```

```
LCD_RS = 1;
LCD_EN = 1;
Hardware_Delay_T0(500);

LCD_RS = 0;
LCD_EN = 0;

// Now wait for the display to initialize
// - data sheet says at least 40 ms
Hardware_Delay_T0(400);

// Data sheet says send this instruction 3 times...
for (loop = 0; loop < 3; loop++)
{
    // Using a 4-bit bus, 2 display lines and a 5x7 dot font
    LCD_Send_Byt(LCD_4BIT_2LINE_5x8FONT,0);
    Hardware_Delay_T0(100);
}

// Increment display address for each character but do not scroll
LCD_Send_Byt(LCD_INC_ADDR_NO_SCROLL,0);
Hardware_Delay_T0(50);

// Turn the display on and the cursor off
LCD_Send_Byt((LCD_CURSOR_OFF | LCD_DISPLAY_ON),0);
Hardware_Delay_T0(50);

// Clear the display
LCD_Send_Byt(LCD_CLEAR_DISPLAY,0);
Hardware_Delay_T0(50);

// Invisible cursor (dummy function call to avoid library error)
LCD_Control_Cursor(0,0,0);
Hardware_Delay_T0(200);

// Set up user-defined character(s) - if required
LCD_Create_Character(LCD_UDC_DEGREES_C, LCD_UDC_Degrees_C);
Hardware_Delay_T0(200);

LCD_Create_Character(LCD_UDC_POUNDS, LCD_UDC_Pounds);
Hardware_Delay_T0(200);

// Update all characters in the display
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()

This function updates one character in the LCD panel
(if it requires updating).

Duration: ~0.1 ms.

Schedule roughly every 25 ms (2-line x 20-char display) to
force one complete display update every second.

/*-----*/
void LCD_Update(void)
{
    static tByte Line;
    static tByte Character;

    tByte Tests, Data;
    bit Update_required;

    // Find next character to be updated
    Tests = LCD_CHARACTERS * LCD_LINES;
    do {
        if (++Character == LCD_CHARACTERS)
        {
            Character = 0;

            if (++Line == LCD_LINES)
            {
                Line = 0;
            }
        }

        // Array contents set to \0 after data are written to LCD
        Update_required = (LCD_data_G[Line][Character] != '\0');
    } while ((Tests-- > 0) && (!Update_required));

    if (!Update_required)
    {
        return; // No data in LCD requires updating
    }

    // Set DDRAM address which character is to be written to
    // - Assumes 2 line by 20 character display
}

```

```

// - See Table 22-3 for adjustments needed for other display sizes
if (Line == 0)
{
    LCD_SetDDRAM(0x00 + Character); // First line
}
else
{
    LCD_SetDDRAM(0xC0 + Character); // Second line
}

// This is the data for updating
Data = LCD_data_G[Line][Character];

// Send single data byte
LCD_Send_Byte(Data,1);

// Once data has been written to LCD
LCD_data_G[Line][Character] = '\0';
}

/*-----*/
LCD_Send_Byte()

This function writes a byte to the LCD panel.

Duration < 0.1 ms .

Parameters: DATA
            The byte to be written to the display.

DATA_FLAG:
            If DATA_FLAG == 1, a data byte is sent
            If DATA_FLAG == 0, a command byte is sent

/*-----*/
void LCD_Send_Byte(const tByte DATA, const bit DATA_FLAG)
{
    // Delays *are* needed
    // [you may find it possible to reduce them on
    // on some displays]
    LCD_D4 = 0;
    LCD_D5 = 0;
    LCD_D6 = 0;
    LCD_D7 = 0;
    LCD_RS = DATA_FLAG; // Data register
    LCD_EN = 0;
    LCD_Delay();
}

```

```

// Write the data (high nybble)
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; // Latch the high nybble
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();

// Write the data (low nybble)
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; // Latch the low nybble
LCD_Delay();
LCD_EN = 0;
LCD_Delay();
}

/*-----*
LCD_Control_Cursor()

This function enables or clears the cursor and moves
it to a specified point.

Params: Visible - Set if the cursor should be visible.
        Blinking - Set if character should be blinking
        Address - Address (DDRAM) we want to adjust.

*-----*/
void LCD_Control_Cursor(const bit VISIBLE, const bit BLINKING,
                        const tByte ADDRESS)
{
    // Cursor / blinking appears at current DDRAM address

```

```

// - use SetDDRAM() to alter the cursor position
tByte Command = 0x0C;

if (VISIBLE)
{
    Command |= 0x02;
}

if (BLINKING)
{
    Command |= 0x01;
}

LCD_Send_Byte(Command,0);
LCD_SetDDRAM(ADDRESS);
}

```

LCD_SetDDRAM()

Set the DDRAM to a particular address.

Used to determine where we write to in the LCD RAM and - thus - whether the text appears on Line 0, Line 1, etc.

See text for details.

Params: The DDRAM address we wish to write to.

```

*-----*/
void LCD_SetDDRAM(tByte ADDRESS)
{
    ADDRESS &= 0x7f;
    ADDRESS |= 0x80;
    LCD_Send_Byte(ADDRESS,0);
}

```

LCD_Create_Character()

Stores a user-defined character in the CG RAM. Up to 8 characters may be stored in this way. Note: characters are assumed to be 5x8 in size: if you need 5x11 characters you will need to adapt this code.

Parameters: The character data (see start of file)

```

*-----*/
void LCD_Create_Character(const tByte UDC_ID,
                           const tByte* const pUDC_PAT)

```

```

{
tByte Row;
tByte Address;

// Select CG RAM, appropriate address
Address = 0x40 + (UDC_ID << 3);
LCD_Send_Byte(Address, 0);

// Now write the data
for (Row = 0; Row < 8; Row++)
{
    LCD_Send_Byte(pUDC_PAT[Row], 1);
}

// Select DD RAM - address 0
Address = 0x80;

// Ensure that next data write is to DDRAM
LCD_Send_Byte(Address, 0);
}

/*-----*/
LCD_Delay()

This function provides a short delay for the LCD library.

/*-----*/
void LCD_Delay(void)
{
int x;

x++;
x++;
x++;
}

/*-----*/
---- END OF FILE
/*-----*/

```

Listing 22.3 Part of a demonstration program that displays elapsed time on an LCD character panel

Example: LCD and keypad

In this example we demonstrate a combined LCD–keypad interface (Figure 22.4).

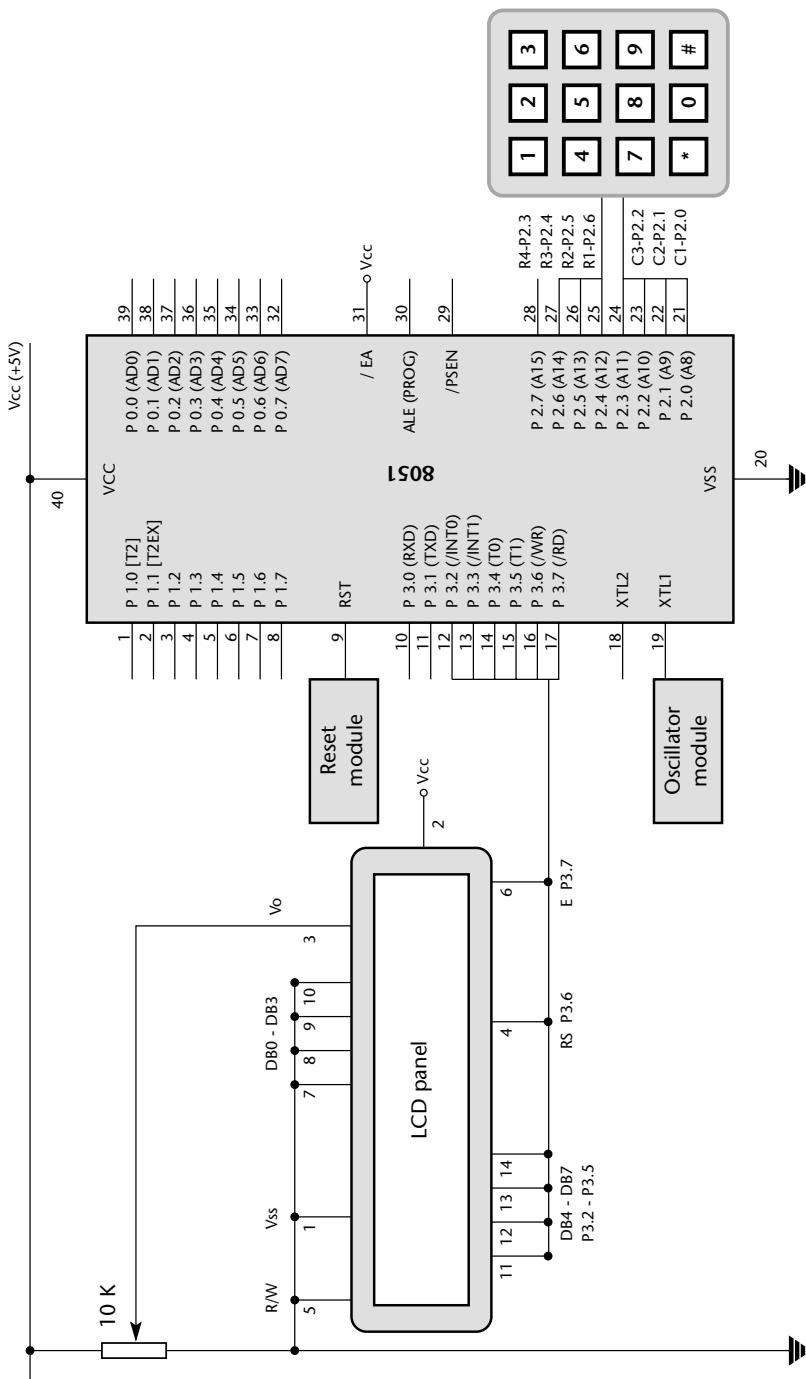


FIGURE 22.4 Hardware used to demonstrate a user interface consisting of an LCD display and a small keypad

The key source files for this example follow (Listings 22.4 to 22.6): all of the files required in this example are included on the CD.

```
/*-----*
Port.H (v1.00)

-----*

'Port file' (see Chapter 10) for the project SCH_TEST.PRJ

-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
// #define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#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 -----
// NOTE: Any combination of 6 pins may be used (any ports, any order)
// NOTE: Number in [] are pin numbers on *MANY* LCDs

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; // Display register select output [4]
sbit LCD_EN = P3^7; // Display enable output [6]

// Connect Vss [1] on LCD to Gnd
// Connect Vcc [2] on LCD to +5V
```

```
// Connect Vo [3] on LCD to Gnd  
// Connect RW [5] on LCD to Gnd  
  
/*-----*  
---- END OF FILE -----  
-----*/
```

Listing 22.4 Part of the software used to demonstrate a user interface consisting of an LCD display and a small keypad

```
/*-----*  
Main.c (v1.00)  
  
-----  
Demonstration program for:  
Keypad - LCD.  
This version for '8052' / 11.059MHz / 5 ms tick  
Required linker options (see Chapter 14 for details):  
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)  
{  
    // Prepare for the tasks  
    LCD_Init(); // 3x!!!  
    LCD_Init();  
    LCD_Init();  
  
    KEYPAD_Init();  
    LCD_KEY_Init();  
    LED_Flash_Init();
```

```

    // Set up the scheduler
    SCH_Init_T2();

    // Update the LCD from LCD buffer
    SCH_Add_Task(LCD_Update, 1000, 24);

    // Transfer keypad data to the LCD buffer
    SCH_Add_Task(LCD_KEY_Update, 2000, 50);

    // Start the scheduler
    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 22.5 Part of the software used to demonstrate a user interface consisting of an LCD display and a small keypad

```

/*-----*
LCD_Key.C (v1.00)

-----*
Simple demonstration function for transferring keypad inputs
to an LCD display.

*-----*/
#include "Main.h"
#include "Keypad.h"
#include "LCD_Key.h"
#include "LCD_A.h"
// ----- Private variables -----
static tByte Char_G;
// ----- Public variable declarations -----
extern char LCD_data_G[LCD_LINES][LCD_CHARACTERS+1];
/*-----*/

```

```
LCD_KEY_Init()  
    Init function for simple keypad - LCD library.  
/*-----*/  
void LCD_KEY_Init(void)  
{  
    tByte c;  
    char* pPrompt = "Type on the keypad:      ";  
    // Clear the keypad buffer  
    KEYPAD_Clear_Buffer();  
  
    // Prepare the display  
    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()  
    Function for displaying keypad inputs on LCD display.  
/*-----*/  
void LCD_KEY_Update(void)  
{  
    char Key, FnKey;  
    KEYPAD_Update();  
  
    if (KEYPAD_Get_Data_From_Buffer(&Key, &FnKey) == 0)  
    {  
        // Buffer is empty  
        return;  
    }  
    // Function key has been pressed (with another key)  
    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;
}
}

/* -----
   --- END OF FILE ---
*/
```

Listing 22.6 Part of the software used to demonstrate a user interface consisting of an LCD display and a small keypad

Further reading

Using serial peripherals

In Part E we consider how two powerful and influential serial communication protocols ('I²C' and 'SPI') may be utilized in applications with a time-triggered architecture.

Use of these protocols has two main advantages:

- They are designed to allow microcontrollers to be linked to a wide range of different peripherals – memory, displays, ADCs and similar devices – without requiring the use of large numbers of port pins.
- A common set of software code may be used with all SPI peripherals (for example), reducing the development effort required.

In Chapter 23, we consider the I²C bus, developed by Philips. I²C is a simple protocol, and may be easily generated in software. This allows the full range of 8051 devices (including Small 8051s, with very few spare port pins) to communicate with a wide range of peripheral devices.

In Chapter 24, we consider the SPI bus, developed by Motorola. Increasing numbers of 'Standard' and 'Extended' 8051 devices have hardware support for SPI and we will make use of these facilities in Chapter 24.

chapter **23**

Using 'I²C' peripherals

Introduction

The I²C bus is a two-wire serial communication bus, originally introduced by Philips in 1992. I²C is now supported by a wide range of semiconductor manufacturers, with the result that many different peripheral devices (LCDs, LED displays, EEPROMs, ADC and DAC devices and so on) can be connected to a microcontroller without using large numbers of port pins.

The pattern in this chapter addresses the following problem:

- Should you use the I²C protocol to link your microcontroller to peripheral devices and, if so, how do you do so?

I²C PERIPHERAL

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a scheduler.
- The microcontroller in your application will be interfaced to one or more peripherals, such as a keypad, EEPROM, digital-to-analog converter, or similar device.

Problem

Should you use the I²C protocol to link your microcontroller to peripheral devices and, if so, how do you do so?

Background

There are five key features of I²C as far as the developer of embedded applications is concerned:

- I²C is a protocol designed to allow microcontrollers to be linked to a wide range of different peripherals – memory, displays, ADCs and similar devices – and requires only two port pins to connect to (typically) up to 20 peripherals.
- There are many I²C peripherals available for purchase ‘off the shelf’.
- I²C is a simple protocol and may be easily generated in software. This allows *all* 8051 devices to communicate with a wide range of peripheral devices.
- A common set of software code may be used with all I²C peripherals.
- I²C is fast enough (even when generated in software) to be compatible with time-triggered architectures. Typical data transfer rates will be up to 1,000 bytes / second (with a 1 ms scheduler tick).

We provide some background necessary for understanding and using the I²C bus in this section. Please note that much of the material here is adapted from the Philips (1998) I²C specification. For further details, the reader is referred to this document, a copy of which can be obtained from the Philips WWW site.¹

Hardware

We begin by considering essential I²C hardware features.

1. www.philips.com

The bus

In the two-wire I²C bus the serial data (SDA) and serial clock (SCL) lines carry the information between the various devices (Figure 23.1). Due to the variety of different technologies (CMOS, NMOS, bipolar) used to create devices which can be connected to the I²C bus, the levels of the logical '0' (LOW) and '1' (HIGH) are not fixed and depend on the system supply voltage.

When the bus is free, both SCL and SDA lines are HIGH.

Both SDA and SCL are bidirectional lines. The output stages of devices connected to the bus should be open drain (open collector) in order to match the requirements of this protocol. This will often mean using pins on Port 0 of an 8051 device (but see following box).

The number of devices that can be connected to the I²C bus is limited only by the maximum bus load capacitance of 400 pF: in the absence of suitable information, assume that each peripheral device and its wiring contributes a total of 20 pF to the total capacitance.

Each of the lines is connected to the Vcc supply via a common pull-up resistor. If we assume a capacitance of 20 pF per device, the required value of resistor is determined by the maximum rise time in the I²C specification (1,000 ns in Standard I²C). This can be shown to translate into the following:

$$R = \frac{50}{d}$$

where: R is the required resistance ($\text{K}\Omega$)

d is the number devices on the bus

Note that, in various applications, we have successfully used 8051 devices with 'conventional' (not open-drain) pins for I²C communications, without the use of external pull-up resistors.

If you do the same, we recommend that you do so only where a small number of peripherals are used and that you test the resulting application thoroughly.

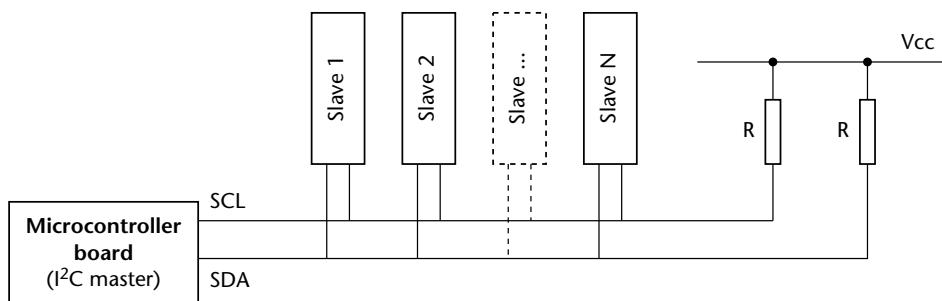


FIGURE 23.1 Using the I²C bus

Device addresses

In I²C, the communication is address based: that is, each device on the bus has a unique address and messages can be directed from anywhere on the bus to the device with a particular address.

The device addresses are ten-bits long in the latest version of this standard (seven bits in the original), allowing more than 1,000 different addresses to be used. Note that devices with 7-bit and 10-bit addresses may be used on the same bus.

While the facility for 7-bit or 10-bit addresses are included in the I²C specification, in most practical cases, the address of a device is partially hard wired into the device.

Consider, for example, a useful I²C peripheral, the Atmel 24C64 serial EEPROM.² This device provides 8192 × 8 bits of non-volatile data storage, with data retention for 100 years. As such it is ideal, for example, for storing small amounts of data in monitoring applications or for storing system passwords in other devices.

The pinout of the 24C64 is given in Figure 23.2. Note from the figure that there are only three address lines, which will usually be pulled to Vcc or ground to set the required device address. Assuming each device is given a unique address, up to eight 24C64s may be connected to one I²C bus, rather than 128 (2^7) or 1024 (2^{10}) as is suggested by the I²C standard.

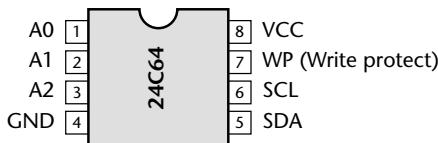


FIGURE 23.2 Pin configuration for the Atmel 24C64 serial EEPROM

As far as the bus is concerned, the 24C64 has, in fact, a 7-bit address. However, the most significant four bits are 'hard wired' into the device. These bits are always '1010' (0xA0), which is a code common to many serial EEPROMs. The full device address is therefore '1 0 1 0 A2 A1 A0'.

The hard-wiring of part of the device address makes practical sense. Few applications require more than eight serial EEPROMs on a single bus, particularly since quite large capacity memory devices are now available with an identical interface. If the full device address needed to be coded by the user for each device, the size of each chip would have to be increased to allow for (at least) four more pins (Figure 23.3).

Very similar schemes are used in other I²C devices. For example, the Dallas 1621 temperature sensor³ is a useful and inexpensive component that provides a temperature reading, in Celsius, over an I²C bus (Figure 23.4).

2. A complete example of an I²C link from an 8051 to a 24C64 device is given on page 510.

3. A complete example of an I²C link from an 8051 to a DS1621 temperature sensor is given on page 515.

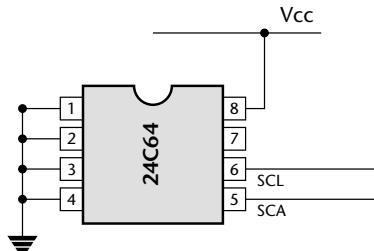


FIGURE 23.3 A 24C64 with device address '1010000'

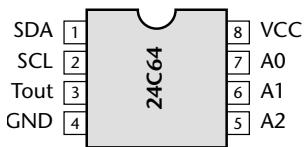


FIGURE 23.4 Pin configuration for the Dallas DS1621 (I²C) temperature sensor

Here, again, only three least significant bits of the address are user adjustable. In this case, the full address of the device is '1 0 0 1 A2 A1 A0'. The hard-wired part of this address (0x90) is shared by many DAC devices.

The I²C protocol: Masters and Slaves

Each device on an I²C bus can operate as either a transmitter or receiver of data. For example, an LCD driver will generally act as a receiver, whereas a memory device can act as both a receiver and a transmitter. In addition to transmitters and receivers, devices can also be considered as Masters or Slaves. A Master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered to be a Slave.

The I²C bus has a multi-Master capability: this means that more than one device capable of controlling the bus can be connected to it. This might be the useful, for example, if I²C were used to transfer data between two microcontrollers.

Please note that in this pattern we are concerned only with I²C applications involving a single microcontroller (the device Master) and one or more Slave peripherals.

Generation of clock signals on the I²C bus is always the responsibility of a Master device; the Master generates its own clock signals when transferring data on the bus. In a single-Master system, bus clock signals from a Master can only be altered when they are stretched by a slow Slave device holding down the clock line.

Performing data transfers

We now consider how data are transferred between devices. Consider, for example, that we wish to read a temperature value from a DS1621 temperature sensor discussed earlier: we would do so as follows:

- 1 Generate a START condition
- 2 Send DS1621 device address (with WRITE access request)
- 3 Make sure that the Slave (DS1621) generates an ACKNOWLEDGE bit
- 4 Send the command 'Read Temperature' (0xAA)
- 5 Make sure that the Slave (DS1621) generates an ACKNOWLEDGE bit
- 6 Generate another START condition
- 7 Send DS1621 device address (with READ access request this time)
- 8 Make sure that the Slave (DS1621) generates an ACKNOWLEDGE bit.
- 9 Receive the first (most significant) byte of temperature data from the I²C bus
- 10 Perform a MASTER – ACKNOWLEDGE
- 11 Receive the second byte of temperature data from the I²C bus
- 12 Perform a MASTER – NOT ACKNOWLEDGE
- 13 Generate a STOP condition

Although this process appears at first glance rather complex, it contains all the core I²C routines that are required for communication with any I²C peripheral. As a result, when we have considered this process, you will be in a position to develop I²C interfaces for a very wide range of devices.

To understand the discussions that follow, it will be helpful to refer to Figure 23.5.

We begin by considering the START and STOP conditions.

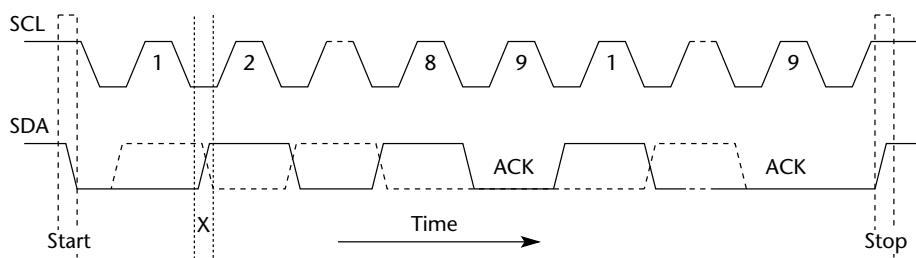


FIGURE 23.5 The basic format of I²C communications

[Note: that one clock pulse is generated for each data bit transferred. Note also that the data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW.]

START and STOP conditions

I²C communications begin with a START condition. Further START conditions may be generated within the message, which will end with a STOP condition.

A START condition is represented as follows (see Figure 23.5): a HIGH to LOW transition on the SDA line while SCL is HIGH.

A STOP condition is represented as follows (see Figure 23.5): a LOW to HIGH transition on the SDA line while SCL is HIGH.

START and STOP conditions are always generated by the Master.

Byte format

Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an ACKNOWLEDGE bit (see following sections).

Data are transferred with the most significant bit first. If a Slave cannot receive or transmit another complete byte of data until it has performed some (internal) operation, for example updating memory, it can hold the SCL line LOW to force the Master into a wait state. Data transfer then continues when the Slave is ready for another byte of data and releases SCL.

Slave 'ACKNOWLEDGE' and 'NOT ACKNOWLEDGE'

In many situations the Master (the microcontroller in our case) will be sending data to the Slave (for example, an EEPROM). In these cases, the 'Slave-receiver' must generate an 'Slave-receiver ACKNOWLEDGE' signal when it has received a byte of data.

The ACKNOWLEDGE-related clock pulse is always generated by the Master. The transmitter releases the SDA line (HIGH) during the ACKNOWLEDGE clock pulse. The Slave-receiver must pull down the SDA line during the ACKNOWLEDGE clock pulse so that it remains stable LOW during the HIGH period of this clock pulse.

Usually, an active Slave-receiver is obliged to generate an ACKNOWLEDGE after each byte has been received.

When a Slave does not acknowledge the Slave address (for example, it is unable to receive or transmit because it is performing some internal operation), the data line must be left HIGH by the Slave. The Master can then generate either a STOP condition to abort the transfer or a repeated START condition to start a new transfer.

If a Slave-receiver does acknowledge the Slave address but, some time later in the transfer cannot receive any more data bytes, the Master must again abort the transfer. This is indicated by the Slave generating a 'NOT ACKNOWLEDGE' response on the first byte to follow: this means that the Slave leaves the data line HIGH during the ACKNOWLEDGE clock pulse generated by the Master. The Master will then generate a STOP (or a repeated START) condition and – probably after a delay – will attempt the transmission again.

Master-receiver 'ACKNOWLEDGE' and 'NOT ACKNOWLEDGE'

In some situations the Master (the microcontroller in our case) will be receiving data from the Slave (for example, the temperature sensor). In some cases, the 'Master-receiver' must generate a 'Master-receiver ACKNOWLEDGE' signal when it has received a byte of data. However, at the end of the data transfer (when the Master-receiver has received the last byte of data it requires), it must signal the end of data to the Slave-transmitter by not generating an ACKNOWLEDGE on the last byte that was clocked out of the Slave. The Slave-transmitter must release the data line to allow the Master to generate a STOP or repeated START condition.

Synchronization

All Masters generate their own clock on the SCL line to transfer messages on the I²C bus. Data are only valid during the HIGH period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

Clock synchronization is performed using the wired AND connection of I²C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line will cause the devices concerned to start counting off their LOW period and, once a device clock has gone LOW, it will hold the SCL line in that state until the clock HIGH state is reached. However, the LOW to HIGH transition of this clock may not change the state of the SCL line if another clock is still within its LOW period. The SCL line will therefore be held LOW by the device with the longest LOW period. Devices with shorter LOW periods enter a HIGH wait-state during this time.

When all devices concerned have counted off their LOW period, the clock line will be released and go HIGH. There will then be no difference between the device clocks and the state of the SCL line and all the devices will start counting their HIGH periods. The first device to complete its HIGH period will again pull the SCL line LOW. In this way, a synchronized SCL clock is generated with its LOW period determined by the device with the longest clock LOW period and its HIGH period determined by the one with the shortest clock HIGH period.

Further details

A complete technical specification for the I²C bus will be found on the Philips WWW site.

Solution

Should you use I²C?

In order to determine whether use of an I²C bus is appropriate in your time-triggered application, we consider some key questions that should be asked when considering the use of any communications protocol or related technique.

Main application areas

The I²C bus was designed mainly to allow the interconnection of components within a single application. Although the bus may be used, for example, to connect processors (usually microcontrollers) to one another or to other computer systems, its main application area is in the connection of standard peripheral devices, such as LCD panels or EEPROMs, or 'smart' components in consumer applications (e.g. TV tuners, PLL synthesizers, video processors) to microcontrollers.

Ease of development

I²C can be used to communicate with a large number and range of peripherals. By using the same protocol to talk to a range of devices, development efforts may be reduced.

Scalability

The maximum size of an I²C bus is limited by the maximum capacitance of 400 pF. To estimate the maximum network size, we can add up the input capacitance of all the devices we wish to connect to the bus.

For example, the SGS-Thomson ST24C16 16K I²C-compatible EEPROM has an input capacitance of around 8 pF, and we could attach some 50 of these devices together on the same bus. Note that this rough calculation ignores any effects of stray capacitance due to the cabling itself. If we assume a 'standard' capacitance per device of 20 pF (including stray capacitance), we can still hope to connect around 20 peripherals to the microcontroller over a single bus.

Overall, the bus supports large enough collections of peripherals to meet the needs of its main application area.

Flexibility

I²C is flexible. Networks of one Master and multiple Slaves and networks with multiple Masters can both be implemented. Note: we consider only single-Master applications in this pattern.

Speed of execution and size of code

As we have mentioned, I²C can operate over a wide speed range: from 100 kbits/s in the 'standard' version (1992), to 400 kbits/s in the 'fast' version, and now 3.4 Mbit/s in the latest 'high-speed' version introduced in 1998.

However, in this pattern we are concerned with I²C interfaces created entirely in software. Such interfaces are only practical at the low end of the speed range.

Generation of the I²C protocol inevitably adds to the code size: see the core I²C library (in the following example) for details.

Cost

The cost of licence fees for use of the bus is included in the cost of the peripheral components which you purchase: in most circumstances, there are no additional fees to pay.

Note that this may not be the case if, for example, you are implementing a I²C peripheral (to be sold for connection to an I²C bus). If in doubt, please contact Philips for further details.

Choice of implementations and vendors

The I²C library presented here may be used with any 8051 device.

Suitability for use in time-triggered applications

As we saw in Chapter 18, the RS-232 communication protocol is appropriate for use in time-triggered applications. This suitability arises because the task duration associated with transmission (and reception) of data on an RS-232 network is very short. Note that this transmission time is not directly linked to the baud rate of the network, largely because almost all of the 8051 family have on-chip hardware support for RS-232, with the result that messages are transmitted and received ‘in the background’.

The situation with I²C is rather different. Specifically, in this pattern, we are concerned with software-based I²C protocols; this undoubtedly adds to the software load. For example, if we consider the process of sending one byte of data to an I²C-based ROM chip (an example of this is presented in full later), then the total task duration is approximately 0.5 ms.

This task duration can be supported on a time-triggered application, even with 1 ms timer ticks, if the maximum data rate (1000 bytes / second) matches the needs of the application.

How do you use I²C in a time-triggered application?

The examples that follow include a complete I²C library which may be used with any member of the 8051 family.

Hardware resource implications

I²C requires the use of two port pins. This is considerably fewer than would be required to create a parallel interface to most peripheral devices.

There is, however, a significant CPU load: see ‘Solution’ for details.

Reliability and safety implications

The I²C protocol incorporates only minimal error-checking mechanisms: detection of data corruption (for example) during the transfer of information to or from a periph-

eral device must be carried out in software, if required. However, in most cases, damage to cabling will be quickly detected.

Portability

I²C is a simple protocol and may be easily generated in software. This allows the techniques presented here to be used with *all* 8051 devices.

Overall strengths and weaknesses

- ☺ I²C is supported by a wide range of peripheral devices.
- ☺ I²C requires only two port pins to connect to (typically) up to 20 peripherals.
- ☺ I²C is a simple protocol and may be easily generated in software. This allows *all* 8051 devices to communicate with a wide range of peripheral devices.
- ☺ A common set of software code may be used with all I²C peripherals.
- ☹ Although I²C is fast enough (even when generated in software) to be compatible with time-triggered architectures, typical data transfer rates will be comparatively slow (up to 1000 bytes / second with a 1 ms scheduler tick).

Related patterns and alternative solutions

See SPI PERIPHERAL [page 521].

Remember also that, if possible, the best approach is to avoid peripherals altogether and use a microcontroller with on-chip facilities if possible. For example, rather than using a basic 8051 plus serial EEPROM, consider using the Atmel AT89LS8252, which has two kbytes of EEPROM on the chip.

Example: I²C core library

In this section we present a library of core I²C routines (Listings 23.1 to 23.3). In most cases you will need to add some additional functions to use these files with particular I²C peripherals: we illustrate this process in the examples that follow.

Please note that not all I²C devices use ACKNOWLEDGE / NOT ACKNOWLEDGE signals (discussed earlier). Similarly, not all devices need a 'read byte' facility (PWM and DAC outputs, for example). You can omit these library features by means of these lines at the top of the I2C_Core.C file:

```
// Comment out these lines if these functions are *not* required  
//#define I2C_ACK_NACK  
#define I2C_READ_BYTE
```

As usual, all of these files are included on the CD.

```

/*-----*
Port.H (v1.00)

-----'
'Port Header' (see Chapter 10) for the I2C core library.

*-----*/
// ----- I2C_Core.C -----
// The two-wire I2C bus
// NOTE: If using external pull-ups,
//       you will generally want to use Port 0
sbit I2C_SCL = P1^7;
sbit I2C_SDA = P1^6;

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 23.1 Part of the I²C core library

```

/*-----*
I2C_Core.H (v1.00)

-----'
- See I2C_Core.C for details.

*-----*/
#include "Main.h"

// ----- Public function prototypes -----
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);

// ----- Public constants -----
#define I2C_READ      0x01    // Read command
#define I2C_WRITE     0x00    // Write command

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 23.2 Part of the I²C core library

```
/* ----- *-
I2C_CORE.C (v1.00)

----- */

Core of the I2C library.

You will typically need other components
(see, for example, I2C_ROM.C) to create a complete library.

-*----- */
```

```
#include "Main.h"
#include "Port.h"

#include "I2C_Core.h"
#include "TimeoutH.H"

// ----- Private function prototypes -----
static tByte I2C_Get_Ack_From_Slave(void);
static bit I2C_Sync_The_Clock_T0(void);
static void I2C_Delay(void);

// Comment out this line if these functions are *not* required
//#define I2C_ACK_NACK

/* ----- */
I2C_Send_Start()

Generates a 'start' condition.

-*----- */
void I2C_Send_Start(void)
{
    // Prepare the bus
    I2C_SCL = 1;
    I2C_SDA = 1;
    I2C_Delay();

    // Generate the START condition
    I2C_SDA = 0;
    I2C_Delay();
    I2C_SCL = 0;
}

/* ----- */
I2C_Send_Stop()

Generates a 'stop' condition.

-*----- */
```

```
void I2C_Send_Stop(void)
{
    I2C_SDA = 0;
    I2C_Delay();
    I2C_SCL = 1;
    I2C_Delay();
    I2C_SDA = 1;
}

/*-----*
I2C_Get_Ack_From_Slave()

We are implementing a 'Master-Slave' communication protocol
here, with the microcontroller as the Master. This function
waits (with timeout) for an acknowledgement from the Slave device.
*-----*/
tByte I2C_Get_Ack_From_Slave(void)
{
    // Prepare the bus
    I2C_SDA = 1;
    I2C_SCL = 1;

    if(I2C_Sync_The_Clock_T0())
    {
        return 1; // Error - failed to sync
    }

    // Managed to synchronize the clock
    I2C_Delay();

    if (I2C_SDA)
    {
        // Generate a clock cycle
        I2C_SCL = 0;

        return 1; // Error - No ack from Slave
    }

    I2C_SCL = 0; // Generate a clock cycle

    return 0; // OK - Slave issued ack
}

/*-----*
I2C_Write_Byt()

Send a byte of data to the Slave.
Supports slow Slaves by allowing 'clock stretching'.
*-----*
```

Duration ~100 µs (except in the event of bus error).

```
-----*/
tByte I2C_Write_Byte(tByte Data)
{
    tByte Bit = 0;

    // Sending data one bit at a time (MS bit first)
    for (Bit = 0; Bit < 8; Bit++)
    {
        I2C_SDA = (bit)((Data & 0x80) >> 7);
        I2C_SCL = 1;

        if (I2C_Sync_The_Clock_T0())
        {
            return 1; // Error - failed to sync
        }

        I2C_Delay();

        // Generate a clock cycle
        I2C_SCL = 0;

        // Prepare to send next bit
        Data <= 1;
    }

    // Make sure the Slave acknowledges
    return(I2C_Get_Ack_From_Slave());
}

-----*/
I2C_Read_Byte()
Read a byte of data from the Slave.
Supports slow Slaves by allowing 'clock stretching'.

-----*/
tByte I2C_Read_Byte(void)
{
    tByte result = 0; // Return value with read I2C byte
    tByte Bit = 0; // Bitcounter

    for (Bit = 0; Bit < 8; Bit++)
    {
        I2C_SDA = 1; // Release SDA
        I2C_SCL = 1; // Release SCL

        if (I2C_Sync_The_Clock_T0())
        {
```

```

        return 1; // Error - failed to sync
    }

    I2C_Delay();

    result <<= 1;      // Shift left the result

    if (I2C_SDA)
    {
        result |= 0x01; // Set actual SDA state to LSB
    }

    I2C_SCL = 0;       // Force a clock cycle
    I2C_Delay();
}

return(result);
}

/*-----*
I2C_Sync_The_Clock_T0()
Low-level function used during I2C data transfers.
*** With 1ms hardware (Timer 0) timeout ***
RETURNS: 1 - Error (not synchronized)
         0 - OK (clock synchronized)
*-----*/
bit I2C_Sync_The_Clock_T0(void)
{
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 1ms
    TH0 = T_01ms_H; // See TimeoutH.H for T_ details
    TL0 = T_01ms_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    // Try to synchronize the clock
    while ((I2C_SCL == 0) && (TF0 != 1));

    TR0 = 0; // Stop the timer

    if (TF0 == 1)
    {
        return 1; // Error - Timeout condition failed
    }
}

```

```
    }

    return 0; // OK - Clock synchronized
}

/*-----*
 I2C_Delay()

 A short software delay (around 10 µs).

 Adjust this for a minimum of 5.425 µs to work with
 'standard' I2C devices. Any delay longer than this will also work.
 With modern devices shorter delays may also be used.

 NOTE: Cannot generally do this with a Hardware Delay.

-----*/
void I2C_Delay(void)
{
    int x;

    x++;
    x++;
}

#ifndef I2C_ACK_NACK
/*-----*
 I2C_Send_Master_Ack()

 Generates an 'ACKNOWLEDGE' condition.

-----*/
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()

 Generates a 'NOT ACKNOWLEDGE' condition.

-----*/
void I2C_Send_Master_NAck(void)
{
    I2C_SDA = 1; I2C_SCL = 1;
```

```

I2C_Sync_The_Clock_T0();

I2C_Delay();
I2C_SCL = 0;
}
#endif

/* -----
--- END OF FILE
----- */

```

Listing 23.3 Part of the I²C core library

Example: I²C EEPROM interface

In this example we consider how to link an 8051 microcontroller to a serial EEPROM using an I²C bus. Note that use of such EEPROMs is a common way of retaining system settings after power is removed: the storage is non-volatile, for a period of around 100 years.

Hardware

The required hardware is illustrated in Figure 23.6.

Note that, in this case, we are not using an open-collector port and do not require pull-up resistors. If you are using more than one I²C device, it is recommended that you adapt the code to work with Port 0 (a trivial change) and use pull-up resistor values calculated as described in 'Background'.

Software

The required software is contained in Listings 23.4 to 23.6. Note that you will also require the core I²C library from the 'Solution' section.

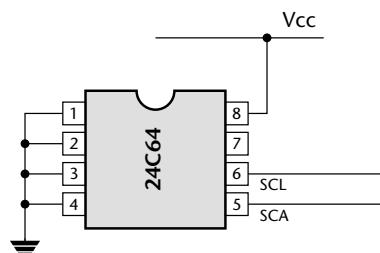


FIGURE 23.6 A 24C64 with device address '1010000'

```
/*-----*
Main.C (v1.00)

-----
Simple test program for I2C (24C64) library.

Connect a 24C64 to the SDA and SCL pins described
in the library file (I2C_Core.C).

Terminating resistors not generally required on the bus.

Pull all three address lines (on the EEPROM) to ground.

-----*/
#include "Main.h"
#include "I2C_ROM.h"
#include "Delay_T0.h"

// In this test program, we define the error code variable here
// (usually in the scheduler library)
tByte Error_code_G = 0;

void main( void )
{
    tByte data1 = 0;
    tByte data2 = 0;

    tWord Data_address = 0;

    while (1)
    {
        Data_address = data1;

        // Write to eeprom
        I2C_Write_Byte_AT24C64(Data_address, data1);

        // Read back from eeprom
        data2 = I2C_Read_Byte_AT24C64(Data_address);

        // Display value from eeprom
        P1 = 255 - data2;

        // Display error code
        P2 = 255 - Error_code_G;

        if (++data1 == 255)
        {
            data1 = 0;
        }
}
```

```

        Hardware_Delay_T0(1000);
    }
}

/* -----
--- END OF FILE
--- */

```

Listing 23.4 Part of an I²C EEPROM library

```

/* -----
I2C_ROM.H (v1.00)

-----
- See I2C_ROM.C for details.

--- */
#include "Main.h"
// ----- Public function prototypes -----

// Read byte from EEPROM
tByte I2C_Read_Byte_AT24C64(tWord);

// Write byte to EEPROM
void I2C_Write_Byte_AT24C64(tWord,tByte);

/* -----
--- END OF FILE
--- */

```

Listing 23.5 Part of an I²C EEPROM library

```

/* -----
I2C_ROM.C (v1.00)

-----
I2C library functions which are intended to allow
use of 2-wire serial EEPROMs (specifically, AT24C64).

Easily extended / adapted to work with other 2-wire EEPROMs.

--- */
#include "Main.h"
#include "Delay_T0.h"
#include "I2C_Core.h"
#include "I2C_ROM.h"

```

```
// ----- Public variable declarations -----
extern tByte Error_code_G;

// ----- Private constants -----
// Device identifier of the EEPROM used in this example
// - see text / Philips documentation for IDs used by other devices
#define I2C_EEPROM_ID 0xA0

/*-----*
 *----- I2C_Write_Byte_AT24C64()
 Send a byte of data to the EEPROM.
 *-----*/
void I2C_Write_Byte_AT24C64(const tWord address, tByte content)
{
    tByte MSByte; // Most significant byte of data address
    tByte LSByte; // Least significant byte of data address

    I2C_Send_Start(); // Generate START condition

    // Send SLAVE address with write request
    if (I2C_Write_Byte(I2C_EEPROM_ID | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
        return;
    }

    // MSByte of address
    MSByte = (address >> 8) & 0x00FF;

    // LSByte of Address
    LSByte = address & 0x00FF;

    // Send memory address
    if (I2C_Write_Byte(MSByte))
    {
        Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
        return;
    }

    // Send memory address
    if (I2C_Write_Byte(LSByte))
    {
        Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
        return;
    }

    // Send content to memory address
```

```
    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.

*t-----*/
tByte I2C_Read_Byte_AT24C64(tWord address)
{
    tByte MSByte; // Most significant byte of data address
    tByte LSByte; // Least significant byte of data address
    tByte Result = 0;

    I2C_Send_Start(); // Generate START condition

    // Send SLAVE address with dummy write request
    if (I2C_Write_Byte(I2C_EEPROM_ID|I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_READ_BYTE_AT24C64;
        return 0;
    }

    // MSByte of address
    MSByte = (address >> 8) & 0x00FF;

    // LSByte of address
    LSByte = address & 0x00FF;

    // Send memory address
    if (I2C_Write_Byte(MSByte))
    {
        Error_code_G = ERROR_I2C_READ_BYTE_AT24C64;
        return 0;
    }

    // Send memory address
    if (I2C_Write_Byte(LSByte))
    {
        Error_code_G = ERROR_I2C_READ_BYTE_AT24C64;
        return 0;
    }
}
```

```

I2C_Send_Start(); // Generate START condition
// Send SLAVE address with read request
if (I2C_Write_Byte(I2C_EEPROM_ID | I2C_READ))
{
    Error_code_G = ERROR_I2C_READ_BYTE_AT24C64;
    return 0;
}
Result = I2C_Read_Byte(); // Read memory content
// Don't perform a MASTER ACK
I2C_Send_Stop();
return(Result);
}

/* -----
----- END OF FILE -----
----- */

```

Listing 23.6 Part of an I²C EEPROM library

Example: I²C Temperature sensor interface

In this example we consider how to link an 8051 microcontroller to Dallas DS1621 (I²C) temperature sensor.

Hardware

See Figure 23.4.

Software

The key software files are in Listings 23.7 to 23.9: all the files required for the project are included on the CD, in the directory associated with this chapter.

```

/* -----
----- Main.C (v1.00)
-----
Simple test program for I2C (DS1621) library.
Connect a DS1621 to the SDA and SCL pins described
in the library file (I2C_Core.C).
Terminating resistors not generally required on the bus.
----- */

```

```

#include "Main.h"
#include "I2C_1621.h"
#include "Delay_T0.h"

tByte Temperature_G;

// In this test program, we define the error code variable here
// (usually in the scheduler library)
tByte Error_code_G = 0;

void main( void )
{
    I2C_Init_Temperature_DS1621();

    while(1)
    {
        I2C_Read_Temperature_DS1621();
        P1 = 255 - Temperature_G;
        P2 = 255 - Error_code_G;
        Hardware_Delay_T0(1000);
    }
}

/*-----*
----- END OF FILE -----*
-----*/

```

Listing 23.7 Part of an I²C temperature sensor example

```

/*-----*
----- I2C_1621.H (v1.00)

-----*
----- See I2C_1621.H for details.

-----*
-----*/
#include "Main.h"

// ----- Public function prototypes -----
void I2C_Read_Temperature_DS1621(void);
void I2C_Init_Temperature_DS1621(void);

/*-----*
----- END OF FILE -----*
-----*/

```

Listing 23.8 Part of an I²C temperature sensor example

```
/*-----*
I2C_1621.C (v1.00)

-----
I2C-based library for DS1621 temperature sensor.

*-----*/
#include "Main.h"
#include "I2C_core.h"
#include "I2C_1621.h"
#include "Delay_T0.h"

// ----- Public variable declarations -----
extern tByte Temperature_G;
extern tByte Error_code_G;

// ----- Private constants -----
#define I2C_DS1621_ID 0x90

/*-----*
I2C_Init_Temperature_DS1621()
Sets the sensor to 'continuous convert' mode to allow
temperature readings to be subsequently obtained.

*-----*/
void I2C_Init_Temperature_DS1621(void)
{
    I2C_Send_Start(); // Generate START condition
    // Send SLAVE address with write request
    if (I2C_Write_Byte(I2C_DS1621_ID | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_DS1621;
        return;
    }

    // Send control byte :
    // configure command
    if (I2C_Write_Byte(0xAC))
    {
        Error_code_G = ERROR_I2C_DS1621;
        return;
    }

    // Send configuration data - CONTINUOUS mode
    if (I2C_Write_Byte(0x00))
```

```

    {
        Error_code_G = ERROR_I2C_DS1621;
        return;
    }

    I2C_Send_Stop();      // Generate STOP condition
    // Must delay here to allow EEPROM (in sensor)
    // to store these data. Sheet says 10ms.
    Hardware_Delay_T0(100);

    // Now start temperature conversions
    I2C_Send_Start();    // Generate START condition
    // Send SLAVE address with write request
    if (I2C_Write_Byte(I2C_DS1621_ID | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_DS1621;
        return;
    }

    // Send command data - start converting
    if (I2C_Write_Byte(0xEE))
    {
        Error_code_G = ERROR_I2C_DS1621;
        return;
    }

    I2C_Send_Stop();      // Generate STOP condition
}

/*-----*/
I2C_Read_Temperature_DS1621()

The sensor samples continuously (around 1 new value per second).
We obtain the latest value.

/*-----*/
void I2C_Read_Temperature_DS1621(void)
{
    tByte result = 0;

    I2C_Send_Start();    // Generate START condition
    // Send DS1621 device address (with write access request)
    if (I2C_Write_Byte(I2C_DS1621_ID | I2C_WRITE))
    {
        Error_code_G = ERROR_I2C_DS1621;
        return;
    }
}

```

```
// Send command - Read Temperature (0xAA)
if (I2C_Write_Byte(0xAA))
{
    Error_code_G = ERROR_I2C_DS1621;
    return;
}

I2C_Send_Start(); // Generate START condition (again)

// Send DS1621 device address (with READ access request this time)
if (I2C_Write_Byte(I2C_DS1621_ID | I2C_READ))
{
    Error_code_G = ERROR_I2C_DS1621;
    return;
}

// Receive first (MS) byte from I2C bus
Temperature_G = I2C_Read_Byte();
I2C_Send_Master_Ack(); // Perform a MASTER ACK

// Here we require temperature only accurate to 1 degree C
// - we discard LS byte (perform a dummy read)
I2C_Read_Byte();
I2C_Send_Master_NAck(); // Perform a MASTER NACK

I2C_Send_Stop(); // Generate STOP condition
}

/* -----
----- END OF FILE -----
----- */
```

Listing 23.9 Part of an I²C temperature sensor example

Example: I²C ADC

See Chapter 32 for an example of an I²C interface to an analogue-to-digital converter.

Further reading

See the Philips I²C specification (see www.philips.com) for further details.

Using 'SPI' peripherals

Introduction

When using asynchronous serial protocols, such as 'RS-232', the two devices that are communicating must agree on a communication frequency (the baud rate) and each device then uses (say) an independent crystal-based clock to ensure that it operates at the required rate. One consequence of this approach is that, if the clocks in the transmitter and receiver devices vary by more than a few per cent, the receiving device will be unable to decode the incoming data correctly.

By contrast, the serial peripheral interface (SPI) uses a *synchronous* communication protocol: this means that both the transmitter and receiver devices share a common clock. The transitions of this common clock determine when to send and receive the various bits. For example, in a simple synchronous protocol, the transmitter device may write a bit on the rising edge of the clock and the receiving device will then read this bit when it detects the falling clock edge. Note that the clock frequency does not need to be held constant.

Such synchronous interfaces are primarily intended for use over distances measured in centimetres rather than in metres and we will restrict our discussions here to the use of SPI to link the 8051 with compatible peripherals.

SPI PERIPHERAL

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a scheduler.
- The microcontroller in your application will be interfaced to one or more peripherals, such as a keypad, EEPROM, digital-to-analogue converter or similar device.
- Your microcontroller has hardware support for the SPI protocol.

Problem

Should you use the SPI bus to link your microcontroller to peripheral devices and, if so, how do you do so?

Background

There are five key features of SPI as far as the developer of embedded applications is concerned:

- SPI is a protocol designed to allow microcontrollers to be linked to a wide range of different peripherals – memory, displays, ADCs and similar devices – and requires (typically) three port pins for the bus, plus one chip-select pin per peripheral.
- There are many SPI-compatible peripherals available for purchase ‘off the shelf’.
- Increasing numbers of ‘Standard’ and ‘Extended’ 8051 devices have hardware support for SPI and we will make use of such facilities in this pattern.
- A common set of software code may be used with all SPI peripherals.
- SPI is compatible with time-triggered architectures and, as implemented in this book, is faster than I²C (largely due to the use of on-chip hardware support). Typical data transfer rates will be up to 5,000–10,000 bytes / second (with a 1 ms scheduler tick).

We provide some background to SPI in this section.

History

Serial peripheral interface (SPI) was developed by Motorola and included on the 68HC11 and other microcontrollers. Recently, this interface standard has been adopted by manufacturers of other microcontrollers. Increasing numbers of ‘Standard’ and ‘Extended’ 8051 devices (see Chapter 3) have hardware support for SPI and we will make use of such facilities in this pattern.

Basic SPI operation

SPI is often referred to as a three-wire interface. In fact, almost all implementations require two data lines, a clock line, a chip select line (usually one per peripheral device) and a common ground: this is at least four lines, plus ground.

The data lines are referred to as MOSI ('Master out Slave in') and MISO ('Master in Slave out').

The overall operation of SPI is easy to understand if you remember that the protocol is based on the use of two 8-bit shift registers, one in the Master, one in the Slave (Figure 24.1).

The key operation in SPI involves transferring a byte of data between the Master and the currently selected Slave device; simultaneously, a byte of data will be transferred back from the Slave to the Master.

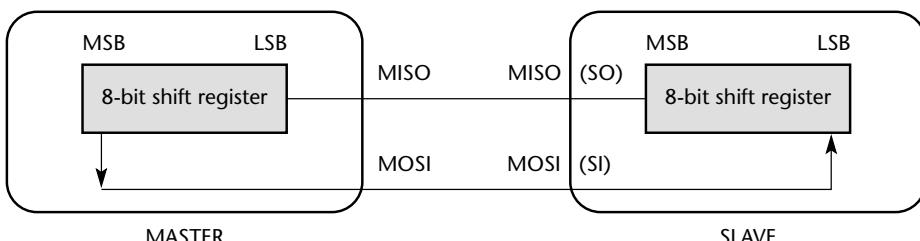


FIGURE 24.1 The two 8-bit shift registers at the heart of the SPI protocol

[Note: that some Slave devices (such as EEPROMs) label the data lines SI ('Slave in') and SO ('Slave out'). The SI line on the Slave is connected to the MOSI line on the Master and the SO line on the Slave is connected to the MISO line on the Master.]

Single-Master, multi-Slave

SPI is a single-Master, multi-Slave interface. The Master generates the clock signal. As far as we are concerned here, the microcontroller will form the Master device and one or more peripheral devices will act as Slaves.

Choice of clock polarities

SPI supports two clock polarities. With polarity 0, the clock line is low in the quiescent state: when active, the data to be sent are written on the rising clock edge and the data are read on the falling clock edge. With polarity 1, the clock line is high in the quiescent state: when active, the data to be sent are written on the falling clock edge and the data are read on the rising clock edge.

Polarity 0 is more widely used.

Maximum clock rate

The maximum clock rate for SPI is currently 2.1 MHz. Allowing for the fact that it takes eight clock cycles to transfer a byte of data and the fact that there are other

overheads too (instructions and addresses, for example), the maximum **data** transfer rates will be around 130,000 bytes per second.

Microwire

Note that the Microwire interface standard (developed by National Semiconductor) is similar to SPI, although the connection names, polarities and other details vary: Microwire is not discussed further in this book.

Solution

Should you use SPI?

In order to determine whether use of an SPI bus is appropriate in your time-triggered application, we consider some key questions that should be asked when considering the use of any communications protocol or related technique.

Main application areas

Although the SPI bus may be used, for example, to connect processors (usually microcontrollers) to one another or to other computer systems, its main application area is – like I²C – in the connection of standard peripheral devices, such as LCD panels or EEPROMs to microcontrollers.

Ease of development

SPI can be used to communicate with a large number and range of peripherals. By using the same protocol to talk to a range of devices, development efforts may be reduced.

Scalability

Each SPI Slave device requires a separate /CS (chip select) line from the Master node. This increases the number of pins required on the microcontroller if large numbers of peripherals are used.

Flexibility

Individual SPI-compatible microcontrollers may act as Master or Slave nodes. We consider only the use of the microcontroller as the Master node in this pattern.

Speed of execution and size of code

The maximum clock rate for SPI is currently 2.1 MHz.

As we will be using hardware-based SPI in this pattern, the code overhead will be small.

Cost

The cost of licence fees for use of the bus is included in the cost of the peripheral components which you purchase: in most circumstances, there are no additional fees to pay.

Note that this may not be the case if, for example, you are implementing an SPI peripheral (to be sold for connection to an SPI bus). If in doubt, contact Motorola for further details.

Choice of implementations and vendors

The SPI library presented here may be used only with 8051 devices that have hardware support for SPI.

Suitability for use in time-triggered applications

As we saw in Chapter 18, the RS-232 communication protocol is appropriate for use in time-triggered applications. This suitability arises because the task duration associated with transmission (and reception) of data on an RS-232 network is very short. Note that this transmission time is not directly linked to the baud rate of the network, largely because almost all members of the 8051 family have on-chip hardware support for RS-232, with the result that messages are transmitted and received ‘in the background’.

The situation with SPI is similar. Specifically, in this pattern, we are concerned with hardware-based SPI protocols. These typically impose a low software load and allow a short task duration. For example, if we consider the process of sending one byte of data to an SPI-based ROM chip (an example of this is presented in full later), then the total task duration is approximately 0.1 ms; note that this is considerably shorter than the equivalent operation using the I²C library presented in this book.

This task duration can be easily supported in a time-triggered application, even with 1 ms timer ticks.

How do you use SPI in a time-triggered application?

The discussions will centre around the Atmel AT89S53, a Standard 8051 device with on-chip SPI support. Note that hardware support provided by other manufacturers is very similar.

The AT89S53 SPI features include the following:

- Full-duplex, three-wire synchronous data transfer
- Master or Slave operation
- 1.5 MHz bit frequency (max.)
- LSB first or MSB first data transfer
- Four programmable bit rates
- End of transmission interrupt flag

- Write collision flag protection
- Wakeup from idle mode (Slave mode only)

The interconnection between Master and Slave CPUs with SPI is as shown in Figure 24.1.

The SCK pin is the clock output in the Master mode (the only mode we will use here). Writing to the SPI data register of the Master CPU starts the SPI clock generator and the data written shifts out of the MOSI pin and into the MOSI pin of the Slave CPU. After shifting one byte, the SPI clock generator stops, setting the end of transmission flag (SPIF). If both the SPI interrupt enable bit (SPIE) and the serial port interrupt enable bit (ES) are set, an interrupt is requested. We will not use these interrupt facilities.

The Slave Select input, SS/P1.4, is set low to select an individual SPI device as a Slave. When SS/P1.4 is set high, the SPI port is deactivated and the MOSI/P1.5 pin can be used as an input.

There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL (see Table 24.1).

Most of the features of the SPI interface in the AT89S53 are illustrated in the example that follows.

Hardware resource implications

With on-chip hardware support, SPI PERIPHERAL imposes a minimal software load.

Reliability and safety issues

The SPI protocol incorporates only minimal error-checking mechanisms: detection of data corruption (for example) during the transfer of information to or from a peripheral device must be carried out in software, if required.

Portability

This pattern requires hardware support for SPI: it cannot be used with microcontrollers without such support.

The discussions here are based on the Atmel 89S53. Use with other 8051 microcontrollers – including many Infineon 8051s – is straightforward.

Overall strengths and weaknesses

- ☺ SPI is supported by a wide range of peripheral devices.
- ☺ SPI requires (typically) three port pins for the bus, plus one chip-select pin per peripheral.
- ☺ Use of hardware-based SPI (as discussed here) facilitates the design of tasks with short durations; as a consequence the protocol is well matched to the

TABLE 24.1 The SPI control register (SPCR) in the Atmel 89S53

Bit	Name	Overview															
7	SPIE	SPI interrupt enable. Not used in this text Usually set SPIE = 0															
6	SPE	SPE = 1 enables the SPI channel. Note that this means that the upper nybble of Port 1 is not available for general-purpose I/O Usually set SPE = 1															
5	DORD	The data order. DORD = 0 selects 'most significant first' data transmission; DORD = 1 selects 'least significant first' data transmission Usually set DORD = 0															
4	MSTR	Master / Slave select. MSTR = 1 selects Master SPI mode We will only use Master mode in this text: set MSTR = 1															
3	CPOL	The clock polarity. When CPOL = 0, the SCK of the Master device is low when no data are being transmitted; when CPOL = 1, the SCK is high under these circumstances Usually set CPOL = 0															
2	CPHA	The clock phase. Refer to the Atmel documents for details Usually set CPHA = 0															
1	SPR1	The clock-rate select. These two bits control the SCK rate of the device, when it is configured as a Master															
0	SPR0	The relationship between SCK and the oscillator / resonator frequency ('Osc') is as follows: <table style="margin-left: 100px;"> <tr> <td>SPR1</td> <td>SPR0</td> <td>SCK</td> </tr> <tr> <td>0</td> <td>0</td> <td>Osc / 4</td> </tr> <tr> <td>0</td> <td>1</td> <td>Osc / 16</td> </tr> <tr> <td>1</td> <td>0</td> <td>Osc / 64</td> </tr> <tr> <td>1</td> <td>1</td> <td>Osc / 128</td> </tr> </table> Note that the AT89S53 has a maximum (SPI) bit rate of 1.5 MHz	SPR1	SPR0	SCK	0	0	Osc / 4	0	1	Osc / 16	1	0	Osc / 64	1	1	Osc / 128
SPR1	SPR0	SCK															
0	0	Osc / 4															
0	1	Osc / 16															
1	0	Osc / 64															
1	1	Osc / 128															

needs of time-triggered applications. Typical data transfer rates will be up to 5,000–10,000 bytes / second (with a 1 ms scheduler tick).

- ☺ A common set of software code may be used with all SPI peripherals.
- ☹ The use of this pattern is restricted to microcontrollers with hardware support for SPI.

Related patterns and alternative solutions

The use of this pattern is restricted to microcontrollers with hardware support for SPI: see [I²C PERIPHERAL](#) [page 494] for an alternative solution that provides very similar facilities without the need for hardware support.

Example: SPI core library

Our SPI implementation consists of an SPI ‘core’, to which additional libraries are added to match the needs of particular EEPROMs etc.

The core is given in Listings 24.1 to 24.3.

```
/*
-----*
Port.H (v1.00)

-----
'Port Header' (see Chap 10) for the SPI Core Library

-----*
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif

// ----- SPI_Core.C -----
// Create sbits for all required chip selects here
sbit SPI_CS = P1^4;

// NOTE: pins P1.4, P1.5, P1.6 and P1.7 also used - see text

/*
-----*
----- END OF FILE -----
-----*
-----*/
```

Listing 24.1 Part of the SPI core library for the Atmel AT89S53

```

/* -----
   SPI_Core.H (v1.00)

-----
   - See SPI_Core.C for details.

-*----- */

#include "Main.h"

// ----- Public function prototypes -----
void SPI_Init_AT89S53(const tByte);
tByte SPI_Exchange_Bytes(const tByte);

/* -----
   ----- END OF FILE -----
-*----- */

```

Listing 24.2 Part of the SPI core library for the Atmel AT89S53

```

/* -----
   SPI_Core.C (v1.00)

-----
   Core SPI library for Atmel AT89S53.

-*----- */

#include "Main.h"
#include "Port.h"

#include "SPI_Core.h"
#include "TimeoutH.H"

// ----- Public variable declarations -----
// The error code variable
//
// See Main.H for port on which error codes are displayed
// and for details of error codes
extern tByte Error_code_G;

/* -----
   SPI_Init_AT89S53()

   Set up the on-chip SPI module.

-*----- */

```

```

void SPI_Init_AT89S53(const tByte SPI_MODE)
{
    // SPI Control Register (SPCR)
    // Bit 7 = SPIE (enable SPI interrupt, if ES is also 1)
    // Bit 6 = SPE (enable SPI)
    // Bit 5 = DORD (data order, 1 for LSB first, 0 for MSB first)
    // Bit 4 = MSTR (1 for Master, 0 for Slave)
    // Bit 3 = CPOL (clock polarity, 1 = high when idle, 0 = low when idle)
    // Bit 2 = CPHA (transfer format)
    // Bit 1 = SPR1 (SPR0, SPR1 control the clock rate)
    // Bit 0 = SPR0
    SPCR = SPI_MODE;
}

/*-----*
 * SPI_Exchange_Bytes()
 *
 Exchange a byte of data with the Slave device.
 *-----*/
tByte SPI_Exchange_Bytes(const tByte OUT)
{
    // Write byte to SPI register (starts clock)
    // - these data will be transferred to the Slave device
    SPDR = OUT;

    // Wait until byte transmitted with 5ms timeout - START

    // Configure Timer 0 as a 16-bit timer for timeout
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 5ms
    TH0 = T_05ms_H; // See TimeoutH.H for T_ details
    TL0 = T_05ms_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    while (((SPSR & SPIF_) == 0) && (!TF0));

    TR0 = 0;

    if (TF0 == 1)
    {
        // SPI device timed out
        Error_code_G = ERROR_SPI_EXCHANGE_BYTES_TIMEOUT;
    }
}

```

```

    // Clear SPIF and WCOL
    SPSR &= 0x3F;

    // Return contents of SPI register
    // - these are the data from the Slave device
    return SPDR;
}

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 24.3 Part of the SPI core library for the Atmel AT89S53

Example: Using an SPI EEPROM (X25320 or similar)

In this example we present an SPI library allowing communication with an external EEPROM. In this case we have used a X25320 (4k × 8-bit) device, but any similar SPI EEPROM can be used without difficulty. Such devices are very useful as a means of storing non-volatile information such as passwords and similar information.

The hardware is based on the Atmel 89S53 (see Figure 24.2).

Note that in `main()` we do the following:

```
SPI_Init_AT89S53(0x51);
```

In this case, with a 12 MHz crystal on the board, this sets the SPI clock rate to 750,000 bits/second: this is roughly 100 bytes / millisecond, meaning that the duration of the basic data transfer tasks is approximately 0.01 ms.

The key files are given in Listings 24.4 to 24.6. You will also need the core SPI files presented earlier. As usual, all the files for this project are included on the CD.

```

/*-----*
----- Main.C (v1.00)
-----*

Simple test program for SPI code library.

Writes and reads Xicor X25320 (4k x 8-bit) EEPROM

*-----*/
#include "Main.h"
#include "SPI_Core.h"
#include "SPI_X25.h"
#include "Delay_T0.h"

// In this test program, we define the error code variable here.
tByte Error_code_G = 0;

```

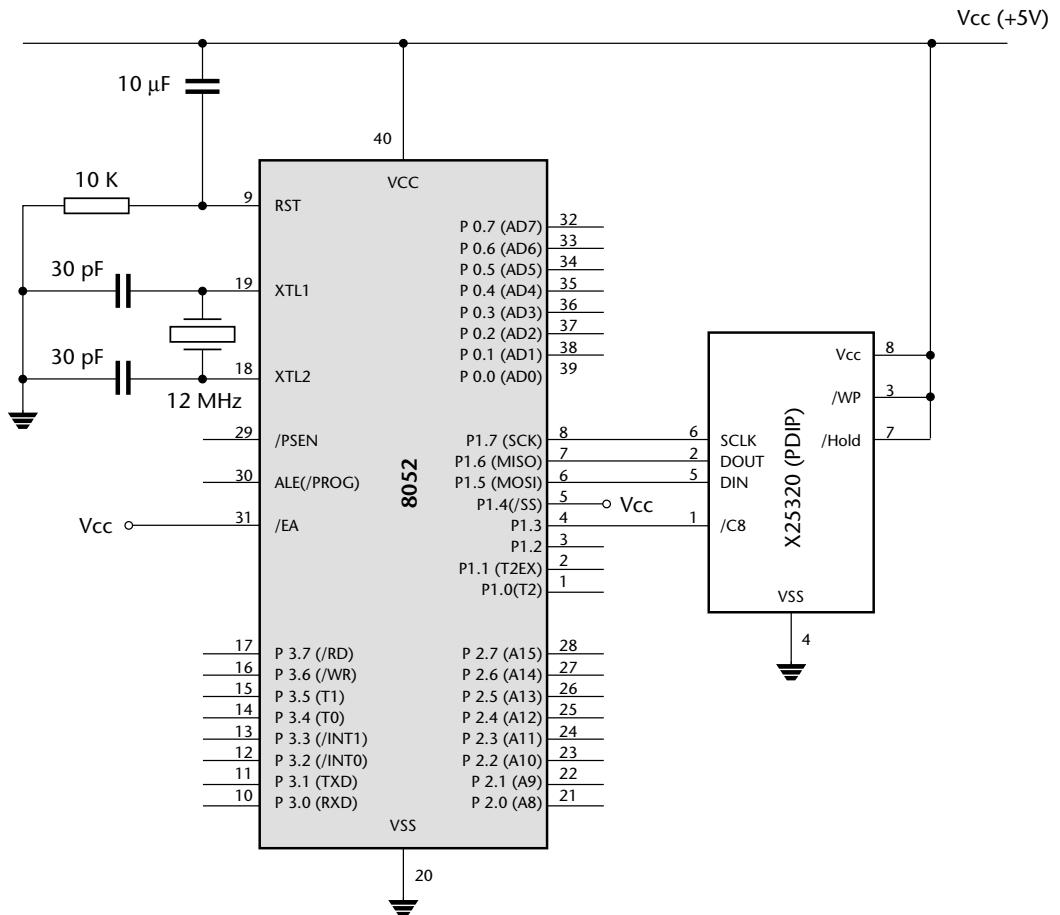


FIGURE 24.2 Connecting an SPI EEPROM to the Atmel AT89S53

```

void main(void)
{
    tByte Data1 = 0;
    tByte Data2 = 0;

    tWord Data_address = 0;

    // See text for details
    SPI_Init_AT89S53(0x51);

    while (1)
    {
        // Write to EEPROM
        SPI_X25_Write_Byte(Data_address, Data1++);
    }
}

```

```

    // Read back from EEPROM
    Data2 = SPI_X25_Read_Byte(Data_address);

    // Display value from EEPROM
    P2 = 255 - Data2;

    // Display error codes (if any)
    P3 = 255 - Error_code_G;

    if (++Data_address == 4095)
    {
        Data_address = 0;
    }

    Hardware_Delay_T0(1000);
}

}

/*-
----- END OF FILE -----
*/

```

Listing 24.4 Part of an example linking an SPI EEPROM to the Atmel AT89S53

```

/*-
----- SPI_x25.H (v1.00)

-----
- See SPI_x25.C for details.

----- */
#include "Main.h"

// ----- Public function prototypes -----
void SPI_X25_Write_Byte(const tWord, const tByte);
tByte SPI_X25_Read_Byte(const tWord);

/*-
----- END OF FILE -----
*/

```

Listing 24.5 Part of an example linking an SPI EEPROM to the Atmel AT89S53

```
/* ----- *-
 * SPI_X25.C (v1.00)
 *
-----*
Simple SPI library for Atmel AT89S53
- allows data storage on Xicor X25138 EEPROM (or similar)
* /
```

```
#include "Main.H"
#include "Port.h"

#include "SPI_Core.h"
#include "SPI_X25.h"
#include "TimeoutH.h"

// ----- Public variable declarations -----
// The error code variable
//
// See Main.H for port on which error codes are displayed
// and for details of error codes
extern tByte Error_code_G;

// ----- Private function prototypes -----
void SPI_Delay_T0(void);
void SPI_X25_Read_Status_Register(void);

/* ----- *-
 * SPI_X25_Write_Byte()
 *
Store a byte of data on the EEPROM.
* /
```

```
void SPI_X25_Write_Byte(const tWord ADDRESS, const tByte DATA)
{
    // 0. We check the status register
    SPI_X25_Read_Status_Register();

    // 1. Pin /CS is pulled low to select the device
    SPI_CS = 0;

    // 2. The 'Write Enable' instruction is sent (0x06)
    SPI_Exchange_Bytes(0x06);

    // 3. The /CS must now be pulled high
    SPI_CS = 1;

    // 4. Wait (briefly)
    SPI_Delay_T0();
```

```
// 5. Pin /CS is pulled low to select the device
SPI_CS = 0;

// 6. The 'Write' instruction is sent (0x02)
SPI_Exchange_Bytes(0x02);

// 7. The address we wish to read from is sent.
//     NOTE: we send a 16-bit address:
//           - depending on the size of the device, some bits may be ignored.
SPI_Exchange_Bytes((ADDRESS >> 8) & 0x00FF); // Send MSB
SPI_Exchange_Bytes(ADDRESS & 0x00FF);           // Send LSB

// 8. The data to be written is shifted out on MOSI
SPI_Exchange_Bytes(DATA);

// 9. Pull the /CS pin high to complete the operation
SPI_CS = 1;
}

/*-----*/
SPI_X25_Read_Byte()

Read a byte of data from the EEPROM.

*/
tByte SPI_X25_Read_Byte(const tWord ADDRESS)
{
    tByte Data;

    // 0. We check the status register
    SPI_X25_Read_Status_Register();

    // 1. Pin /CS is pulled low to select the device
    SPI_CS = 0;

    // 2. The 'Read' instruction is sent (0x03)
    SPI_Exchange_Bytes(0x03);

    // 3. The address we wish to read from is sent.
    //     NOTE: we send a 16-bit address:
    //           - depending on the size of the device, some bits may be ignored.
    SPI_Exchange_Bytes((ADDRESS >> 8) & 0x00FF);
    SPI_Exchange_Bytes(ADDRESS & 0x00FF);

    // 4. The data requested is shifted out on S0 by sending a dummy byte
    Data = SPI_Exchange_Bytes(0x00);

    // 5. We pull the /CS pin high to complete the operation
    SPI_CS = 1;
```

```

        return Data; // Return SPI data byte
    }

/*-----*/
SPI_X25_Read_Status_Register()

We read the status register only to make sure that previous
'Write' operations (if any) are now complete.
To do this, we check the WIP ('Write In Progress') flag.

*** NB: THE INTERNAL EEPROM WRITE OPERATION TAKES UP 10ms ***
*** Schedule writes (and reads after writes) at sensible ***
*** intervals - or you will get task jitter ***

We timeout after ~15ms.

Uses T0 for (hardware) timeout - see Chapter 15.

*/
void SPI_X25_Read_Status_Register()
{
    tByte Data;
    bit Wip;

    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 15ms
    TH0 = T_15ms_H; // See TimeoutH.H for T_ details
    TL0 = T_15ms_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    do {
        // 0. Pin /CS is pulled low to select the device
        SPI_CS = 0;

        // 1. The 'RDSR' instruction is sent (0x05)
        SPI_Exchange_Bytes(0x05);

        // 2. The contents of the ROM status register are read
        Data = SPI_Exchange_Bytes(0x00);

        // 3. Pull the /CS pin high to complete the operation
        SPI_CS = 1;

        // Check the WIP flag
        Wip = (Data & 0x01);
    } while ((Wip != 0) && (TF0 != 1));
}

```

```

TR0 = 0;

if (TF0 == 1)
{
    // ROM timed out
    Error_code_G = ERROR_SPI_X25_TIMEOUT;
}

/*
 *-----*
SPI_Delay_T0()

A delay of approx 50 µs using 'timeout' code.

-*-----*/
void SPI_Delay_T0(void)
{
    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

    ET0 = 0; // No interrupts

    // Simple timeout feature - approx 50 µs
    TH0 = T_50micros_H; // See TimeoutH.H for T_ details
    TL0 = T_50micros_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer

    while (!TF0);

    TR0 = 0;
}

/*
-----*
----- END OF FILE -----
-*-----*/

```

Listing 24.6 Part of an example linking an SPI EEPROM to the Atmel AT89S53

Example: Using an SPI ADC

See Chapter 32, where an SPI-based analogue-to-digital converter is used.

Further reading

Time-triggered architectures for multiprocessor systems

In Part F, we turn our attention to multiprocessor applications. As we will see, an important advantage of the time-triggered (co-operative) scheduling architecture is that it is inherently scalable and that its use extends naturally to multiprocessor environments.

In Chapter 25, we consider some of the advantages – and disadvantages – that can result from the use of multiple processors. We then go on to introduce the shared-clock scheduler and illustrate how this operating environment may be used to create efficient time-triggered applications involving two or more microcontrollers.

In Chapter 26, we consider shared-clock schedulers that are kept synchronized through the use of external interrupts on the Slave microcontrollers. These simple schedulers impose little memory, CPU or hardware overhead. However, they are generally suitable only for use for system prototyping or where the Master and Slave microcontrollers are on the same circuit board.

In Chapter 27, we describe in detail techniques for creating shared-clock schedulers that can link multiple controllers over large distances, using the ubiquitous RS-232 and RS-485 protocols and suitable transceiver hardware. In addition, we demonstrate that the same techniques may be applied at short distances without the need for any transceiver components.

Finally, in Chapter 28, we consider shared-clock schedulers that communicate via the powerful ‘controller area network’ (CAN) bus. The CAN bus is now very widely used in automotive, industrial and other sectors: it forms an excellent platform for reliable, multi-processor applications, particularly where there is a need to move comparatively large amounts of data around the network. Like UART-based techniques, the CAN protocol is suitable for use in both local and distributed systems.

chapter **25**

An introduction to shared-clock schedulers

In this chapter, we consider one additional important characteristic of embedded applications: the use of multiple processors. As we will see, the scheduler architecture introduced in previous chapters may be extended without difficulty in order to support such applications.

25.1 Introduction

Despite the diverse nature of the embedded applications we have discussed in previous chapters, each of these has involved only a single microcontroller. By contrast, many modern embedded systems contain more than one processor. For example, a modern passenger car might contain some 40 such devices (Leen *et al.*, 1999), controlling brakes, door windows and mirrors, steering, air bags and so forth. Similarly, an industrial fire detection system might typically have 200 or more processors, associated, for example, with a range of different sensors and actuators.

We begin this chapter by considering two key advantages of multiprocessor systems and then go on to introduce a form of co-operative scheduler – the shared-clock scheduler – that can help the developer get the most from such a design.

We conclude by discussing some of the reliability implications of multiprocessor implementations.

25.2 Additional CPU performance and hardware facilities

Suppose we require a microcontroller with the following specification:

- 60+ port pins
- Six timers

- Two USARTS
- 128 kbytes of ROM
- 512 bytes of RAM
- A cost of around \$1.00 (US)

We can meet many of these requirements with an **EXTENDED 8051** [page 46]: however, this will typically cost at the very least around 5–10 times the \$1.00 price we require. By contrast, the ‘microcontroller’ in Figure 25.1 matches these requirements very closely.

Figure 25.1 shows two standard 8051 microcontrollers linked together by means of a single port pin: as we demonstrate in **SCI SCHEDULER (TICK)** [page 554], this type of scheduler can be created with a minimal software and hardware load. The result is a flexible environment with 62 free port pins, five free timers, two USARTs and so on. Note that further microcontrollers may be added without difficulty and the communication over a single wire (plus ground) will ensure that the tasks on all processors are perfectly synchronized.

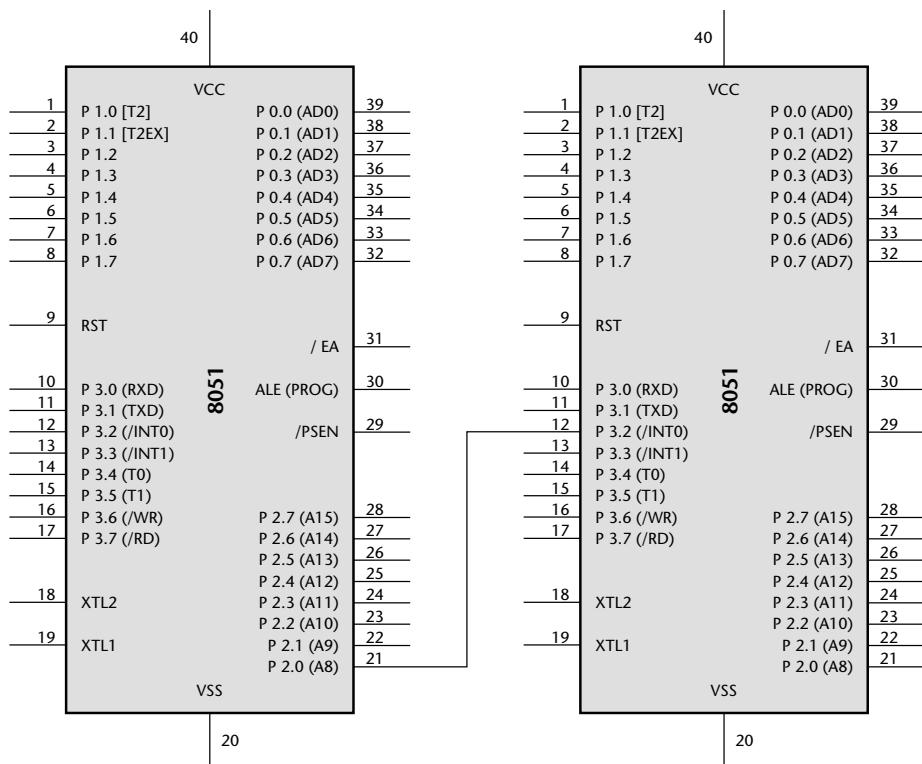


FIGURE 25.1 Linking two 8051 microcontrollers using a simple shared-clock scheduler

Of course, in addition to the features listed, the two-microcontroller design also has two CPUs. In many (but not all) cases, this can allow you to perform tasks more quickly or to carry out more tasks within a given time interval.

The patterns **LONG TASK** [page 716] and **DOMINO TASK** [page 720], sometimes used in conjunction with **DATA UNION** [Page 712], encapsulate effective software architectures that allow you to get the best performance out of such a multiprocessor design.

25.3 The benefits of modular design

Suppose we are required to produce a range of different clocks, with various forms of display (Figure 25.2).



FIGURE 25.2 Various displays for a range of clock products

Some of the clocks may have different features (for example, the ability to set an alarm), but the key tasks are the same in all cases: to keep accurate track of the time and display this information on a display.

In some circumstances, it may be useful to distribute the application over two modules, each with a separate microcontroller. The first module would deal with the basic timekeeping and time adjustment facilities; the second module would provide support for the different displays, such as an LCD driver or a stepper motor. This approach may provide economic benefits, since it allows us to produce many thousands of the basic timekeeping modules at low cost. We can then produce different displays, as required, to match the needs of particular customers.

This type of modular approach is very common in the automotive industry where increasing numbers of microcontroller-based modules are used in new vehicle designs.

Consider another example. Suppose we have a data-acquisition system with a single processor and a number of distributed (simple) sensors (Figure 25.3).

In this arrangement, if the cable to (say) Sensor 1 is damaged, then no data will be obtained from this sensor until the link is repaired; worse, if an inappropriate data representation has been used, the acquisition system may not even be aware that the link has been damaged.

Consider now an alternative solution using ‘intelligent’ sensors (Figure 25.4).

In this version of the system, Sensor 1 is (we assume) in very close proximity to a microcontroller ('MCU A'); together, these two components make up our ‘intelligent’ sensor. Communication between the intelligent sensor and the main acquisition

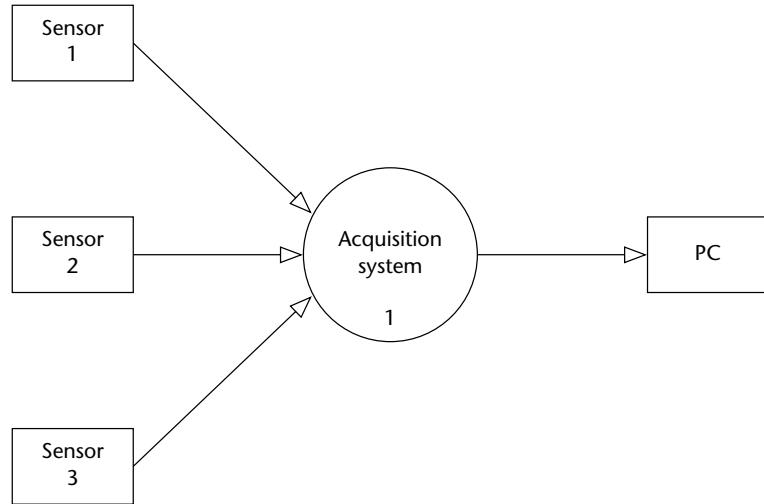


FIGURE 25.3 An outline design for a simple data acquisition system

system will be arranged in such a way that a broken connection is easily detected. In these circumstances, we could arrange for the sensor node to continue to collect data while the link is repaired, with the result that, following a repair, the ‘missing’ data could be recovered by the main acquisition board.

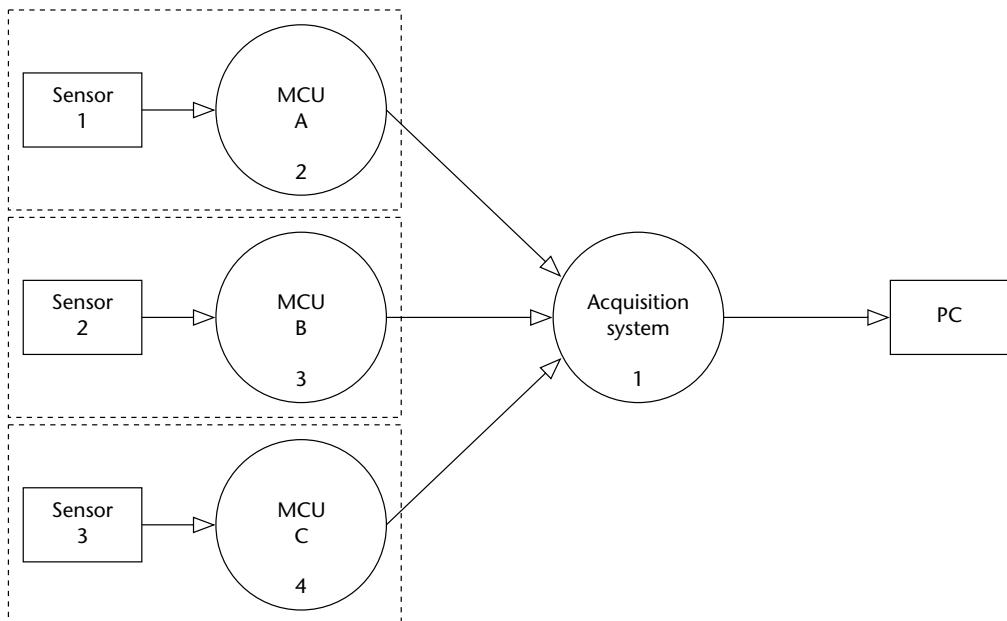


FIGURE 25.4 An outline design for an ‘intelligent’ data-acquisition system

This type of ‘intelligent’ node behaviour can be very useful in many circumstances. For example, in the A310 Airbus, the slat and flap control computers form an ‘intelligent’ actuator subsystem. If an error is detected during landing, the wings are set to a safe state and then the actuator subsystem shuts itself down (Burns and Wellings, 1997, p. 102).

As we will see in the remaining chapters in Part F, most S-C schedulers support the creation of backup nodes, which may be made ‘intelligent’ if this is required.

25.4 How do we link more than one processor?

We will now begin to consider some of the challenges that face developers who wish to design multiprocessor applications. We begin with a fundamental problem:

- How do we keep the clocks on the various nodes synchronized?

We then go on to address two further problems that can arise with many such systems:

- How do we transfer data between the various nodes?
- How does one node check for errors on the other nodes?

As we will see, by using a shared-clock (S-C) scheduler, we can address all three problems. Moreover, the time division multiple access (TDMA) protocol we employ to achieve this is a ‘natural extension’ (Burns and Wellings, 1997, p. 484), to the time-triggered architectures for single-processor systems which we have described in earlier parts of this book.

Synchronizing the clocks

Why do we need to synchronize the tasks running on different parts of a multiprocessor system?

Consider a simple example. Suppose we are developing a portable traffic-light system designed to control the flow of traffic on a narrow road while repairs are carried out. The system is to be used at both ends of the area of road works and will allow traffic to move in only one direction at a time (Figure 25.5).

The conventional ‘red’, ‘amber’ and ‘green’ bulbs will be used on each node, with the usual sequencing (Figure 25.6).

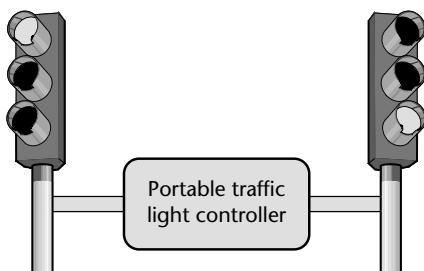


FIGURE 25.5 A portable traffic-light control system

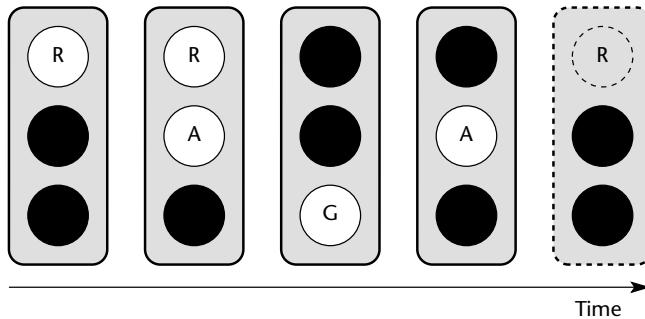


FIGURE 25.6 The required light sequence for the traffic-light example

We will assume that there will be a microcontroller at each end of the traffic-light application to control the two sets of lights. We will also assume that each microcontroller is running a scheduler and that each is driven by an independent crystal oscillator circuit.

The problem with this arrangement is that the schedulers on the two microcontrollers are likely to get quickly 'out of sync'. This will happen primarily because the two boards will never run at exactly the same temperature and, therefore, the crystal oscillators will operate at different rates.

This can cause real practical difficulties. In this case, for example, we run the risk that both sets of traffic lights will show 'green' at the same time, a fact likely to result, quickly, in an accident.

The S-C scheduler tackles this problem by sharing a single clock between the various processor boards, as illustrated schematically in Figure 25.7.

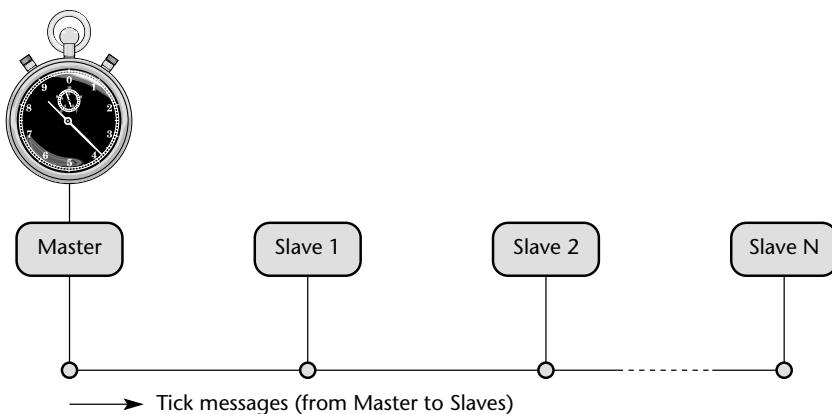


FIGURE 25.7 The simplest form of S-C scheduler

[Note: this involves the sending of tick messages from the Master to the Slave(s) at regular intervals. All S-C schedulers provide this facility.]

Here we have one, accurate, clock on the Master node in the network. This clock is used to drive the scheduler in the Master node in exactly the manner discussed in Part C.¹

The Slave nodes also have schedulers: however, the interrupts used to drive these schedulers are derived from ‘tick messages’ generated by the Master (Figure 25.8). Thus, in a CAN-based network (for example), the Slave node will have a S-C scheduler driven by the ‘receive’ interrupts generated through the receipt of a byte of data sent by the Master.

In the case of the traffic lights, changes in temperature will, at worst, cause the lights to cycle more quickly or more slowly: the two sets of lights will not, however, get out of sync.

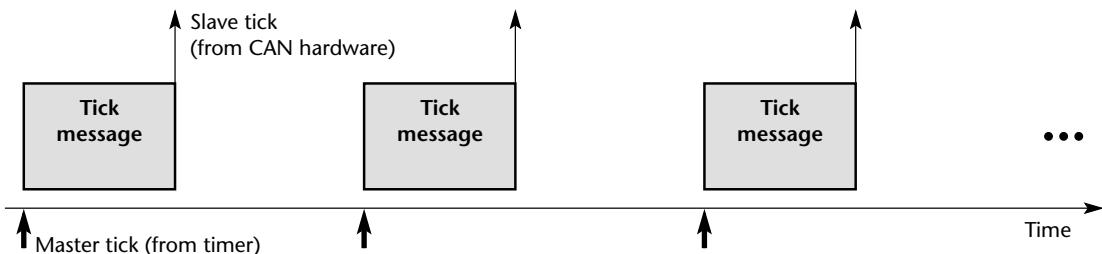


FIGURE 25.8 Communication between the Master and Slave nodes in the simplest (CAN-based) S-C network

[Note: This involves the transmission of these tick messages, triggered by the timer ticks in the Master. The receipt of the regular messages by the Slave is the source of the interrupt used to drive the scheduler in this node: the Slave does **not** have a separate timer-based interrupt. By default, there is a slight delay between the tick generation on the Master and Slave nodes in some networks (notably CAN- and UART-based networks). This delay is short (a fraction of a millisecond in most cases); equally important, it is predictable and fixed. We discuss how to calculate and, where necessary, eliminate this delay in Chapters 27 and 28.]

Transferring data

So far we have focused on synchronizing the schedulers in individual nodes. In many applications, we will also need to transfer data between the tasks running on different processor nodes.

To illustrate this, consider again the traffic-light controller. Suppose that a bulb blows in one of the light units. When a bulb is missing, the traffic control signals are ambiguous: we therefore need to detect bulb failures on each node and, having detected a failure, notify the other node that a failure has occurred. This will allow us, for example, to extinguish all the (available) bulbs on both nodes or to flash all the bulbs on both nodes: in either case, this will inform the road user that something is amiss and that the road must be negotiated with caution.

1. Where necessary, a temperature-compensated or satellite-based clock may be used in the Master node to ensure accurate timing throughout the network: see Chapter 4.

If the light failure is detected on the Master node, then this is straightforward. As we discussed earlier, the Master sends regular tick messages to the Slave, typically once per millisecond. These tick messages can – in most S-C schedulers – include data transfers: it is therefore straightforward to send an appropriate tick message to the Slave to alert it to the bulb failure.

To support the transfer of data from the Slave to the Master, we need an additional mechanism: this is provided through the use of ‘acknowledgement’ messages (Figure 25.9). The end result is a simple and predictable ‘time division multiple access’ (TDMA) protocol (e.g. see Burns and Wellings, 1997), in which acknowledgement messages are interleaved with the tick messages. For example, Figure 25.10 shows the mix of tick and acknowledgement messages that will be transferred in a typical two-Slave (CAN) network.

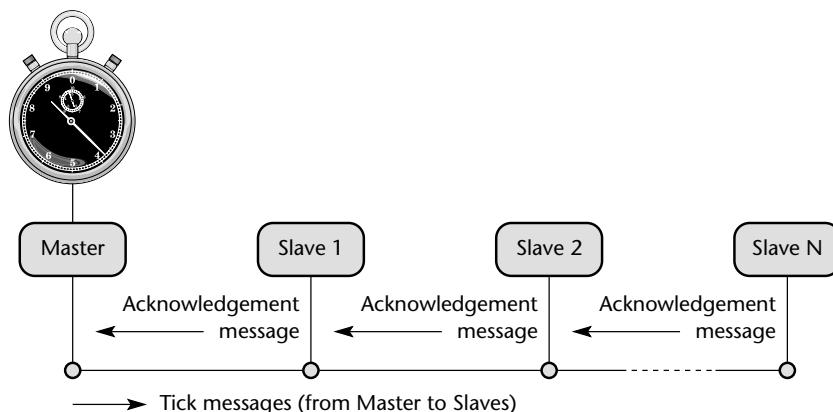


FIGURE 25.9 Most S-C schedulers support both ‘tick’ messages (sent from the Master to the Slaves) and ‘acknowledgement’ messages (sent by the Slaves to the Master)

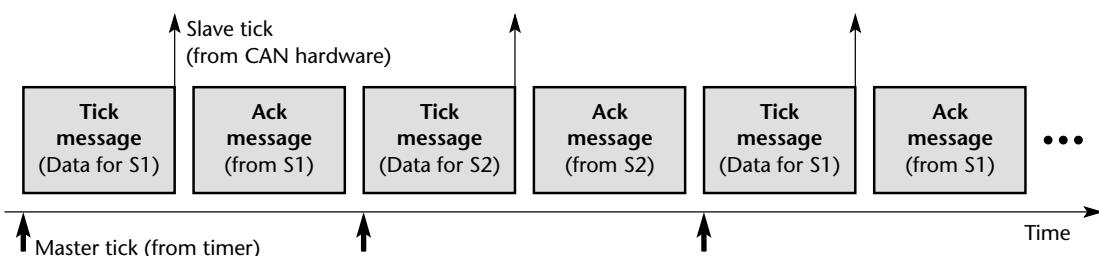


FIGURE 25.10 Communication between the Master and Slave nodes in a S-C (CAN) scheduler with one Master and two Slave nodes

[Note: Here the tick messages update the scheduler in all Slave nodes. However, each tick message will require an acknowledgement message from only one node: this acknowledgement will be sent in the period before the next tick message. This, again, meets the requirements of this simple and predictable (TDMA) network protocol.]

Note that, in a shared-clock scheduler, *all* data transfers are carried out using the interleaved tick and acknowledgement messages: **no additional messages are permitted on the bus**. As a result, we are able to pre-determine the network bandwidth required to ensure that all messages are delivered precisely on time.

Detecting network and node errors

Consider the traffic light control system one final time. We have already discussed the synchronization of the two nodes and the mechanisms that can be used to transfer data. What we have not yet discussed is problems caused by the failure of the network hardware (cabling, tranceivers, connectors and so on) or the failure of one of the network nodes.

For example, a simple problem that might arise is that the cable connecting the two sets of lights becomes damaged or is severed completely. This is likely to mean that the ‘tick messages’ from the Master are not received by the Slave, causing the slave to ‘freeze’. If the Master is unaware that the Slave is not receiving messages then again we run the risk that the two sets of lights will both, simultaneously, show green, with the potential risk of a serious accident (see Figure 25.11).

The S-C scheduler deals with this potential problem using the error detection and recovery mechanisms which we discuss in the next section.

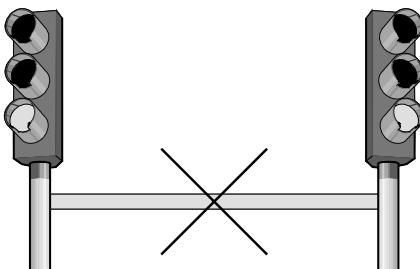


FIGURE 25.11 A portable traffic-light system in a dangerous state, as a result of a cable failure

Detecting errors in the Slave(s)

The use of a shared-clock scheduler makes it straightforward for the Slave to detect errors very rapidly. Specifically, because we know from the design specification that the Slave should receive ticks at (say) 1 ms intervals, we simply need to measure the time interval between ticks; if a period greater than 1 ms elapses between ticks, we conclude that an error has occurred.

In many circumstances an effective way of achieving this is to set a watchdog timer² in the Slave to overflow at a period slightly longer than the tick interval. Under normal circumstances, the ‘update’ function in the Slave will be invoked by the arrival of each tick and this update function will, in turn, refresh the watchdog timer. If a tick is not received, the timer will overflow and we can invoke an appropriate error-handling routine.

We discuss the required error-handling functions further next.

Detecting errors in the Master

Detecting errors in the Master node requires that each Slave sends appropriate acknowledgement messages to the Master at regular intervals (see Figure 25.10). A simple way of achieving this may be illustrated by considering the operation of a particular one-Master, ten-Slave network:

- The Master node sends tick messages to all nodes, simultaneously, every millisecond; these messages are used to invoke the update function in all Slaves every millisecond.
- Each tick message will, in most schedulers, be accompanied by data for a particular node. In this case, we will assume that the Master sends tick messages to each of the Slaves in turn; thus, each Slave receives data in every tenth tick message (every 10 milliseconds in this case).
- Each Slave sends an acknowledgement message to the Master only when it receives a tick message with its ID; it does **not** send an acknowledgement to any other tick messages.

As mentioned previously, this arrangement provides the predictable bus loading that we require and a means of communicating with each Slave individually. It also means that the Master is able to detect whether or not a particular Slave has responded to its tick message.

Handling errors detected by the Slave

We will assume that errors in the Slave are detected with a watchdog timer. To deal with such errors, the shared-clock schedulers presented in this book all operate as follows:

- Whenever the Slave node is reset (either having been powered up or reset as a result of a watchdog overflow), the node enters a ‘safe state’.
- The node remains in this state until it receives an appropriate series of ‘start’ commands from the Master.

This form of error handling is easily produced and is effective in most circumstances.

One important alternative form of behaviour involves converting a Slave into a Master node in the event that failure of the Master is detected. This behaviour can be

2. See **HARDWARE WATCHDOG** [page 217]

very effective, particularly on networks (such as CAN networks) which allow the transmission of messages with a range of priority levels. We will not consider this possibility in detail in the present edition of this book.

Handling errors detected by the Master

Handling errors detected by the Slave node(s) is straightforward in a shared-clock network. Handling errors detected by the Master is more complicated. We consider and illustrate three main options in this book:

- The ‘Enter safe state then shut down’ option
- The ‘Restart the network’ option
- The ‘Engage backup Slave’ option

We consider each of these options now.

Enter a safe state and shut down the network

Shutting down the network following the detection of errors by the Master node is easily achieved. We simply stop the transmission of tick messages by the Master. By stopping the tick messages, we cause the Slave(s) to be reset too; the Slaves will then wait (in a safe state). The whole network will therefore stop, until the Master is reset.

This behaviour is the most appropriate behaviour in many systems in the event of a network error, **if a ‘safe state’ can be identified**. This will, of course, be highly application dependent.

For example, we have already mentioned the A310 Airbus’ slat and flap control computers which, on detecting an error during landing, restore the wing system to a safe state and then shut down. In this situation, a ‘safe state’ involves having both wings with the same settings; only asymmetric settings are hazardous during landing (Burns and Wellings, 1997, p.102).

The strengths and weaknesses of this approach are as follows:

- ☺ **It is very easy to implement.**
- ☺ **It is effective in many systems.**
- ☺ **It can often be a ‘last line of defence’ if more advanced recovery schemes have failed.**
- ☹ It does not attempt to recover normal network operation or to engage backup nodes.

This approach may be used with any of the networks we discuss in this book (interrupt based, UART based or CAN based). We illustrate the approach in detail in Chapter 26.

Reset the network

Another simple way of dealing with errors is to reset the Master and, hence, the whole network. When it is reset, the Master will attempt to re-establish communication with each Slave in turn; if it fails to establish contact with a particular Slave, it will attempt to connect to the backup device for that Slave.

This approach is easy to implement and can be effective. For example, many designs use 'N-version' programming to create backup versions of key components.³ By performing a reset, we keep all the nodes in the network synchronized and we engage a backup Slave (if one is available).

The strengths and weaknesses of this approach are as follows:

- ☺ **It allows full use to be made of backup nodes.**
- ☹ It may take time (possibly half a second or more) to restart the network; even if the network becomes fully operational, the delay involved may be too long (for example, in automotive braking or aerospace flight-control applications).
- ☹ With poor design or implementation, errors can cause the network to be continually reset. This may be rather less safe than the simple 'enter safe state and shut down' option.

This approach may be used with any of the UART- or CAN-based networks we discuss in this book. We illustrate the approach in detail in Chapter 27.

Engage a backup Slave

The third and final recovery technique we discuss in the present edition of this book is as follows. If a Slave fails, then – rather than restarting the whole network – we start the corresponding backup unit.

The strengths and weaknesses of this approach are as follows:

- ☺ **It allows full use to be made of backup nodes.**
- ☺ **In most circumstances it takes comparatively little time to engage the backup unit.**
- ☹ The underlying coding is more complicated than the other alternatives discussed in this book.

This approach may be used with any of the UART- or CAN-based networks we discuss in this book. We illustrate the approach in detail in Chapter 28.

25.5 Why additional processors may not always improve reliability

It is very important to appreciate that – without due care – increasing the numbers of processors in a network can have a detrimental impact on overall system reliability.

3. See Leveson, 1995, for detailed discussion of the strengths and weaknesses of N-version programming techniques.

It is not difficult to see why this is the case. For example, we will ignore the possibility of failures in the links between processors and the need for a more complex (software) operating environment. Instead, we will assume that a network has 100 microcontrollers and that each of these devices is 99.99% reliable. As a result, a multiprocessor application which relies on the correct, simultaneous operation of all 100 nodes will have an overall reliability of $99.99\% \times 99.99\% \times 99.99\% \dots$ This is 0.9999¹⁰⁰, or approximately 37%. **This is a huge decrease in reliability:** a 99.99% reliable device might be assumed to fail once in 10,000 years, while the corresponding 37% reliable device would then be expected to fail approximately every 18 months.⁴

It is only where the **increase in reliability** resulting from the shared-clock design outweighs the **reduction in reliability** known to arise from the increased system complexity that an overall increase in system reliability will be obtained. Unfortunately, making predictions about the costs and benefits (in reliability terms) of any complex design feature remains – in most non-trivial systems – something of a black art.

For example, consider the use of ‘redundant nodes’ as discussed earlier. Specifically, suppose we are developing an automotive cruise-control system (Figure 25.12).

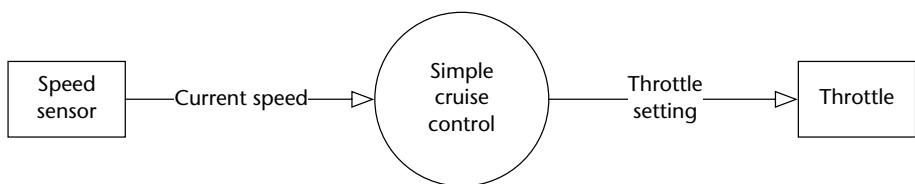


FIGURE 25.12 An outline design for an automotive cruise-control system

The cruise-control application has clear safety implications: if the application suddenly fails and sets the car at full throttle, fatalities may result. As a result, we may wish to use two microcontroller-based nodes in order to provide a backup unit in the event that the first node fails (Figure 25.13).

This can be an effective design solution: for example, if we have a network with two essentially identical nodes and we are able to activate the second node when the first one fails then it seems likely that this will improve the overall system reliability. In effect, this is the approach used to good effect in many aircraft flight-control applications where the ‘main’, ‘backup’ and ‘limp home’ controllers may be switched in, as required, by the pilot or co-pilot (e.g. Storey, 1996).

However, the mere presence of redundant networks does not itself guarantee increased reliability. For example, in 1974, in a Turkish Airlines DC-10 aircraft, the cargo door opened at high altitude. This event caused the cargo hold to depressurize, which in

4. See Storey, 1996, for a rather more rigorous analysis

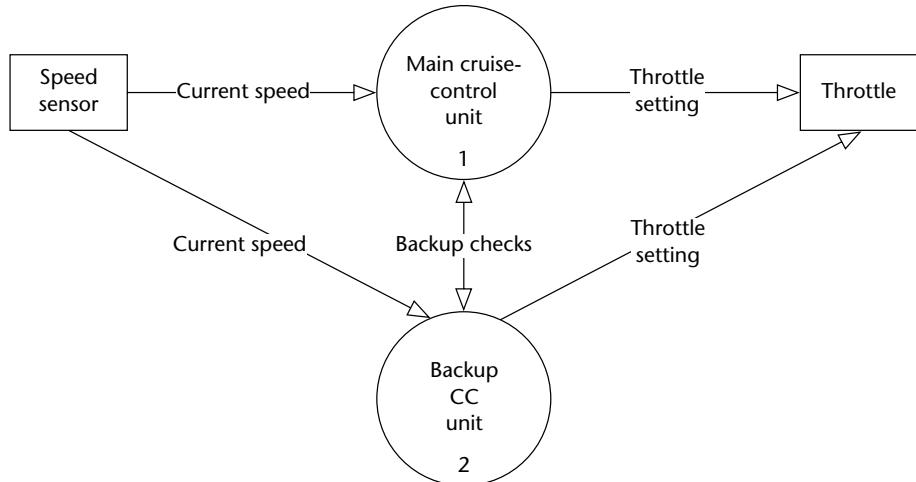


FIGURE 25.13 An outline design for an automotive cruise-control system with backup control unit

turn caused the cabin floor to collapse. The aircraft contained two (redundant) control lines, in addition to the main control system – but all three lines were under the cabin floor. Control of the aircraft was therefore lost and it crashed outside Paris, killing 346 people (Bignell and Fortune, 1984, pp. 143–4; Leveson, 1995, pp. 50 and 434).

In addition, in many embedded applications, there is either no human operator in attendance or the time available to switch over to a backup node (or network) is too small to make human intervention possible. In these circumstances, if the component required to detect the failure of the main node and switch in the backup node is complicated (as often proves to be the case), then this ‘switch’ component may itself be the source of severe reliability problems (see Leveson, 1995).

Note that these comments should not be taken to mean that multiprocessor designs are inappropriate for use in high-reliability applications. Multiple processors can be (and are) safely used in such circumstances. However, all multiprocessor developments must be approached with caution and must be subject to particularly rigorous design, review and testing.

25.6 Conclusions

In this chapter we have considered some of the advantages and disadvantages that can result from the use of multiple processors. We also introduced the shared-clock scheduler and sought to demonstrate that this operating environment may be used to create efficient time-triggered applications involving two or more microcontrollers.

We will provide detailed descriptions of a range of shared-clock schedulers in the chapters that follow.

Shared-clock schedulers using external interrupts

Introduction

In this chapter we present two simple implementations of the shared-clock scheduler architecture. In each case the Master and Slave devices are kept in synchrony by means of pulses (generated by the Master) that invoke interrupts on the Slave microcontrollers: these interrupts, in turn, drive the ‘update’ functions in the Slave schedulers.

These simple schedulers are very powerful and impose little memory, CPU or hardware overhead.

Note that the use of external interrupts to convey tick messages means that the resulting applications are at particular risk from sources of electromagnetic interference (EMI). As a result, these schedulers are only suitable for ‘local’ networks: that is, for applications containing multiple microcontrollers which are displaced by no more than (at most) a few centimetres; in most cases, the microcontrollers will be contained in the same, shielded system box and mounted on the same PCB.

SCI SCHEDULER (TICK)

Context

- You are developing an embedded application using more than one 8051 microcontroller.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you schedule tasks on a local network of two (or more) 8051 microcontrollers connected together via an interrupt link?

Background

To summarize the material in Chapter 25, the key characteristics of S-C schedulers are as follows:

- Scheduling is co-operative.
- The network includes one Master and N Slaves, where $N \geq 1$.
- The Master is attached to an accurate oscillator. Overflow of an internal timer (driven by this oscillator) runs the Master scheduler and in turn causes 'tick messages' to be sent to the Slaves.
- The Slaves run their schedulers based on interrupts generated through the arrival of the tick messages from the Master. That is, the Slaves do not use internal timers to drive the schedulers.
- Each Slave runs a (low-accuracy) watchdog timer to detect failure of the Master device.

In addition, *most* S-C schedulers have the following characteristics:

- The Master expects to have received an acknowledgement from every Slave after N messages have been sent. That is, each Slave acknowledges 1 out of N messages.
- The maximum delay before the Slaves detect the loss of the Master can be as little as two tick periods (this will depend on the setting of the Slaves' internal watchdog timers).
- The maximum delay before the Master detects the loss of all Slaves (for example, following a complete network failure) will generally be one tick period.
- The maximum delay before the Master detects the loss of a particular Slave (for example, following failure of a single node) will generally be N tick periods.

Solution

We discuss two possible implementations of a simple shared-clock scheduler in this section.

Basic technique

As we discussed in Chapter 25, the first problem we face when scheduling tasks across more than one microcontroller is ensuring that the tasks on the various boards remain synchronized. A shared-clock (S-C) scheduler tackles this problem by sharing a single clock between the various processor board, as illustrated schematically in Figure 26.1.

Here we have one, accurate, clock on the Master node in the network. This clock is used to drive the scheduler in the Master node in the normal way: that is, by interrupts generated through the overflow of one of the on-chip timers. Thus, the ‘update’ function in the Master units typically looks something like this:

```
void SCI_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    ...
}
```

The Slave nodes also have schedulers: however, the interrupts used to drive these schedulers are derived not from timers but, instead, from the ‘tick messages’ generated by the Master. There are various ways in which messages from the Master may be used to generate interrupts, as we will see throughout Part F. In the case of all of the schedulers in the present chapter, the source of the interrupts is a change in the voltage level at one of the ‘external interrupt’ input pins, such as Pin 3.2 (Interrupt 0):

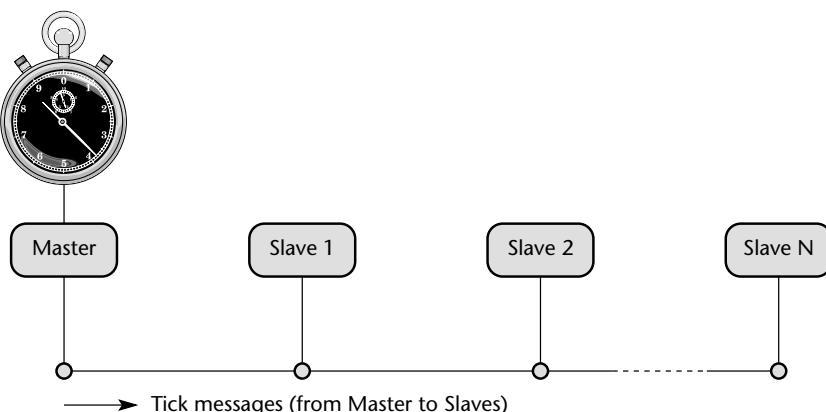


FIGURE 26.1 The simplest form of S-C scheduler

[Note: This involves the sending of tick messages from the Master to the Slave(s) at regular intervals. All S-C schedulers provide this facility.]

```

void SCI_SLAVE_Update(void) interrupt INTERRUPT_EXTERNAL_0
{
    ...
}

```

Generating this voltage level change at the required time is straightforward:

- In the ‘update’ function in the Master, we change a port pin (say Pin 2.0) from Logic 1 to Logic 0.
- In the Slave, we configure the ‘update’ function in the scheduler so that it is invoked by an external interrupt on, say, Pin 3.2.
- We connect the Master and Slave microcontrollers together, as illustrated in Figure 26.2.

The two microcontrollers will now operate precisely in step.

Note that there is no reason why this approach cannot be used with more than one Slave (Figure 26.3).

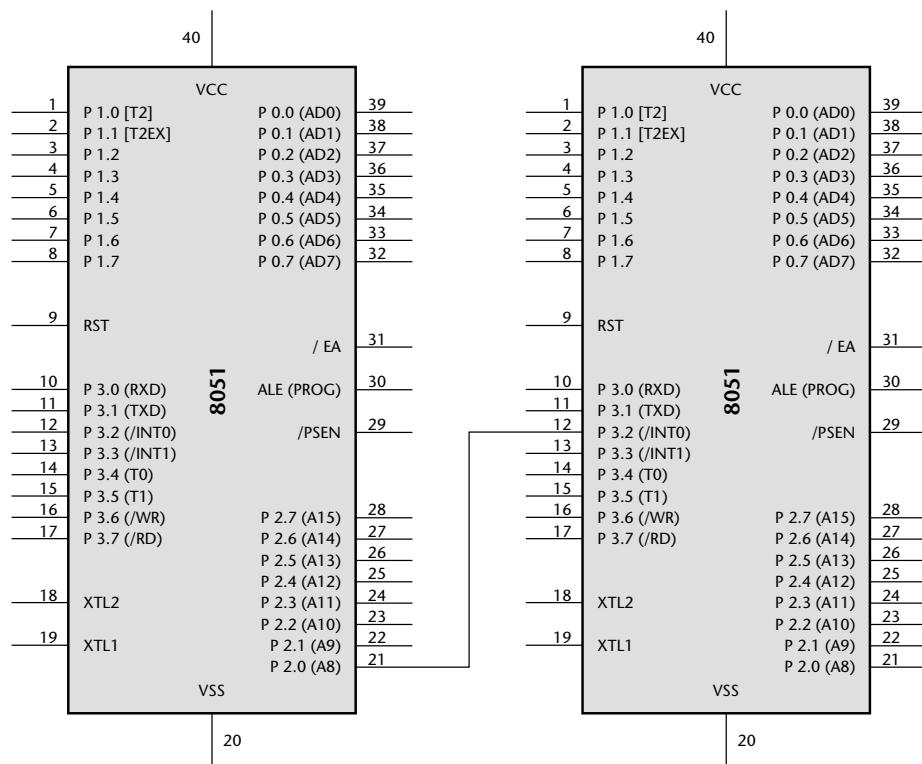


FIGURE 26.2 A very simple shared-clock (interrupt) scheduler.

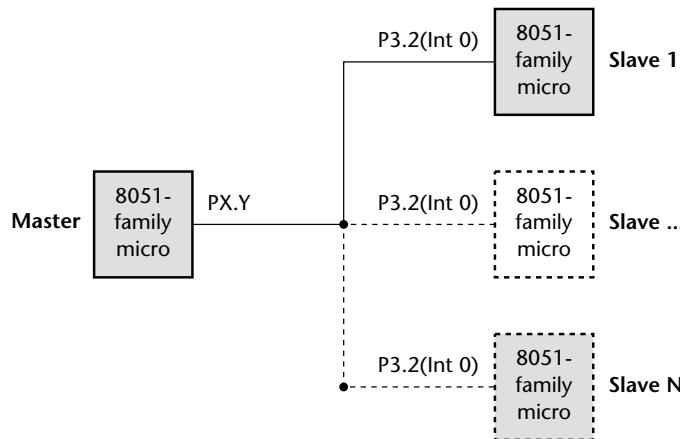


FIGURE 26.3 Simple shared-clock (interrupt) schedulers may involve multiple Slaves

Adding flesh on the bones

We now consider the implementation of this scheduler in more detail: Figure 26.4 illustrates a practical hardware framework.

As we discussed in Chapter 25, we can improve the reliability of the shared-clock network by adding a watchdog timer to the Slave unit(s). When this is added then – with appropriate coding – the Slaves are able to detect a failure in the Master or elsewhere in the network.

Of course, it is not necessary to use an external watchdog timer: Figure 26.5 shows an alternative hardware platform employing an internal watchdog timer.

A complete code implementation for this scheduler is given in the examples that follow.

Increasing the system reliability

The shared-clock (interrupt) scheduler just described is simple and powerful. In this environment, failure of the Master is quickly detected by the Slave(s). However, if one of the Slave fails, this will not be detected by either the Master or the other Slaves.

This scheduler is therefore not adequate for applications where we need to be sure that all nodes are running correctly. Figure 26.6 shows one way of addressing this problem: note that this solution is only appropriate for two-node networks.

In Figure 26.6, we again have two 8051-based nodes. The Slave can detect the failure of the Master (or the communication link) through use of the watchdog. This time, however, the signal used by the Slave to refresh the external Slave watchdog is also fed to the Master node: by monitoring activity on the watchdog pin, the Master can determine whether the Slave is still operational.

A complete code example for this form of scheduler is given later in this chapter.

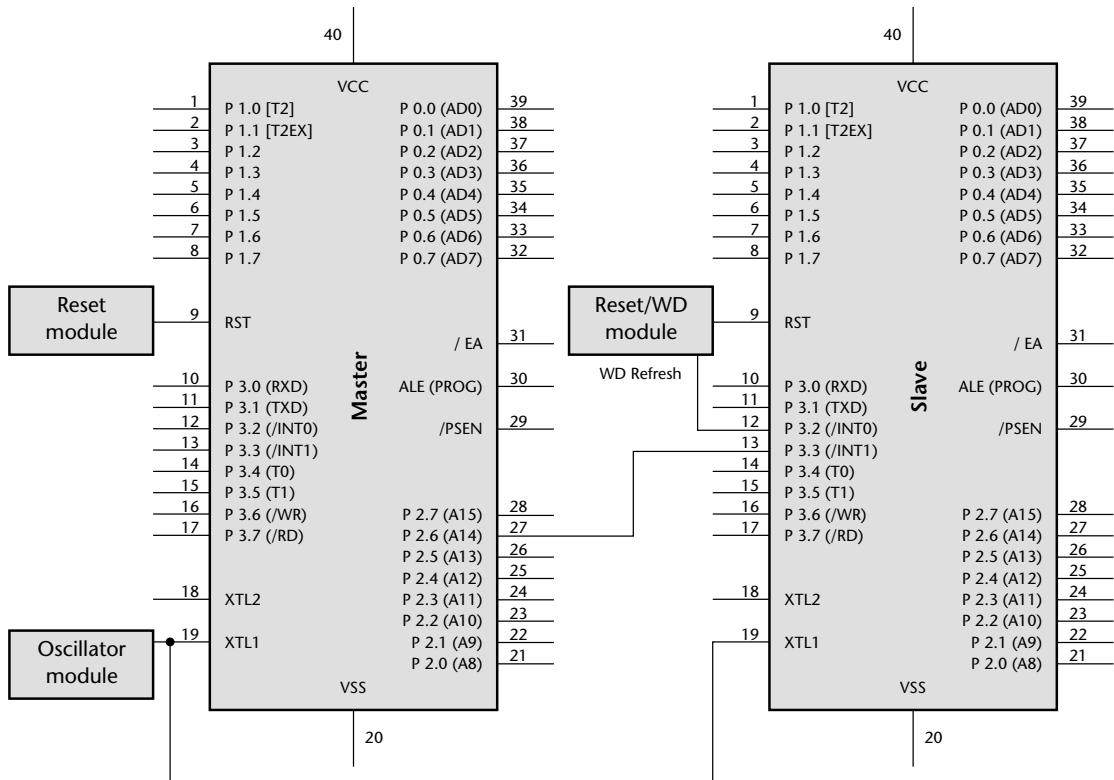


FIGURE 26.4 A possible hardware platform for an SCI scheduler (with external watchdog timer on Slave unit)

Hardware resource implications

This is a very efficient scheduler. The additional software load in the Master node – compared with a standard co-operative scheduler – is generally too small to measure. The hardware required is only two port pins on each of the Master and Slave devices.

In addition, in the Slave(s), Timer 0, Timer 1 and, where applicable, Timer 2 are all available.

Reliability and safety implications

The use of **SCI SCHEDULER (TICK)** has a number of important safety and reliability implications. Please review the material in this section carefully before deciding to use this architecture in your application.

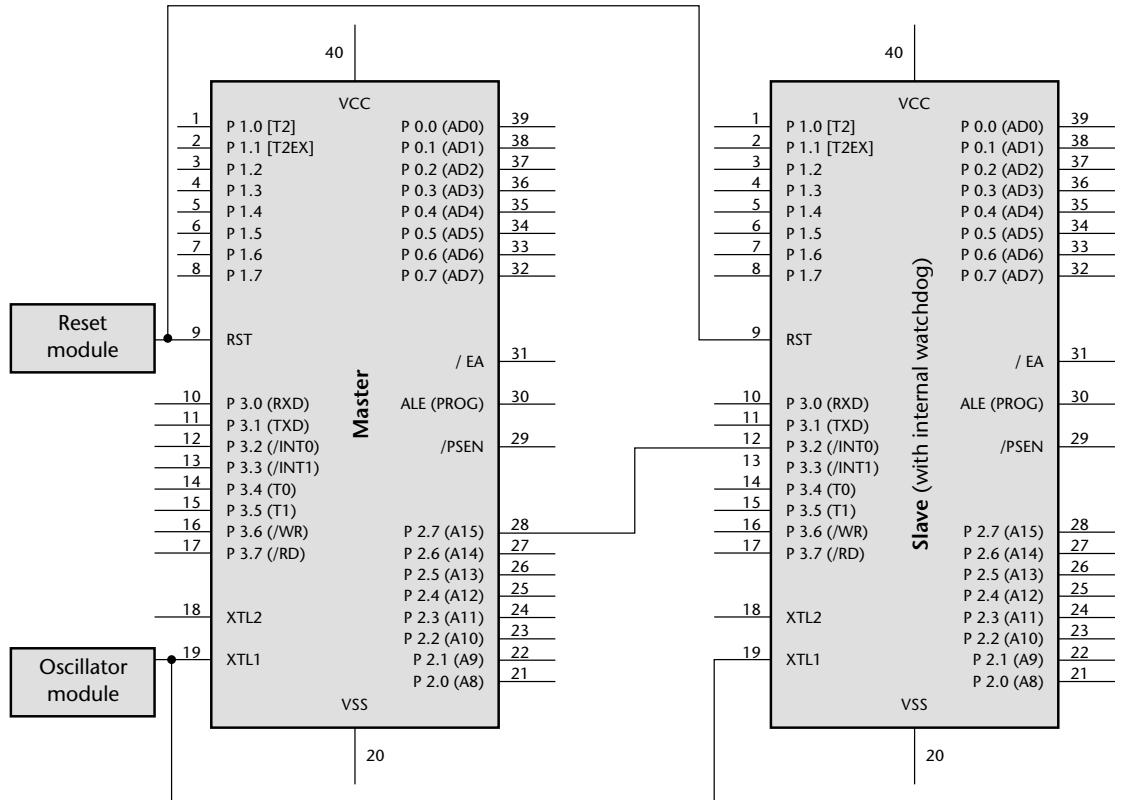


FIGURE 26.5 A possible hardware platform for an SCI scheduler (with internal watchdog timer on Slave unit)

Why additional processors may not always improve reliability

See page 550 for a discussion of the reasons why use of multiple microcontrollers may decrease the system reliability.

The impact of oscillator drift

The accuracy of the timing in the whole network of, say, ten microcontrollers depends on the stability of the clock in the Master. Where necessary, a temperature-compensated or satellite-based clock may be used in the Master node to ensure accurate timing throughout the network.

The techniques required to achieve this are discussed in **STABLE SCHEDULER** [page 932].

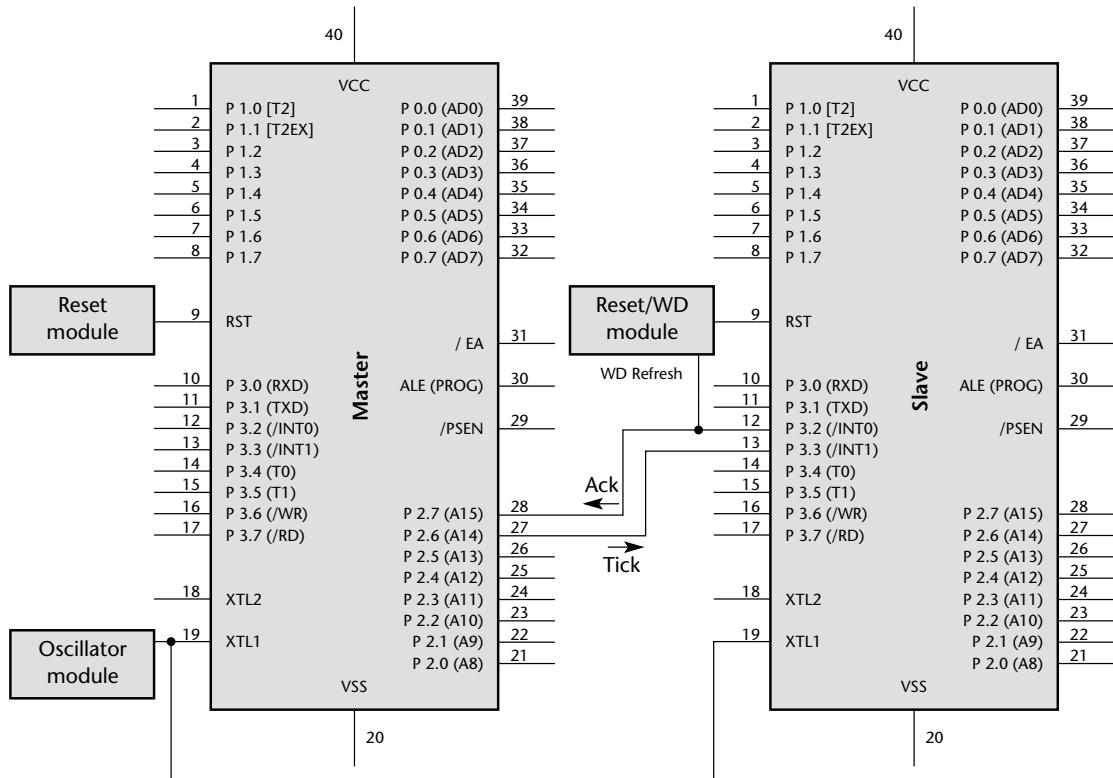


FIGURE 26.6 Another form of shared-clock (interrupt) scheduler.

[Note: In this version, the Master is able to determine whether the Slave is operational.]

The impact of oscillator failure

One legitimate criticism that can be made of the shared-clock scheduler is that the Master node itself is a ‘single point of failure’. That is, if the oscillator attached to the Master node fails completely or if the Master software or hardware fails, the whole network will stop.

This is a legitimate concern, but it may be addressed through the creation of one or more ‘backup Master’ units, that operate as follows:

- Under normal circumstances, incoming ticks from the Master put the backup Master to sleep.
- If the ticks stop (indicating failure of the Master), then the backup Master disconnects the Master from the bus and takes over this role.
- Multiple backup Masters may be used on the same network, by varying the time taken to respond to a Master failure. Thus, one Backup Master may step in after one second, the second after two seconds and so on.

The dangers of external interrupts

The use of external interrupts to convey tick messages means that the resulting applications are at particular risk from sources of electromagnetic interference (EMI), unless sensible precautions are taken.

This sensitivity of the microcontroller to EMI is greatly increased if long leads (that is, aerials) are connected to the interrupt pins. As a result, these schedulers are only suitable for ‘local’ networks: that is, for applications containing multiple microcontrollers which are displaced by no more than (at most) a few centimetres; in most cases, the microcontrollers should be mounted on the same PCB and housed in a shielded box.

Detecting and handling errors

We discussed three techniques for detecting and handling errors in a shared-clock network in Chapter 25. In this chapter we assume that ‘enter safe state then shut down’ error handling is used: see Chapter 25 for further details.

Portability

Almost all microcontroller families have at least one external interrupt pin. As a result, these patterns may be adapted without difficulty for use with these families.

Overall strength and weakness

- ☺ Simple, effective and with very low overheads.
- ☺ Provides one-way communication between Master and Slave(s): Master cannot detect errors on the Slave node(s).
- ☹ There is no mechanism for transferring *data* between the Master and Slave (or vice versa).
- ☹ The system may be vulnerable to EMI unless sensible precautions are taken.

Related patterns and alternative solutions

The other patterns in this chapter and throughout Part F provide alternative techniques for linking together more than one microcontroller.

In addition, there is a further alternative, illustrated in Figure 26.7. In Figure 26.7, the only link between the Master and the Slave is the reset and clock circuits; each device has its own timer-driven scheduler. This approach is not generally recommended, for the following reasons:

- We have to use an important hardware resource (a timer) on each of the nodes.
- The tasks on the two (or more) nodes will not, generally, be synchronized, since the start of scheduling on each node will depend on the time taken to perform initialization functions (if any) on each node.

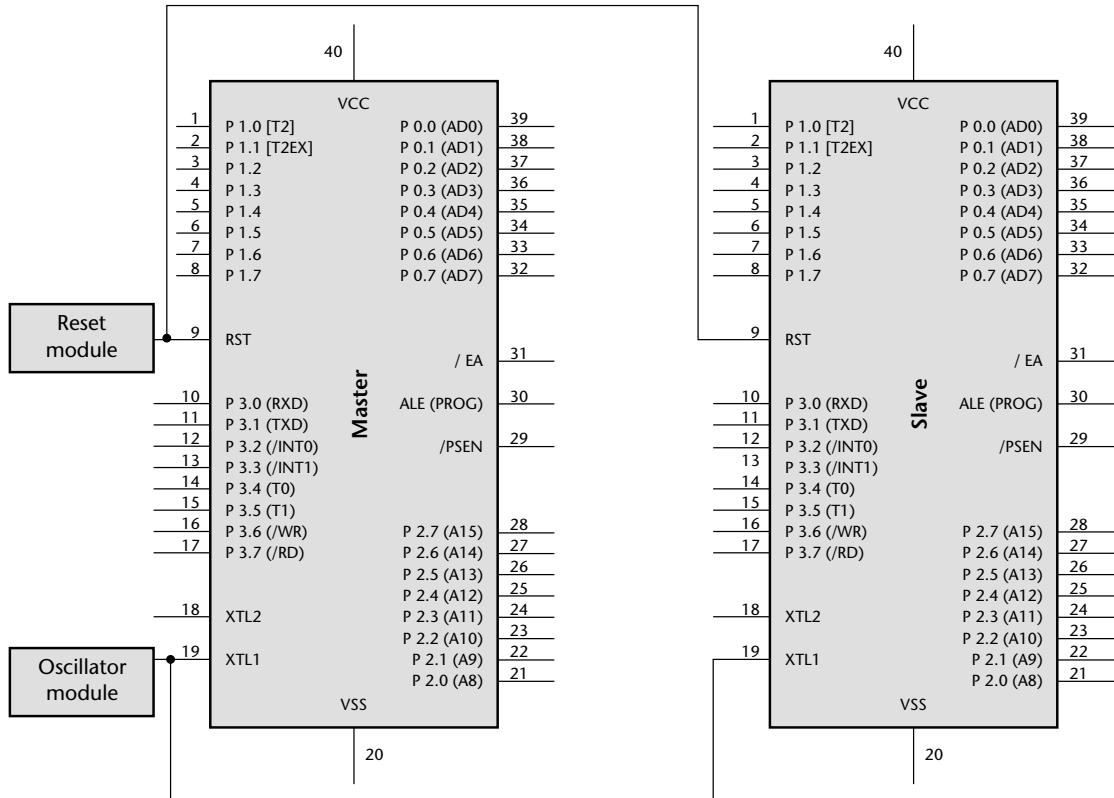


FIGURE 26.7 An alternative ‘shared-clock’ scheduler, where the Master and Slave units are very loosely coupled.

- The Master and Slaves are completely independent: no node has any knowledge of the status of the other nodes.

Example: Precise timer ticks and standard baud rates

As illustrated in Figure 26.6, an SCI scheduler will typically use a single oscillator circuit to drive the two (or more) processors: this reduces the cost and the physical board size.

However, this need not be the best option. If, for example, we use a 12 (or 24) MHz oscillator to drive the Master, all the microcontrollers can be operated with precise 1 ms tick timing. If one of the Slaves has a separate 11.059 MHz oscillator circuit, the Slave will operate at 1 ms ticks, but will also be capable of generating standard (e.g. 9,600) baud rates (Figure 26.8).

This combination can be very useful in some applications.

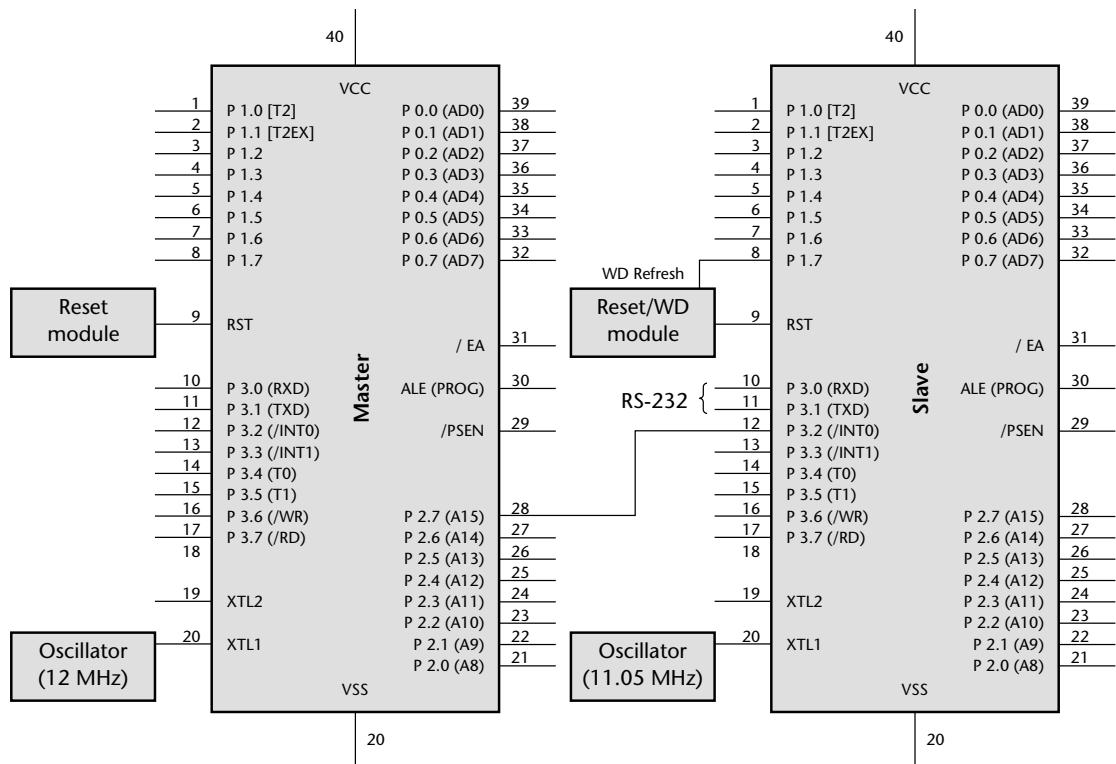


FIGURE 26.8 A shared-clock scheduler with Master and Slave employing different oscillator frequencies

Example: Traffic lights (Version 1)

We will illustrate the use of the schedulers in this chapter using different versions of the ‘traffic-light’ example introduced in Chapter 25.

Figure 26.9 shows the hardware to be used in the first example of this system. The three LEDs on pins 2.0, 2.1 and 2.2 represent the lights (red, amber and green, respectively). The additional LED on Pin 0.7 is a ‘flashing LED’ used simply to illustrate the operation of the prototype.

The software for the Master and Slave nodes, based on **SCI SCHEDULER (TICK)**, is in Listings 26.1 to 26.7.

Software – Master node

```
/*-----*
```

```
Port.H (v1.00)
```

```
*-----*/
```

```
'Port Header' (see Chap 10) for the project SCI_Ti1M (Chap 26)
```

```
*-----*/
```

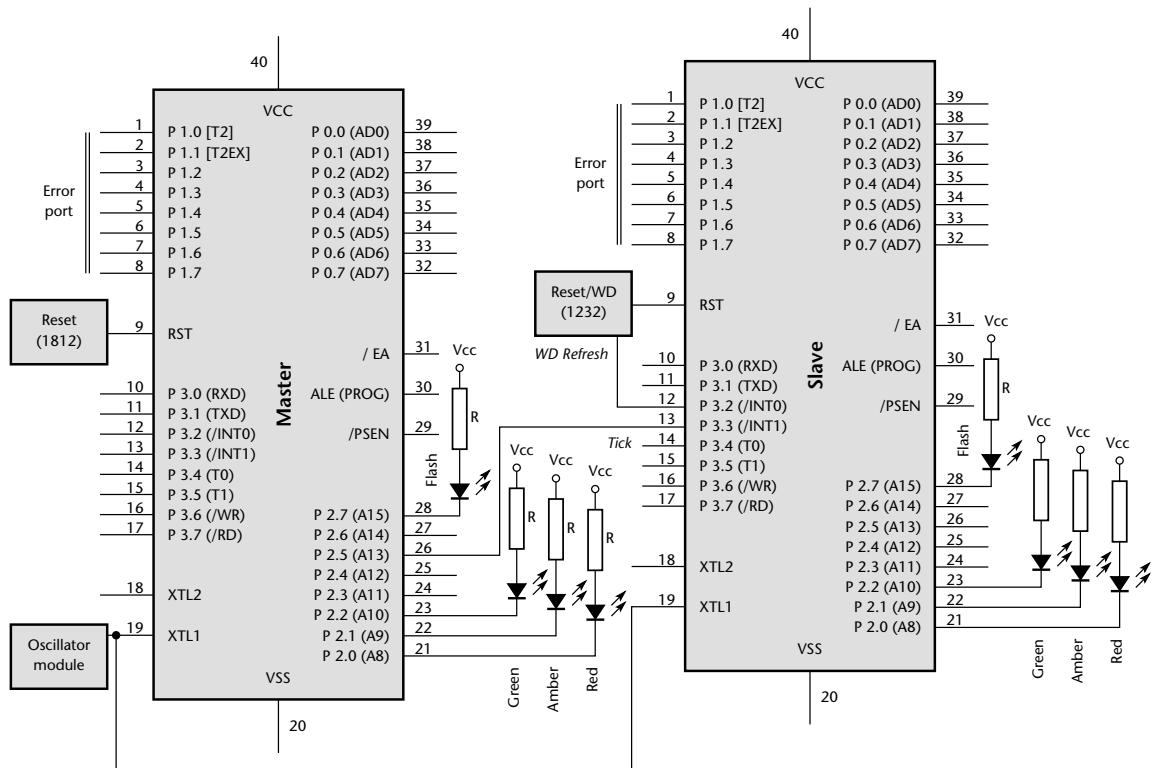


FIGURE 26.9 Hardware for a simple ‘traffic light’ example

[Note: that, in this simple version of the application, the Master cannot detect failures in the Slave node. In addition, no data transfer is possible between Master and Slave (or vice versa). This example is, therefore, intended purely to illustrate the use of this simple shared-clock scheduler architecture.]

```

// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif

// ----- SCI_Ti1m.C -----
// This pin is wired to the interrupt input pin
// (usually Pin 3.2 or P 3.3) of the Slave microcontroller

```

```

sbit Interrupt_output_pin = P2^5;

// ----- TLights_A.C -----
sbit Red_light    = (P2^0);
sbit Amber_light = (P2^1);
sbit Green_light = (P2^2);

// ----- LED_Flas.C -----
// For flashing LED
sbit LED_pin = P2^7;

/*
----- END OF FILE -----
*/

```

Listing 26.1 Part of the software for a simple traffic system (Master node)

[Note: that, in this simple version of the application, the Master cannot detect failures in the Slave node. In addition, no data transfer is possible between Master and Slave (or vice versa).]

```

/*
----- * -
Main.c (v1.00)

-----
Test program for shared-clock (interrupt) scheduler for 89C52, etc.

*** Tick 1 - MASTER CODE ***

*** Both Master and Slave share the same tick rate (1 ms) ***

Required linker options (see text for details):

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)
{

```

```

// Set up the scheduler
SCI_TICK1_MASTER_Init_T2();

// Prepare for the traffic light task
TRAFFIC_LIGHTS_Init();

// Prepare for the flash LED task (demo only)
LED_Flash_Init();

// Add a 'flash LED' task (on for 1000 ms, off for 1000 ms)
SCH_Add_Task(LED_Flash_Update, 0, 1000);

// Add a 'traffic light' task
SCH_Add_Task(TRAFFIC_LIGHTS_Update, 30, 1000);

// Start the scheduler
SCI_TICK1_MASTER_Start();

while(1)
{
    SCH_Dispatch_Tasks();
}

/*
----- END OF FILE -----
*/

```

Listing 26.2: Part of the software for a simple traffic system (Master node)

[Note: that, in this simple version of the application, the Master cannot detect failures in the Slave node. In addition, no data transfer is possible between Master and Slave (or vice versa).]

```

/*
----- *
SCI_Ti1m.c (v1.00)

-----
THIS IS A SHARED-CLOCK INTERRUPT SCHEDULER FOR 8051/52

*** MASTER NODE : TICK-ONLY (DUPLEX) ***
*** Uses T2 for timing, 16-bit auto reload ***
*** 12 MHz oscillator -> 1 ms (precise) tick interval ***
--- Assumes '1232' watchdog on Slave ---

*/
#include "Main.h"
#include "Port.h"

```

```
#include "SCI_Ti1m.H"
#include "Delay_T0.h"
#include "TLight_A.h"

// ----- Public variable declarations -----

// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

/*-----*
SCI_TICK1_MASTER_Init_T2()

Scheduler initialization function. Prepares scheduler data
structures and sets up timer interrupts at required rate.
You must call this function before using the scheduler.

*-----*/
void SCI_TICK1_MASTER_Init_T2(void)
{
    tByte i;

    // No interrupts (yet)
    EA = 0;

    // ----- Set up the scheduler -----
    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // ----- Set up Timer 2 (begin) -----
    // Now set up Timer 2
    // 16-bit timer function with automatic reload

    // Crystal is assumed to be 12 MHz
    // The Timer 2 resolution is 0.000001 seconds (1 µs)
    // The required Timer 2 overflow is 0.001 seconds (1 ms)
    // - this takes 1000 timer ticks
    // Reload value is 65536 - 1000 = 64536 (dec) = 0xFC18
```

```

T2CON = 0x04; // load Timer 2 control register
T2MOD = 0x00; // load Timer 2 mode register

TH2    = 0xFC; // load Timer 2 high byte
RCAP2H = 0xFC; // load Timer 2 reload capture reg, high byte
TL2    = 0x18; // load Timer 2 low byte
RCAP2L = 0x18; // load Timer 2 reload capture reg, low byte

ET2    = 1; // Timer 2 interrupt is enabled

TR2    = 1; // Start Timer 2
// ----- Set up Timer 2 (end) -----
}

/*-----*/
SCI_TICK1_MASTER_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

- *-----*/
void SCI_TICK1_MASTER_Start(void)
{
// Try to place system in 'safe' state at start or after errors
SCI_TICK1_MASTER_Enter_Safe_State();

// Delay here to cause the Slave to time out and reset
// Adjust this delay to match the timeout periods on the Slaves
// Hardware_Delay_T0(500);

// Now send first tick to start the Slave
// (starts on falling edge)
Interrupt_output_pin = 1;
Hardware_Delay_T0(5);
Interrupt_output_pin = 0;
Hardware_Delay_T0(5);

// Start the scheduler
EA = 1;
}

/*-----*/
SCI_TICK1_MASTER_Update_T2

```

This is the scheduler ISR. It is called at a rate determined by the timer settings in SCI_TICK1_MASTER_Init_T2(). This version is triggered by Timer 2 interrupts: timer is automatically reloaded.

```
-----*/  
void SCI_TICK1_MASTER_Update_T2(void) interrupt  
INTERRUPT_Timer_2_Overflow  
{  
    tByte Index;  
  
    TF2 = 0; // Must manually clear this.  
  
    // Send 'tick' message to the Slave  
    Interrupt_output_pin = 0;  
  
    // NOTE: calculations are in *TICKS* (not milliseconds)  
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)  
    {  
        // Check if there is a task at this location  
        if (SCH_tasks_G[Index].pTask)  
        {  
            if (SCH_tasks_G[Index].Delay == 0)  
            {  
                // The task is due to run  
                SCH_tasks_G[Index].RunMe += 1; // Increment the run flag  
  
                if (SCH_tasks_G[Index].Period)  
                {  
                    // Schedule periodic tasks to run again  
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;  
                }  
            }  
        else  
        {  
            // Not yet ready to run: just decrement the delay  
            SCH_tasks_G[Index].Delay -= 1;  
        }  
    }  
    // Prepare for next tick  
    Interrupt_output_pin = 1;  
}  
/*-----*/  
SCI_TICK1_MASTER_Enter_Safe_State()
```

This is the state entered by the system when:

- (1) The node is powered up or reset
- (2) The Slave node fails
- (3) The network has an error
- (4) Ack messages are delayed for any other reason

Try to ensure that the system is in a 'safe' state in these circumstances.

```
----- */
void SCI_TICK1_MASTER_Enter_Safe_State(void) reentrant
{
    // USER DEFINED - Edit as required

    // Here we display a safe output
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

----- END OF FILE -----
----- */
```

Listing 26.3: Part of the software for a simple traffic system (Master node)

[Note: that, in this simple version of the application, the Master cannot detect failures in the Slave node. In addition, no data transfer is possible between Master and Slave (or vice versa).]

```
----- */
TLight_A.C (v1.00)

-----
Traffic light control program

Simplex version
- has no information about bulb status on other node.

----- */
#include "Main.h"
#include "Port.h"

#include "TLight_A.h"

// ----- Private constants -----
// Easy to change logic here
#define ON 0
#define OFF 1
```

```
// Times in each of the (four) possible light states
// (Times are in seconds - must call the update task once per second)
//
#define RED_DURATION (10)
#define RED_AND_AMBER_DURATION (10)

// NOTE:
// GREEN_DURATION must equal RED_DURATION
// AMBER_DURATION must equal RED_AND_AMBER_DURATION
#define GREEN_DURATION RED_DURATION
#define AMBER_DURATION RED_AND_AMBER_DURATION

// Must specify whether this is a MASTER or a SLAVE unit
#define MASTER_SLAVE MASTER

// ----- Private variables -----
// The state of the system
static eLight_State Light_state_G;

// ----- Private function prototypes -----
bit TRAFFIC_LIGHTS_Check_Local_Bulb(void);

/*-----*
 *TRAFFIC_LIGHTS_Init()
 *
 Prepare for scheduling.

-*-----*/
void TRAFFIC_LIGHTS_Init(void)
{
    //
    // Master and Slave must start in opposite states
    if (MASTER_SLAVE == MASTER)
    {
        //
        Light_state_G = RED;
    }
    else
    {
        //
        Light_state_G = GREEN;
    }

    // Display safe output until scheduler starts
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*-----*
 *TRAFFIC_LIGHTS_Update()

```

Must be scheduled once per second.

```
- *-----*/  
void TRAFFIC_LIGHTS_Update(void)  
{  
    static tWord Time_in_state;  
  
    // Check for blown bulbs on this node  
    TRAFFIC_LIGHTS_Check_Local_Bulb();  
  
    // This is the main update code  
    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:
    {
        // Blown bulb detected
        // Switch all bulbs off
        // (Drivers won't be happy, but it will be clear
        // that something is wrong)
        Red_light = OFF;
        Amber_light = OFF;
        Green_light = OFF;

        // We remain in this state until state
        // is changed manually, or system is reset
        break;
    }
}

/*-----*
 * TRAFFIC_LIGHTS_Check_Local_Bulb()

 Check the status of the local bulbs (DUMMY FUNCTION HERE)

 *-----*/

```

```

bit TRAFFIC_LIGHTS_Check_Local_Bulb(void)
{
    // This dummy function confirms the bulbs are OK
    //
    // - See Chapter 32 for complete version of this function.
    return RETURN_NORMAL;
}

/*-----*/
TRAFFIC_LIGHTS_Display_Safe_Output()
Used in the event of system failure, etc.

void TRAFFIC_LIGHTS_Display_Safe_Output(void)
{
    if (TRAFFIC_LIGHTS_Check_Local_Bulb() == RETURN_NORMAL)
    {
        // Bulbs are OK
        // - best thing to do is to display STOP
        Red_light = ON;
        Amber_light = OFF;
        Green_light = OFF;
    }
    else
    {
        // At least one bulb has blown
        // - best thing we can do is extinguish all bulbs
        Red_light = OFF;
        Amber_light = OFF;
        Green_light = OFF;
    }
}

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 26.4 Part of the software for a simple traffic system (Master node).

[Note: that, in this simple version of the application, the Master cannot detect failures in the Slave node. In addition, no data transfer is possible between Master and Slave (or vice versa).]

Software – Slave node

```

/*-----*/
Port.H (v1.00)

-----
'Port Header' (see Chap 10) for the project SCI_Ti1s (see Chap 26)

/*-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1
#endif

// ----- SCI_Ti1s.C -----
// P3^3 used for interrupt input
// Connect 1232 (pin /ST) to the WATCHDOG_pin
sbit WATCHDOG_pin = P3^2;

// ----- TLights_A.C -----
sbit Red_light    = (P2^0);
sbit Amber_light = (P2^1);
sbit Green_light = (P2^2);

// ----- LED_Flas.C -----
// For flashing LED
sbit LED_pin = P2^7;

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 26.5 Part of the software for a simple traffic system (Slave node)

[Note: that, in this simple version of the application, the Master cannot detect failures in the Slave node. In addition, no data transfer is possible between Master and Slave (or vice versa).]

```
/*-----*  
Main.c (v1.00)  
-----  
  
Test program for shared-clock interrupt scheduler.  
*** TICK 2 - SLAVE CODE ***  
--- HARDWARE ASSUMED ---  
--- 89C52 (or any 8051/52 device with T2)  
--- DS1232 (or similar) external watchdog [see text for conns]  
*** Both Master and Slave share the same tick rate (1 ms) ***  
*** - See Master code for details ***  
  
Required linker options (see text for details):  
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_Ti1s.H"  
#include "TLight_A.h"  
  
/* ..... */  
/* ..... */  
  
void main(void)  
{  
    // Set up the scheduler  
    SCI_TICK1_SLAVE_Init();  
  
    // Set up the flash LED task (demo purposes)  
    LED_Flash_Init();  
  
    // Prepare for the traffic light task  
    TRAFFIC_LIGHTS_Init();  
  
    // Add a 'flash LED' task (on for 1000 ms, off for 1000 ms)  
    SCH_Add_Task(LED_Flash_Update, 0, 1000);  
  
    // Add a 'traffic lights' task  
    SCH_Add_Task(TRAFFIC_LIGHTS_Update, 10, 1000);  
  
    // Start the scheduler  
    SCI_TICK1_SLAVE_Start();  
  
    while(1)
```

```

    {
        SCH_Dispatch_Tasks();
    }
}

/* -----
--- END OF FILE ---
-*----- */

```

Listing 26.6 Part of the software for a simple traffic system (Slave node)

[Note: that, in this simple version of the application, the Master cannot detect failures in the Slave node. In addition, no data transfer is possible between Master and Slave (or vice versa).]

```

/* -----
----- SCi_Ti1s.c (v1.00)

----- THIS IS A SHARED-CLOCK SCHEDULER [INTERRUPT BASED] FOR 8051/52
*** TICK1 - SLAVE NODE ***
*** Uses 1232 watchdog timer ***
*** Both Master and Slave share the same tick rate (1 ms) ***
*** - See Master code for details ***

-*----- */
#include "Main.h"
#include "Port.h"

#include "SCI_Ti1s.h"
#include "TLight_A.h"

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// ----- Private function prototypes -----
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()

Scheduler initialization function. Prepares scheduler data structures and sets up timer interrupts at required rate.

Must call this function before using the scheduler.

```
/*-----*/
void SCI_TICK1_SLAVE_Init(void)
{
    tByte i;

    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // ----- External interrupt 0 -----
    // The Slave is driven by an interrupt input
    // The interrupt is enabled
    // It is triggered by a falling edge at pin P3.2
    IT0 = 1;
    EX0 = 1;

    // Start the watchdog
    SCI_TICK1_SLAVE_Watchdog_Init();
}
```

```
/*-----*/
SCI_TICK1_SLAVE_Start()
```

Starts the Slave scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added, to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

```
/*-----*/
void SCI_TICK1_SLAVE_Start(void)
{
    // Place system in a safe state
    // - Slave will keep returning here if Master does not start
```

```

// - or otherwise fails.
SCI_TICK1_SLAVE_Enter_Safe_State();

// Now in a safe state
// Wait here - indefinitely - for the first tick
// (Refresh the watchdog to avoid constant watchdog restarts)
while (IE0 == 0)
{
    SCI_TICK1_SLAVE_Watchdog_Refresh();
}

// Clear the flag
IE0 = 0;

// Start the scheduler
EA = 1;
}

/*-----*
SCI_TICK1_SLAVE_Update

This is the scheduler ISR. It is called at a rate
determined by the timer settings in SCI_TICK1_SLAVE_Init().

This Slave is triggered by external interrupts.

*-----*/
void SCI_TICK1_SLAVE_Update(void) interrupt INTERRUPT_EXTERNAL_0
{
    tByte Index;

    // Feed the watchdog
    // Master will monitor this pin to check for Slave activity
    SCI_TICK1_SLAVE_Watchdog_Refresh();

    // Now do 'standard' scheduler updates

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Incr. the run flag
            }
        }
    }
}

```

```

        if (SCH_tasks_G[Index].Period)
        {
            // Schedule periodic tasks to run again
            SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
        }
    }
    else
    {
        // Not yet ready to run: just decrement the delay
        SCH_tasks_G[Index].Delay -= 1;
    }
}
}

/*-----*
SCI_TICK1_SLAVE_Watchdog_Init()

This function sets up the watchdog timer.

If the Master fails (or other error develops),
no tick messages will arrive, and the scheduler
will stop.

To detect this situation, we have a (hardware) watchdog
running in the Slave. This watchdog - which should be set to
overflow at around 100 ms - is used to set the system into a
known (safe) state. The Slave will then wait (indefinitely)
for the problem to be resolved.

NOTE: The Slave will not be generating Ack messages in these
circumstances. The Master (if running) will therefore be aware
that there is a problem.

*-----*/
void SCI_TICK1_SLAVE_Watchdog_Init(void)
{
    // INIT NOT REQUIRED FOR 1232 EXTERNAL WATCHDOG
    // - May be required with different watchdog hardware
    //
    // Edit as required
}

/*-----*
SCI_TICK1_SLAVE_Watchdog_Refresh()

Feed the external (1232) watchdog.

```

Timeout is between ~60 and 250 ms (hardware dependent)

HARDWARE: Assumes external 1232 watchdog

```
-----*/
void SCI_TICK1_SLAVE_Watchdog_Refresh(void) reentrant
{
    static bit WATCHDOG_state;

    // Change the state of the watchdog pin
    if (WATCHDOG_state == 1)
    {
        WATCHDOG_state = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state = 1;
        WATCHDOG_pin = 1;
    }
}

/*-----*/
SCI_TICK1_SLAVE_Enter_Safe_State()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Master node fails
(3) The network has an error
(4) Tick messages are delayed for any other reason

Try to ensure that the system is in a 'safe' state in these
circumstances.

/*-----*/
void SCI_TICK1_SLAVE_Enter_Safe_State(void)
{
    // USER DEFINED - Edit as required
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*-----*/
---- END OF FILE -----
/*-----*/
```

Listing 26.7 Part of the software for a simple traffic system (Slave node)

[Note: that, in this simple version of the application, the Master cannot detect failures in the Slave node. In addition, no data transfer is possible between Master and Slave (or vice versa).]

Example: Traffic lights (Version 2)

The main drawback with the system implemented in the previous example was that failure of the Slave node would not be detected by the Master. This example solves this problem, by including both ‘tick’ and ‘acknowledgement’ messages in the communication protocol. The required hardware is illustrated in Figure 26.10.

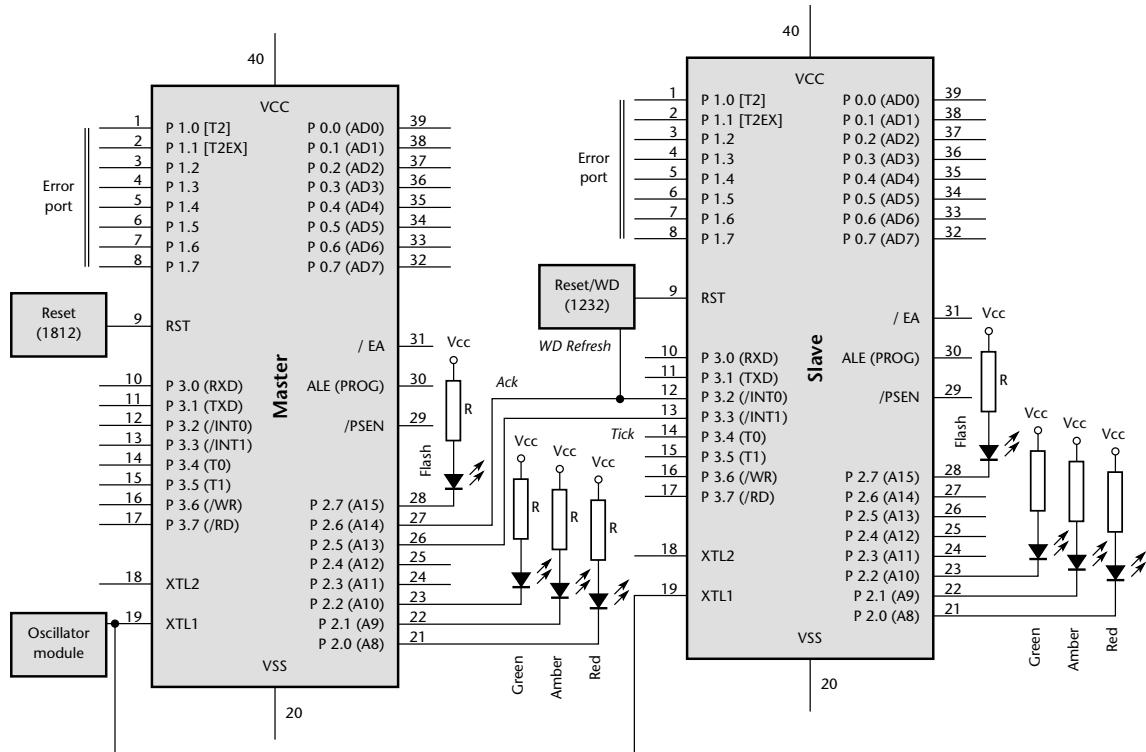


FIGURE 26.10 Hardware for a simple ‘traffic light’ example (Version 2)

[Note: that, in this version of the application, no data transfer is possible between Master and Slave (or vice versa).]

Software – Master node

We present a key software listing (Listing 26.8) for the Master node in this version of the scheduler here: the remainder of the files for this project will be found on the CD.

```

/*-----*
SCI_Ti2m.c (v1.00)

-----
THIS IS A SHARED-CLOCK INTERRUPT SCHEDULER FOR 8051/52

*** MASTER NODE : TICK-ONLY (DUPLEX) ***

*** Uses T2 for timing, 16-bit auto reload ***
*** 12 MHz oscillator -> 1 ms (precise) tick interval ***

--- Assumes '1232' watchdog on Slave ---

*-----*/
#include "Main.h"
#include "Port.h"

#include "SCI_Ti2m.H"
#include "Delay_T0.h"
#include "TLight_A.h"

// ----- Public variable definitions -----
// Used to detect Slave activity
bit First_call_G;
bit Watchdog_input_previous_G;

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// Used to reset system in event of Slave error (see Main.C)
extern bit System_reset_G;

// ----- Private function prototypes -----
static void SCI_TICK2_MASTER_Send_Tick_Message(void);
static bit SCI_TICK2_MASTER_Process_Ack(void);

/*-----*
SCI_TICK2_MASTER_Init_T2()

Scheduler initialization function. Prepares scheduler data
structures and sets up timer interrupts at required rate.
You must call this function before using the scheduler.

*-----*/

```

```

void SCI_TICK2_MASTER_Init_T2(void)
{
    tByte i;

    // No interrupts (yet)
    EA = 0;

    // ----- Set up the scheduler -----
    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // ----- Set up Timer 2 (begin) -----
    // Now set up Timer 2
    // 16-bit timer function with automatic reload

    // Crystal is assumed to be 12 MHz
    // The Timer 2 resolution is 0.000001 seconds (1 µs)
    // The required Timer 2 overflow is 0.001 seconds (1 ms)
    // - this takes 1000 timer ticks
    // Reload value is 65536 - 1000 = 64536 (dec) = 0xFC18

    T2CON = 0x04;    // load Timer 2 control register
    T2MOD = 0x00;    // load Timer 2 mode register

    TH2    = 0xFC;    // load Timer 2 high byte
    RCAP2H = 0xFC;    // load Timer 2 reload capture reg, high byte
    TL2    = 0x18;    // load Timer 2 low byte
    RCAP2L = 0x18;    // load Timer 2 reload capture reg, low byte

    ET2    = 1;    // Timer 2 interrupt is enabled

    TR2    = 1;    // Start Timer 2
    // ----- Set up Timer 2 (end) -----
}
/*-----*-
SCI_TICK2_MASTER_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

```

```

-----*/
void SCI_TICK2_MASTER_Start(void)
{
    // Try to place system in 'safe' state at start or after errors
    SCI_TICK2_MASTER_Enter_Safe_State();

    // Delay here to cause the Slave to time out and reset
    // Adjust this delay to match the timeout periods on the Slaves
    Hardware_Delay_T0(500);

    // Now send first tick to start the Slave
    // (starts on falling edge)
    Interrupt_output_pin = 1;
    Hardware_Delay_T0(5);
    Interrupt_output_pin = 0;
    Hardware_Delay_T0(5);

    // Start the scheduler
    EA = 1;
}

/*
-----*
SCI_TICK2_MASTER_Update_T2

This is the scheduler ISR. It is called at a rate determined by
the timer settings in SCI_TICK2_MASTER_Init_T2(). This version is
triggered by Timer 2 interrupts: timer is automatically reloaded.

-----*/
void SCI_TICK2_MASTER_Update_T2(void) interrupt
INTERRUPT_Timer_2_Overflow
{
    tByte Index;

    TF2 = 0; // Must manually clear this.

    // Get the ack message from the Slave
    if (SCI_TICK2_MASTER_Process_Ack() == RETURN_ERROR)
    {
        // Did not receive ack!
        Error_code_G = ERROR_SCH_LOST_SLAVE;

        // Enter safe state and remain here until reset
        SCI_TICK2_MASTER_Enter_Safe_State();
        while(1);
    }

    // Send 'tick' message to the Slave
    SCI_TICK2_MASTER_Send_Tick_Message();
}

```

```

// NOTE: calculations are in *TICKS* (not milliseconds)
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
    // Check if there is a task at this location
    if (SCH_tasks_G[Index].pTask)
    {
        if (SCH_tasks_G[Index].Delay == 0)
        {
            // The task is due to run
            SCH_tasks_G[Index].RunMe += 1; // Increment the run flag

            if (SCH_tasks_G[Index].Period)
            {
                // Schedule periodic tasks to run again
                SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
            }
        }
        else
        {
            // Not yet ready to run: just decrement the delay
            SCH_tasks_G[Index].Delay -= 1;
        }
    }
}

// Prepare for next tick
Interrupt_output_pin = 1;
}

/*-----*
SCI_TICK2_MASTER_Send_Tick_Message()
This function sends a tick message.

The receipt of this message will cause an interrupt to be generated
in the Slave(s): this will, in turn, invoke the scheduler 'update'
function in the Slave(s).
*-----*/
void SCI_TICK2_MASTER_Send_Tick_Message(void)
{
    // Send tick (falling edge) to the Slave
    Interrupt_output_pin = 0;
}

/*-----*

```

```

SCI_TICK2_MASTER_Process_Ack()

Checks that the Slave is operating.

*-----*/
bit SCI_TICK2_MASTER_Process_Ack(void)
{
    if (First_call_G)
    {
        // This is the first time this function has been called
        First_call_G = 0;

        // Prepare for subsequent checking of the watchdog pin
        Watchdog_input_previous_G = Slave_watchdog_pin;
    }
    else
    {
        // Watchdog pin should change state every time
        // - if the Slave is running correctly
        if (Watchdog_input_previous_G == Slave_watchdog_pin)
        {
            // Error!
            return RETURN_ERROR;
        }

        // Slave is OK
        Watchdog_input_previous_G = Slave_watchdog_pin;
    }

    // Set up port for reading
    SCI_transfer_port = 0xFF;

    return RETURN_NORMAL;
}

*-----*
SCI_TICK2_MASTER_Enter_Safe_State()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Slave node fails
(3) The network has an error
(4) Ack messages are delayed for any other reason

Try to ensure that the system is in a 'safe' state in these
circumstances.

*-----*/

```

```

void SCI_TICK2_MASTER_Enter_Safe_State(void) reentrant
{
    // USER DEFINED - Edit as required

    // Here we display a safe output
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 26.8 Part of the software for a simple traffic system (Master node)

[Note: that no data transfer is possible between Master and Slave (or vice versa).]

Software – Slave node

We present a key software listing (Listing 26.9) for the Slave node in this version of the scheduler here: the remainder of the files for this project will be found on the CD.

```

/*-----*
SCI_Ti2s.c (v1.00)

-----*/

THIS IS A SHARED-CLOCK SCHEDULER [INTERRUPT BASED] FOR 8051/52

*** TICK2 - SLAVE NODE ***

*** Uses 1232 watchdog timer ***

*** Both Master and Slave share the same tick rate (1 ms) ***
*** - See Master code for details ***

*-----*/
#include "Main.h"
#include "Port.h"

#include "SCI_Ti2s.h"
#include "TLight_A.h"

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// ----- Private function prototypes -----

```

```

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()

Scheduler initialization function. Prepares scheduler
data structures and sets up timer interrupts at required rate.

Must call this function before using the scheduler.

/*-----*/
void SCI_TICK2_SLAVE_Init(void)
{
    tByte i;

    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // ----- External interrupt 0 -----
    // The Slave is driven by an interrupt input
    // The interrupt is enabled
    // It is triggered by a falling edge at pin P3.2
    IT0 = 1;
    EX0 = 1;

    // Start the watchdog
    SCI_TICK2_SLAVE_Watchdog_Init();
}

/*-----*/
SCI_TICK2_SLAVE_Start()

Starts the Slave scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

/*-----*/

```

```

void SCI_TICK2_SLAVE_Start(void)
{
    // Place system in a safe state
    // - Slave will keep returning here if Master does not start
    // - or otherwise fails.
    SCI_TICK2_SLAVE_Enter_Safe_State();

    // Now in a safe state
    // Wait here - indefinitely - for the first tick
    // (Refresh the watchdog to avoid constant watchdog restarts)
    while (IE0 == 0)
    {
        SCI_TICK2_SLAVE_Watchdog_Refresh();
    }

    // Clear the flag
    IE0 = 0;

    // Start the scheduler
    EA = 1;
}

/*-----*
SCI_TICK2_SLAVE_Update

This is the scheduler ISR. It is called at a rate
determined by the timer settings in SCI_TICK2_SLAVE_Init().

This Slave is triggered by external interrupts.

*-----*/
void SCI_TICK2_SLAVE_Update(void) interrupt INTERRUPT_EXTERNAL_0
{
    tByte Index;

    // Feed the watchdog
    // Master will monitor this pin to check for Slave activity
    SCI_TICK2_SLAVE_Watchdog_Refresh();

    // Now do 'standard' scheduler updates

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run

```

```

SCH_tasks_G[Index].RunMe += 1; // Incr. the run flag

if (SCH_tasks_G[Index].Period)
{
    // Schedule periodic tasks to run again
    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
}
else
{
    // Not yet ready to run: just decrement the delay
    SCH_tasks_G[Index].Delay -= 1;
}
}

/*
SCI_TICK2_SLAVE_Watchdog_Init()

This function sets up the watchdog timer.

If the Master fails (or other error develops),
no tick messages will arrive, and the scheduler
will stop.

To detect this situation, we have a (hardware) watchdog
running in the Slave. This watchdog - which should be set to
overflow at around 100 ms - is used to set the system into a
known (safe) state. The Slave will then wait (indefinitely)
for the problem to be resolved.

NOTE: The Slave will not be generating Ack messages in these
circumstances. The Master (if running) will therefore be aware
that there is a problem.

*/
void SCI_TICK2_SLAVE_Watchdog_Init(void)
{
    // INIT NOT REQUIRED FOR 1232 EXTERNAL WATCHDOG
    // - May be required with different watchdog hardware
    //
    // Edit as required
}

/*
SCI_TICK2_SLAVE_Watchdog_Refresh()

Feed the external (1232) watchdog.
Timeout is between ~60 and 250 ms (hardware dependent)

```

HARDWARE: Assumes external 1232 watchdog

```

- *-----*/  

void SCI_TICK2_SLAVE_Watchdog_Refresh(void) reentrant  

{  

    static bit WATCHDOG_state;  

    // Change the state of the watchdog pin  

    if (WATCHDOG_state == 1)  

    {  

        WATCHDOG_state = 0;  

        WATCHDOG_pin = 0;  

    }  

    else  

    {  

        WATCHDOG_state = 1;  

        WATCHDOG_pin = 1;  

    }  

}  

/*-----*-  

SCI_TICK2_SLAVE_Enter_Safe_State()  

This is the state entered by the system when:  

(1) The node is powered up or reset  

(2) The Master node fails  

(3) The network has an error  

(4) Tick messages are delayed for any other reason  

Try to ensure that the system is in a 'safe' state in these  

circumstances.  

- *-----*/  

void SCI_TICK2_SLAVE_Enter_Safe_State(void)  

{  

    // USER DEFINED - Edit as required  

    TRAFFIC_LIGHTS_Display_Safe_Output();  

}  

/*-----*-  

--- END OF FILE -----  

- *-----*/
```

Listing 26.9 Part of the software for a simple traffic system (Slave node)

[Note: that no data transfer is possible between Master and Slave (or vice versa).]

Further reading

SCI SCHEDULER (DATA)

Context

- You are developing an embedded application using more than one 8051 microcontroller.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you schedule tasks on (and transfer data over) a local network of two 8051 microcontrollers connected together via an interrupt link?

Background

Refer to Chapter 25 for an introduction to shared-clock schedulers.

Solution

SCI SCHEDULER (TICK) [page 554] is a simple, flexible and reliable scheduler. However, use of such a scheduler means that the system must be partitioned so that the tasks running on the Master and Slave nodes do not need to communicate, since this simple scheduler does not support data transfer between nodes.

SCI SCHEDULER (DATA) builds on **SCI SCHEDULER (TICK)** and provides a facility for transferring data between nodes. The basic hardware framework used to provide this behaviour is illustrated in Figure 26.11.

Like **SCI SCHEDULER (TICK)**, this scheduler keeps the tasks on the two controllers synchronized and allows both the Master and Slave nodes to detect when the other node, or the communication channel, has failed.

However, in addition, this scheduler allows one byte of information to be transferred between the Master and the Slave with each ‘tick’ message; similarly, one byte of data is transferred between the Slave and the Master with each ‘ack’ message.

A complete code listing is given later, to illustrate how the data are transferred.

Hardware resource implications

Like **SCI SCHEDULER (TICK)**, this is an efficient scheduler. The additional software load in the Master node – compared with a standard co-operative scheduler – is too small to measure.

However, this scheduler requires use of two ports (one on the Master, one on the Slave) to support the data transfers, in addition to the two port pins (on each microcontroller) used for the ‘tick’ and ‘ack’ signals.

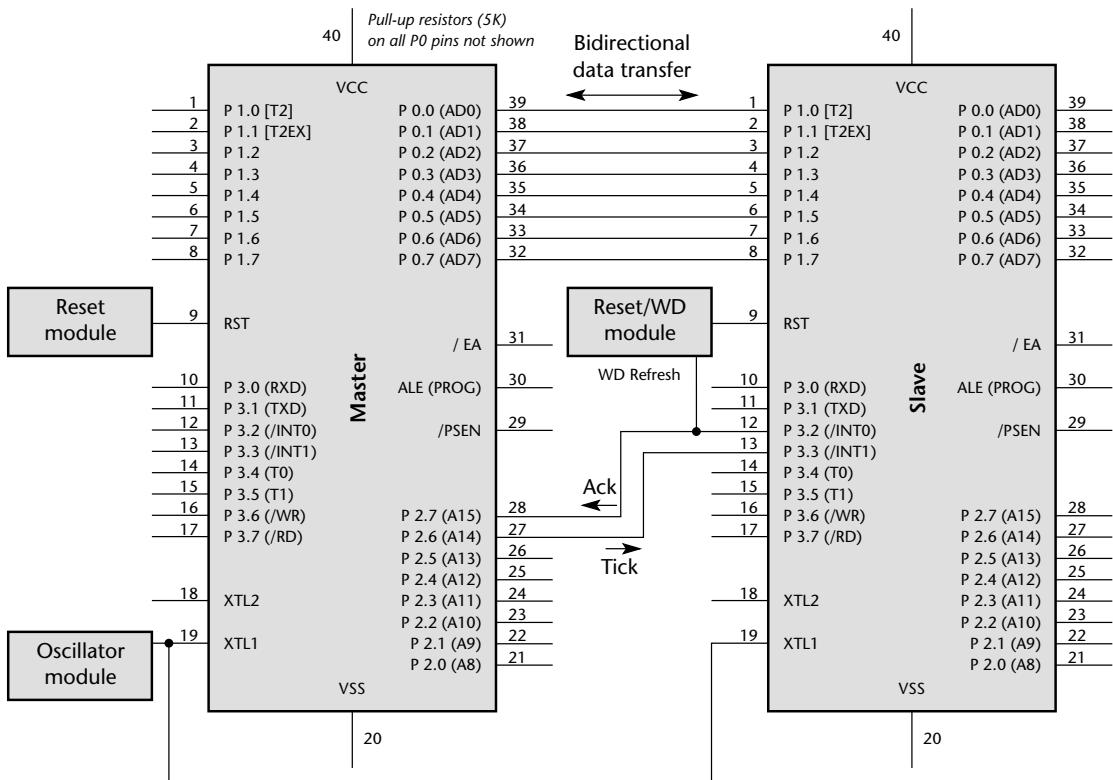


FIGURE 26.11 A shared-clock (interrupt) scheduler which supports bidirectional data transfers between Master and Slave nodes.

Reliability and safety implications

Many of the reliability and safety considerations raised in connection with **SCI SCHEDULER (TICK)** [page 554] also apply here.

Portability

Almost all microcontroller families have at least one external interrupt pin and an additional data port. As a result, these patterns may be adapted without difficulty for use with these families.

Overall strengths and weaknesses

- ☺ Simple, effective and with low software overheads.
- ☺ There is two-way communication between Master and Slave.
- ☺ There is a two-way mechanism for transferring data between the Master and Slave.

- ☺ Up to 30 bits of data may be transferred with each tick, if the ports are not required for other purposes.
- ☹ The system may be vulnerable to EMI unless sensible precautions are taken: see SCI SCHEDULER (TICK) [page 554].

Related patterns and alternative solutions

The other patterns in this chapter and throughout Part F provide alternative techniques for linking together more than one microcontroller.

Example: Traffic lights

We repeat the traffic-light example again here, to illustrate some of the advantages that are obtained by providing both ‘data’ and ‘tick’ transfers.

Hardware

The hardware required is illustrated in Figure 26.12.

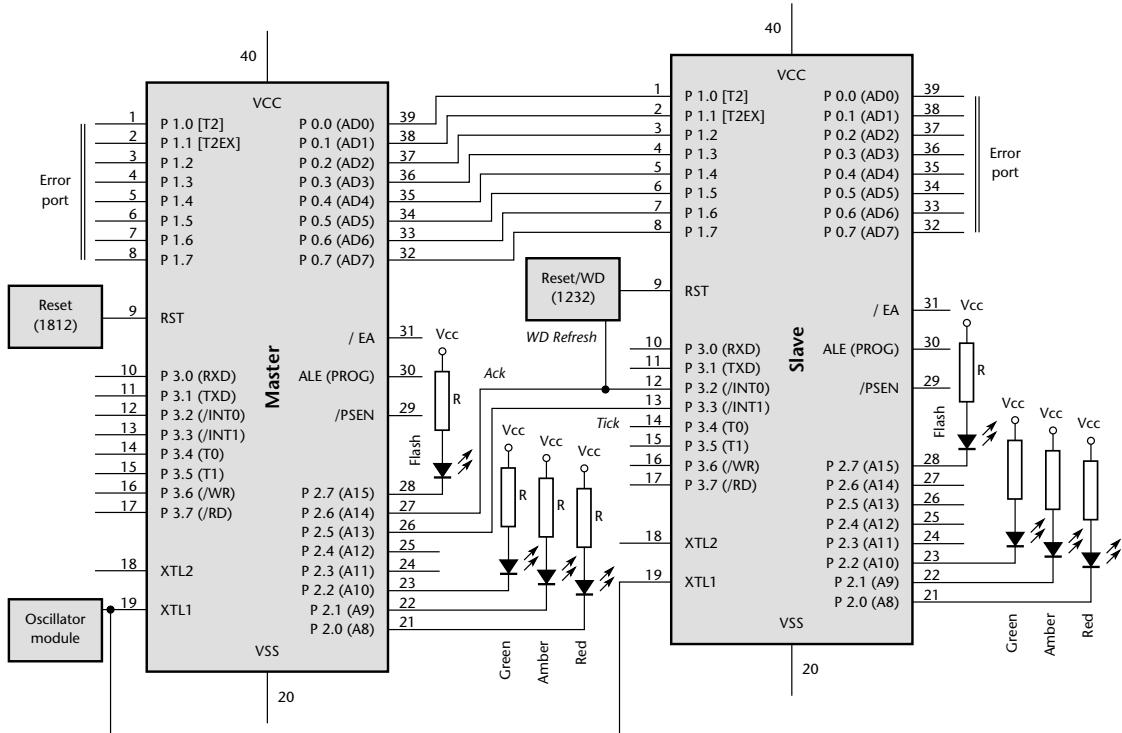


FIGURE 26.12 Hardware required for the SCI scheduler (data) version of the traffic-light controller example
[Note: that, for reliable operation, 10K pull-up resistors should be used on Port 0.]

Software – Master node

We present a key software listing (Listing 26.10) for the Master node in this version of the scheduler here: the remainder of the files for this project will be found on the CD.

```
/*-----*
SCI_Dm.c (v1.00)

-----
THIS IS A SHARED-CLOCK INTERRUPT SCHEDULER FOR 8051/52

*** MASTER NODE : DATA ***

*** Uses T2 for timing, 16-bit auto reload ***
*** 12 MHz oscillator -> 1 ms (precise) tick interval ***

--- Assumes '1232' watchdog on Slave ---

-*-----*/
```

```
#include "Main.h"
#include "Port.h"

#include "SCI_Dm.H"
#include "Delay_T0.h"
#include "TLight_B.h"

// ----- Public variable definitions -----
tByte Tick_message_data_G = RETURN_NORMAL;
tByte Ack_message_data_G = RETURN_NORMAL;

// Used to detect Slave activity
bit First_call_G;
bit Watchdog_input_previous_G;

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// Used to reset system in event of Slave error (see Main.C)
extern bit System_reset_G;
```

```

// ----- Private function prototypes -----
static void SCI_D_MASTER_Send_Tick_Message(void);
static bit SCI_D_MASTER_Process_Ack(void);

/*-----*
SCI_D_MASTER_Init_T2()

Scheduler initialization function. Prepares scheduler data
structures and sets up timer interrupts at required rate.
You must call this function before using the scheduler.

-*-----*/
void SCI_D_MASTER_Init_T2(void)
{
    tByte i;

    // No interrupts (yet)
    EA = 0;

    // ----- Set up the scheduler -----
    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // ----- Set up Timer 2 (begin) -----
    // Now set up Timer 2
    // 16-bit timer function with automatic reload

    // Crystal is assumed to be 12 MHz
    // The Timer 2 resolution is 0.000001 seconds (1 µs)
    // The required Timer 2 overflow is 0.001 seconds (1 ms)
    // - this takes 1000 timer ticks
    // Reload value is 65536 - 1000 = 64536 (dec) = 0xFC18

    T2CON = 0x04;    // load Timer 2 control register
    T2MOD = 0x00;    // load Timer 2 mode register

    TH2      = 0xFC;  // load Timer 2 high byte
    RCAP2H = 0xFC;   // load Timer 2 reload capture reg, high byte

```

```
    TL2      = 0x18; // load Timer 2 low byte
    RCAP2L  = 0x18; // load Timer 2 reload capture reg, low byte
    ET2     = 1;   // Timer 2 interrupt is enabled
    TR2     = 1;   // Start Timer 2
    // ----- Set up Timer 2 (end) -----
}

/*-----*/
SCI_D_MASTER_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

/*-----*/
void SCI_D_MASTER_Start(void)
{
    // Try to place system in 'safe' state at start or after errors
    SCI_D_MASTER_Enter_Safe_State();

    // Delay here to cause the Slave to time out and reset
    // Adjust this delay to match the timeout periods on the Slaves
    Hardware_Delay_T0(500);

    // Now send first tick to start the Slave
    // (starts on falling edge)
    Interrupt_output_pin = 1;
    Hardware_Delay_T0(5);
    Interrupt_output_pin = 0;
    Hardware_Delay_T0(5);

    // Start the scheduler
    EA = 1;
}

/*-----*/
SCI_D_MASTER_Update_T2

This is the scheduler ISR. It is called at a rate determined by
the timer settings in SCI_D_MASTER_Init_T2(). This version is
triggered by Timer 2 interrupts: timer is automatically reloaded.
```

```
-----*/
void SCI_D_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;

    TF2 = 0; // Must manually clear this.

    // Get the ack message from the Slave
    if (SCI_D_MASTER_Process_Ack() == RETURN_ERROR)
    {
        // Did not receive ack!
        Error_code_G = ERROR_SCH_LOST_SLAVE;

        // Enter safe state and remain here until reset
        SCI_D_MASTER_Enter_Safe_State();
        while(1);
    }

    // Send 'tick' message to the Slave
    SCI_D_MASTER_Send_Tick_Message();

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Increment the run flag

                if (SCH_tasks_G[Index].Period)
                {
                    // Schedule periodic tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}
```

```

    // Prepare for next tick
    Interrupt_output_pin = 1;
}

/*-----*/
SCI_D_MASTER_Send_Tick_Message()

This function sends a tick message.

The receipt of this message will cause an interrupt to be generated
in the Slave(s): this will, in turn, invoke the scheduler 'update'
function in the Slave(s).

/*-----*/
void SCI_D_MASTER_Send_Tick_Message(void)
{
    // Apply the tick data to the port
    SCI_transfer_port = Tick_message_data_G;

    // Send tick (falling edge) to the Slave
    Interrupt_output_pin = 0;
}

/*-----*/
SCI_D_MASTER_Process_Ack()

Checks that the Slave is operating.

Reads data from the Slave.

/*-----*/
bit SCI_D_MASTER_Process_Ack(void)
{
    if (First_call_G)
    {
        // This is the first time this function has been called
        First_call_G = 0;

        // Prepare for subsequent checking of the watchdog pin
        Watchdog_input_previous_G = Slave_watchdog_pin;
    }
    else
    {
        // Watchdog pin should change state every time
        // - if the Slave is running correctly
        if (Watchdog_input_previous_G == Slave_watchdog_pin)

```

```

    {
        // Error!
        return RETURN_ERROR;
    }

    // Slave is OK
    Watchdog_input_previous_G = Slave_watchdog_pin;
}

// Set up port for reading
SCI_transfer_port = 0xFF;

// Read ack message
Ack_message_data_G = SCI_transfer_port;

return RETURN_NORMAL;
}

/*-----*/
SCI_D_MASTER_Enter_Safe_State()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Slave node fails
(3) The network has an error
(4) Ack messages are delayed for any other reason

Try to ensure that the system is in a 'safe' state in these
circumstances.

/*-----*/
void SCI_D_MASTER_Enter_Safe_State(void) reentrant
{
    // USER DEFINED - Edit as required

    // Here we display a safe output
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 26.10 Part of the software for a simple traffic system (Master node)

[Note: that data transfer between Master and Slave (and vice versa) is supported in this version.]

Software – Slave node

We present a key software listing (listing 26.11) for the Slave node in this version of the scheduler here: the remainder of the files for this project will be found on the CD.

```
/*-----*
SCI_Ds.c (v1.00)

-----
THIS IS A SHARED-CLOCK SCHEDULER [INTERRUPT BASED] FOR 8051/52

*** SLAVE NODE ***

*** Uses 1232 watchdog timer ***

*** Both Master and Slave share the same tick rate (1 ms) ***
*** - See Master code for details ***

-*-----*/
```

```
#include "Main.h"
#include "Port.h"

#include "SCI_Ds.h"
#include "TLight_B.h"

// ----- Public variable definitions -----
// Data sent from the Master to this Slave
tByte Tick_message_data_G = RETURN_NORMAL;

// Data sent from this Slave to the Master
// - data may be sent on, by the Master, to another Slave
tByte Ack_message_data_G = RETURN_NORMAL;

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// ----- Private function prototypes -----
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()

Scheduler initialization function. Prepares scheduler
data structures and sets up timer interrupts at required rate.

Must call this function before using the scheduler.

----- */
void SCI_D_SLAVE_Init(void)
{
    tByte i;

    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // ----- External interrupt 0 -----
    // The Slave is driven by an interrupt input
    // The interrupt is enabled
    // It is triggered by a falling edge at pin P3.2
    IT0 = 1;
    EX0 = 1;

    // Start the watchdog
    SCI_D_SLAVE_Watchdog_Init();
}

/* -----
SCI_D_SLAVE_Start()

Starts the Slave scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

----- */
void SCI_D_SLAVE_Start(void)
{
    // Place system in a safe state

```

```

// - Slave will keep returning here if Master does not start
// - or otherwise fails.
SCI_D_SLAVE_Enter_Safe_State();

// Now in a safe state
// Wait here - indefinitely - for the first tick
// (Refresh the watchdog to avoid constant watchdog restarts)
while (IE0 == 0)
{
    SCI_D_SLAVE_Watchdog_Refresh();
}

// Clear the flag
IE0 = 0;

// Start the scheduler
EA = 1;
}

/*-----*
SCI_D_SLAVE_Update

This is the scheduler ISR. It is called at a rate
determined by the timer settings in SCI_D_SLAVE_Init().

This Slave is triggered by external interrupts.

*-----*/
void SCI_D_SLAVE_Update(void) interrupt INTERRUPT_EXTERNAL_0
{
    tByte Index;

    // Extract the tick-message data
    SCI_D_SLAVE_Process_Tick_Message();

    // Send data back to Master
    SCI_D_SLAVE_Send_Ack_Message_To_Master();

    // Feed the watchdog
    SCI_D_SLAVE_Watchdog_Refresh();

    // Now do 'standard' scheduler updates

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)

```

```

    {
        if (SCH_tasks_G[Index].Delay == 0)
        {
            // The task is due to run
            SCH_tasks_G[Index].RunMe += 1; // Incr. the run flag

            if (SCH_tasks_G[Index].Period)
            {
                // Schedule periodic tasks to run again
                SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
            }
        }
        else
        {
            // Not yet ready to run: just decrement the delay
            SCH_tasks_G[Index].Delay -= 1;
        }
    }
}

/*-----*/
SCI_D_SLAVE_Send_Ack_Message_To_Master()

Slave must send an 'Acknowledge' message to the Master, after
tick messages are received.

The acknowledge message serves two purposes:
[1] It confirms to the Master that the Slave is alive & well.
[2] It provides a means of sending data to the Master.

/*-----*/
void SCI_D_SLAVE_Send_Ack_Message_To_Master(void)
{
    SCI_Transfer_Port = Ack_message_data_G;
}

/*-----*/
SCI_D_SLAVE_Process_Tick_Message()

The ticks messages are crucial to the operation of this S-C
scheduler: the arrival of a tick message (at regular intervals)
invokes the 'Update' ISR, that drives the scheduler.

The tick messages themselves may contain data. These data are
extracted in this function.

/*-----*/

```

```

void SCI_D_SLAVE_Process_Tick_Message(void)
{
    // Set up port for reading
    SCI_Transfer_Port = 0xFF;

    // Read the data
    Tick_message_data_G = SCI_Transfer_Port;
}

/* -----
SCI_D_SLAVE_Watchdog_Init()
This function sets up the watchdog timer.

If the Master fails (or other error develop),
no tick messages will arrive, and the scheduler
will stop.

To detect this situation, we have a (hardware) watchdog
running in the Slave. This watchdog - which should be set to
overflow at around 100 ms - is used to set the system into a
known (safe) state. The Slave will then wait (indefinitely)
for the problem to be resolved.

NOTE: The Slave will not be generating Ack messages in these
circumstances. The Master (if running) will therefore be aware
that there is a problem.

-----*/
void SCI_D_SLAVE_Watchdog_Init(void)
{
    // INIT NOT REQUIRED FOR 1232 EXTERNAL WATCHDOG
    // - May be required with different watchdog hardware
    //
    // Edit as required
}

/* -----
SCI_D_SLAVE_Watchdog_Refresh()

Feed the external (1232) watchdog.

Timeout is between ~60 and 250 ms (hardware dependent)

HARDWARE: Assumes external 1232 watchdog

-----*/

```

```

void SCI_D_SLAVE_Watchdog_Refresh(void) reentrant
{
    static bit WATCHDOG_state;

    // Change the state of the watchdog pin
    if (WATCHDOG_state == 1)
    {
        WATCHDOG_state = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state = 1;
        WATCHDOG_pin = 1;
    }
}

/* -----
SCI_D_SLAVE_Enter_Safe_State()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Master node fails
(3) The network has an error
(4) Tick messages are delayed for any other reason

Try to ensure that the system is in a 'safe' state in these
circumstances.

----- */

void SCI_D_SLAVE_Enter_Safe_State(void)
{
    // USER DEFINED - Edit as required
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/* -----
----- END OF FILE -----
----- */

```

Listing 26.11 Part of the software for a simple traffic system (Slave node).

[Note that data transfer between Master and Slave (and vice versa) is supported in this version.]

Further reading

Shared-clock schedulers using the UART

Introduction

As we discussed in Chapter 18, UART-based data transfer is a straightforward proposition with the 8051 family, allowing information to be transferred to an embedded application from a desktop PC and vice versa. In this chapter we demonstrate that the UART may also be used as the hardware foundation in a simple, low-cost technique for linking two or more microcontrollers together.

Note that these patterns are suitable for both ‘local’ and ‘distributed’ networks. By local networks, we mean applications containing multiple microcontrollers which are displaced by no more than a few centimetres; in most cases, the microcontrollers will be contained in the same system box; indeed, they will usually be mounted on the same PCB. In distributed networks, the microcontrollers may be displaced by considerable distances; in the case of RS-485 networks, for example, displacements of up to a kilometre are possible and even longer distances can be achieved if ‘repeater’ boards are added at kilometre intervals (e.g. see Sivasothy, 1998).

SCU SCHEDULER (LOCAL)

Context

- You are developing an embedded application using more than one member of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a scheduler.

Problem

How do you schedule tasks on (and transfer data over) a local network of two (or more) 8051 microcontrollers connected together via their UARTs?

Background

See Chapter 25 for general background material on shared-clock schedulers.

Solution

We consider how a UART-based, shared-clock scheduler can be designed in this section.

The basis of the approach

All the shared-clock schedulers we discuss in this book have the same underlying architecture (Figure 27.1).

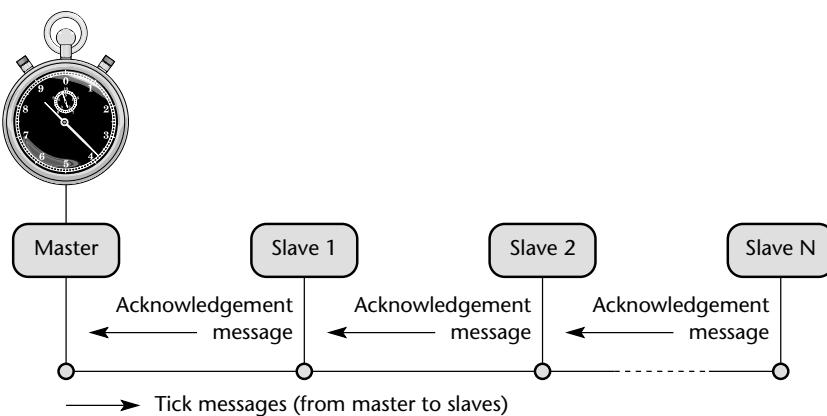


FIGURE 27.1 The architecture of all the shared-clock schedulers discussed in this book

Here we have one, accurate clock on the Master node in the network. This clock is used to drive the scheduler in the Master node in exactly the manner discussed in Part C.

The Slave nodes also have schedulers: however, the interrupts used to drive these schedulers are derived from ‘tick messages’ generated by the Master. Thus, in the UART-based network we are concerned with in this pattern, the Slave node(s) will have a S-C scheduler driven by the ‘receive’ interrupts generated through the receipt of a byte of data sent by the Master.

Note that the ability to generate an interrupt (in a Slave) on receipt of a data byte by the UART makes the underlying scheduler operation an extremely simple two-stage process:

- 1** Timer overflow in the Master causes the scheduler ‘Update’ function to be invoked. This, in turn, causes a byte of data to be sent (via the UART) to all Slaves:

```
void MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    ...
    MASTER_Send_Tick_Message(...);
    ...
}
```

- 2** When these data have been received all Slaves generate an interrupt; this invokes the ‘Update’ function in the Slave schedulers. This, in turn, causes one Slave to send an ‘Acknowledge’ message back to the Master (again via the UART).

```
void SLAVE_Update(void) interrupt INTERRUPT_UART_Rx_Tx
{
    ...
    SLAVE_Send_Ack_Message_To_Master();
    ...
}
```

The message structure

The design of the message structure to be used in a UART-based network requires some care.

For example, consider Figure 27.2. Here we will assume that we wish to control and monitor three hydraulic actuators to control the operation of a mechanical excavator.

Suppose we wish to adjust the angle of Actuator A to 90°; how do we do this? Immediately the 8-bit nature of the UART becomes a limitation, because we need to send a message that identifies both the node to be adjusted and the angle itself.

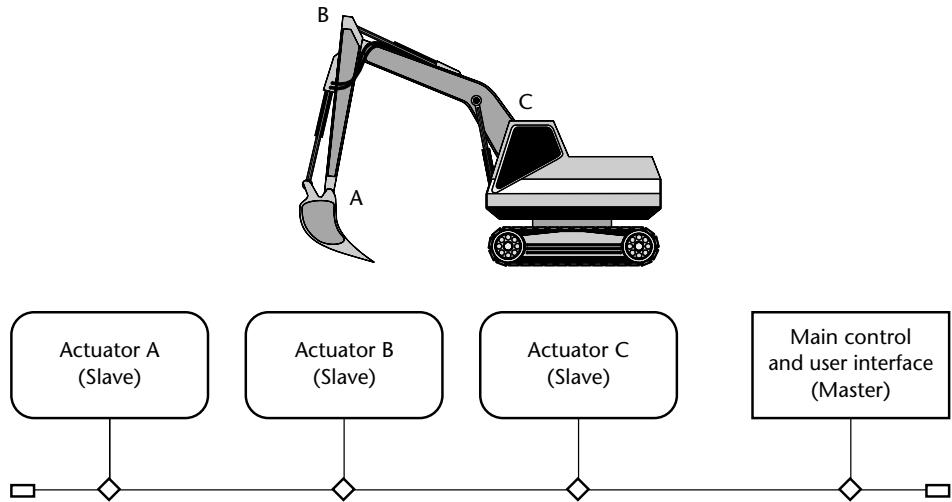


FIGURE 27.2 Controlling and monitoring the operation of a mechanical excavator with Slave nodes at three locations (A, B and C), plus a Master node in the cabin

There is no ideal way of addressing this problem. Here, we adopt the following solution:

- Each Slave is given a unique ID (0x01 to 0xFF).
- Each tick message from the Master is two bytes long; *these two bytes are sent one tick interval apart*. The first byte is an ‘Address byte’, containing the ID of the Slave to which the message is addressed. The second byte is the ‘Message byte’ and contains the message data.
- All Slaves generate interrupts in response to *each* byte of *every* tick message.
- Only the Slave to which a tick message is addressed will reply to the Master; this reply takes the form of an acknowledge message.
- Each acknowledge message from a Slave is two bytes long, the two bytes are, again, sent one tick interval apart. The first byte is an ‘Address byte’, containing the ID of the Slave from which the message is sent. The second byte is the ‘Message byte’ and contains the message data.
- For data transfers requiring more than a single byte of data, multiple messages must be sent (see **DATA UNION** [page 712] for useful techniques).

The transfer of messages is illustrated in Figure 27.3.

There remains one further problem; we need to be able to distinguish between ‘Address bytes’ and ‘Data bytes’, if we are to avoid possible confusion.

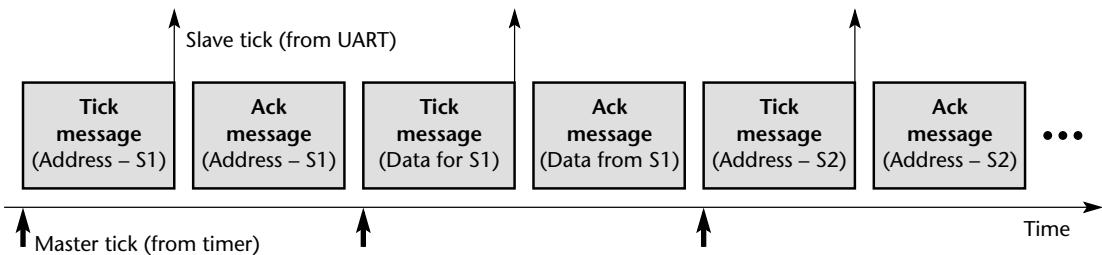


FIGURE 27.3 The communication between the Master and (two) Slave nodes in a UART-based S-C network

We do this by making use of the fact that the 8051 allows transmission of 9-bit serial data (Table 27.1).

TABLE 27.1 The structure of 9-bit serial messages

Description	Size (bits)
Data	9 bits
Start bit	1 bit
Stop bit	1 bit
Total	11 bits / message

In this configuration (typically, the UART used in Mode 3), 11 bits are transmitted / received. Note that the ninth bit is transmitted via bit TB8 in the register SCON, and is received as bit RB8 in the same register. In this mode, the baud rate is controlled as discussed in Chapter 18.

In the code examples presented here and included on the CD, address bytes are identified by setting the ‘command bit’ (TB8) to 1; data bytes set this bit to 0.

Determining the required baud rate

The timing of timer ticks in the Master is set to a duration such that one byte of a tick message can be sent (and one byte of an acknowledge message received) between ticks. Clearly, this duration depends on the network baud rate.

As already discussed, we will use a 9-bit protocol. Taking into account start and stop bits, we require 22 bits (11 for tick message, 11 for acknowledge message) per scheduler tick; that is, the required baud rate is: (scheduler ticks / second) \times 22.

For example, if we require 1 ms ticks, this equates to a minimum baud rate of 22,000 baud: the ‘standard’ baud rate of 28,800 baud provides a good safety margin.

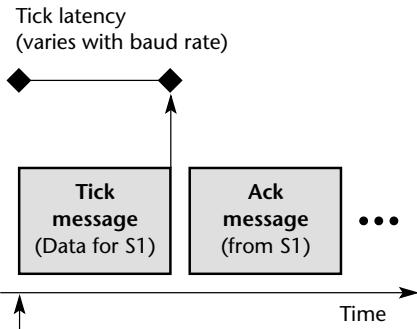


FIGURE 27.4 The latency between tick generation on the Master and on the Slaves

Note that use of ‘standard’ baud rates may help during debugging, since (for example) a PC can be easily used to monitor network traffic. However, standard rates are not otherwise necessary: any baud rate compatible with all the network nodes may be used. This can have advantages; for example, with the UART used in Mode 2, high-speed 9-bit serial operations are supported, but baud-rate timing does not require use of a timer. For example, a 12 Mhz / 12 oscillation per cycle 8051 can generate 375,000 baud signals in Mode 2.

Note also that, as with any shared-clock network, there is a delay between the timer on the Master and the UART-based interrupt on the Slave (Figure 27.4). This delay arises because the Slave interrupt is generated at the end of the tick message. In the absence of network errors, this delay is fixed and derives largely from the time taken to transmit a byte via the UART; that is, it varies with the baud rate. As discussed earlier, most shared-clock applications employ a baud rate of at least 28,800 baud: this gives a tick latency of approximately 0.4 ms. At 375,000 baud, this latency becomes approximately 0.03 ms.

Note that this latency is fixed and can be accurately predicted on paper and then confirmed in simulation and testing. If precise synchronization of Master and Slave processing is required, then please note that:

- All the Slaves operate – within the limits of measurement – precisely in step.
- To bring the Master in step with the Slaves, it is necessary only to add a short delay in the Master ‘Update’ function.

Node hardware

Typical hardware for a network node is shown in Figure 27.5.

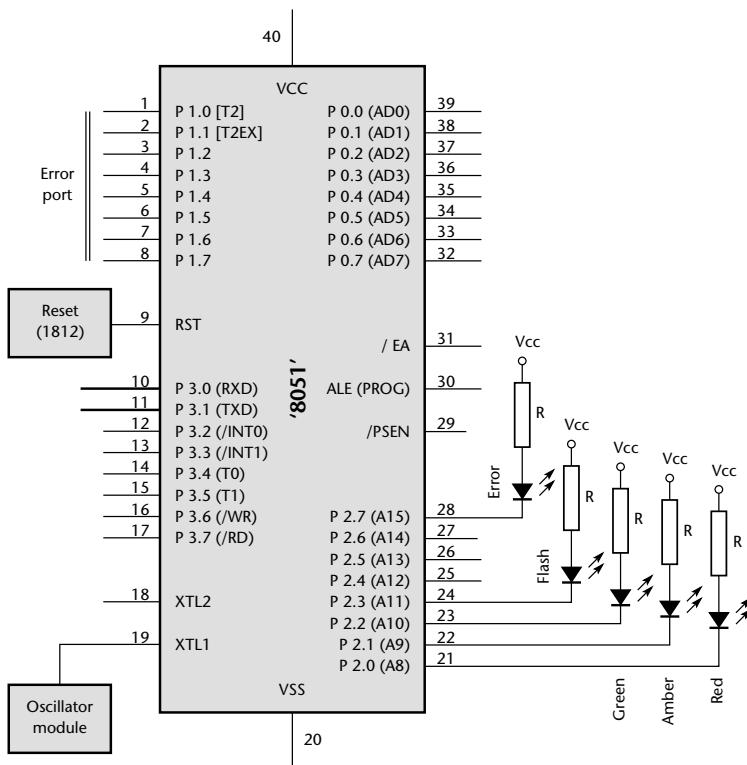


FIGURE 27.5a Possible hardware connections for local SCU networks, using an external 1232 watchdog

Network wiring

Figure 27.6 shows how a two-node network should be connected, while Figure 27.7 shows the connections for a network with three or more Slaves. Note that all cables should be kept as short as possible; ideally, all microcontrollers should be located on the same PCB.

Hardware resource implications

Clearly, the main hardware resource required is the UART; refer to the second example at the end of this pattern for techniques that may allow you to use this pattern in circumstances where a UART is also required for data transfers to a PC.

Note that this pattern does not necessarily require the use of a timer for baud-rate generation; see 'Solution' for details.

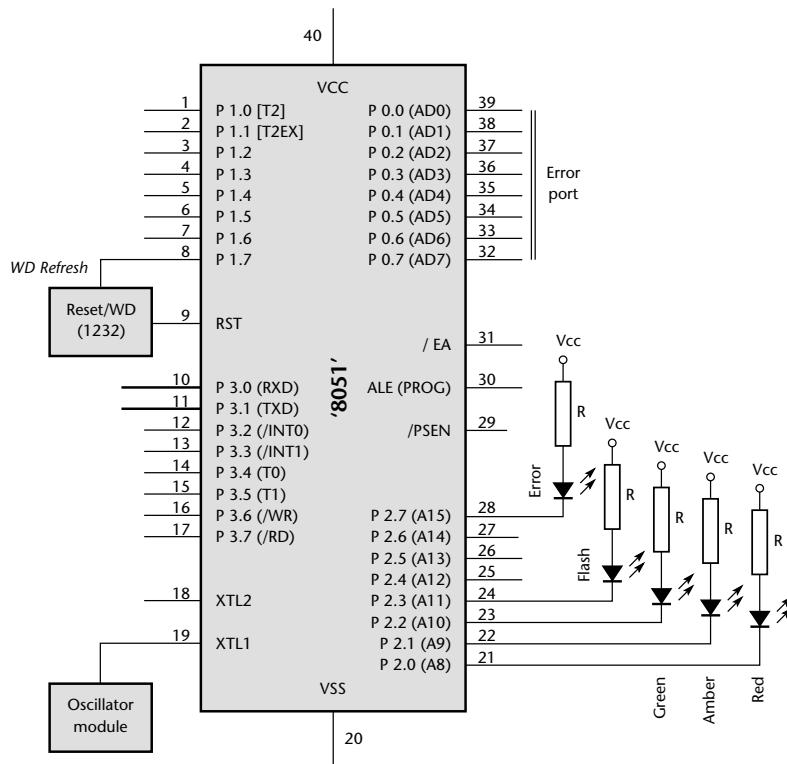


FIGURE 27.5b Possible hardware connections for local SCU networks, assuming an on-chip watchdog

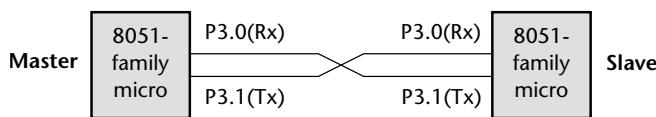


FIGURE 27.6 Inter-node connections in a two-node (local) SCU scheduler

Reliability and safety implications

Many of the reliability and safety considerations raised in connection with **SCI SCHEDULER (TICK)** [page 554] also apply here.

Portability

Can be used with the whole 8051 family and many other microcontrollers / microprocessor / DSP devices.

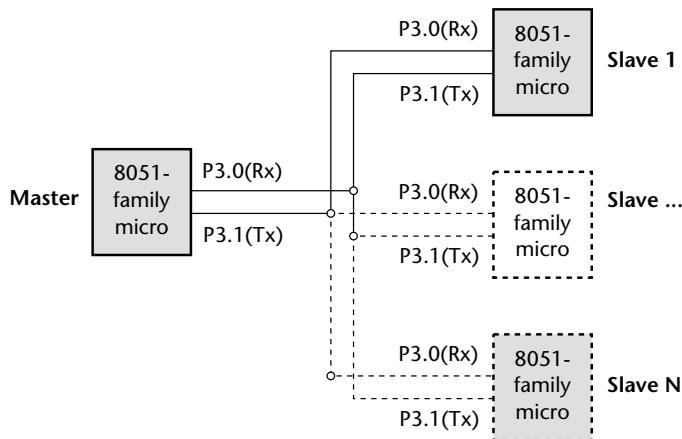


FIGURE 27.7 Inter-node connections in a three-node (local) SCU scheduler

Overall strengths and weaknesses

- ☺ A simple scheduler for local systems with two or more 8051 microcontrollers.
- ☺ All necessary hardware is part of the 8051 core: as a result, the technique is very portable within this family.
- ☺ Easy to implement with minimal CPU and memory overheads.
- ☺ The UART supports byte-based communications only: data transfer between Master and Slaves (and vice versa) is limited to 0.5 bytes per clock tick.
- ☹ Uses an important hardware resource (the UART).
- ☹ Most error detection / correction must be carried out in software.
- ☹ This pattern is not suitable for distributed systems.

Related patterns and alternative solutions

See **SCU SCHEDULER** [page 531] for an architecture suitable for use in multi-drop networks and with the ability to transfer larger amounts of data between network nodes.

Example: A local UART scheduler library

We present a simple code library in this example (Listings 27.1 and 27.2).

The hardware shown in Figure 27.5 may be used here.

Master/software

```

/* -----
   SCU_Am.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.

*** MASTER NODE ***
*** CHECKS FOR SLAVE ACKNOWLEDGEMENTS ***
*** Local / RS-232 version (no 'enable' support) ***
*** Uses 1232 watchdog timer ***
*** Both Master and Slave share the same tick rate (5 ms) ***

----- */

#include "Main.h"
#include "Port.h"

#include "SCU_Am.H"
#include "Delay_T0.h"
#include "TLight_B.h"

// ----- Public variable definitions -----
tByte Tick_message_data_G[NODE_OF_SLAVES] = {'M'};
tByte Ack_message_data_G[NODE_OF_SLAVES];

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// ----- Private variable definitions -----
static tByte Current_slave_index_G = 0;
static bit First_ack_G = 1;
static bit WATCHDOG_state_G = 0

// ----- Private function prototypes -----
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;

// ----- Private constants -----

// Slave IDs may be any NON-ZERO tByte value (all must be different)
// NOTE: ID 0x00 is for error handling.
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)

// Adjust (if required) to increase interval between network resets
// (in the event of sustained network error)
#define SLAVE_RESET_INTERVAL 0U

// ----- Private variables -----

static tWord Slave_reset_attempts_G[NUMBER_OF_SLAVES];

// See MAIN_SLAVE_IDs[] above
static tByte Current_Slave_IDs_G[NUMBER_OF_SLAVES] = {0};

static bit Message_byte_G = 1;

/*-----*/
SCU_A_MASTER_Init_T1_T2()

Scheduler initialization function. Prepares scheduler data
structures and sets up timer interrupts at required rate.
You must call this function before using the scheduler.

BAUD_RATE - The required baud rate.

*/
void SCU_A_MASTER_Init_T1_T2(const tWord BAUD_RATE)
{
    tByte Task_index;
    tByte Slave_index;

    // No interrupts (yet)
    EA = 0;

    // Start the watchdog
    SCU_A_MASTER_Watchdog_Init();

    Network_error_pin = NO_NETWORK_ERROR;
}

```

```
// ----- Set up the scheduler -----
// Sort out the tasks
for (Task_index = 0; Task_index < SCH_MAX_TASKS; Task_index++)
{
    SCH_Delete_Task(Task_index);
}

// Reset the global error variable
// - SCH_Delete_Task() will generate an error code,
//   (because the task array is empty)
Error_code_G = 0;

// We allow any combination of ID numbers in slaves
// - but reserve ID 0x00
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];
}

// ----- Set the baud rate (begin) -----
PCON &= 0x7F; // Set SMOD bit to 0 (don't double baud rates)

// Receiver enabled
// *9-bit data*, 1 start bit, 1 stop bit, variable baud rate
// (asynchronous)
SCON = 0xD2;

TMOD |= 0x20; // T1 in mode 2, 8-bit auto reload

TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
    / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));

TL1 = TH1;
TR1 = 1; // Run the timer
TI = 1; // Send first character (dummy)
// ----- Set the baud rate (end) -----

// Serial interrupt *NOT* enabled
ES = 0;
// ----- Set up the serial link (end) -----

// ----- Set up Timer 2 (begin) -----
// Crystal is assumed to be 12 MHz, 12 osc/increment
//
// The Timer 2 resolution is 0.000001 seconds (1 µs)
// The required Timer 2 overflow is 0.005 seconds (5 ms)
```

```

// - this takes 5000 timer ticks
// Reload value is 65536 - 5000 = 60536 (dec) = 0xEC78

T2CON = 0x04;    // load Timer 2 control register
T2MOD = 0x00;    // load Timer 2 mode register

TH2    = 0xEC;   // load timer 2 high byte
RCAP2H = 0xEC;   // load timer 2 reload capture reg, high byte
TL2    = 0x78;   // load timer 2 low byte
RCAP2L = 0x78;   // load timer 2 reload capture reg, low byte

ET2    = 1;      // Timer 2 interrupt is enabled

TR2    = 1;      // Start Timer 2
// ----- Set up Timer 2 (end) -----
}

/*-----*
SCU_A_MASTER_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

NOTE: Delays here (2 ms) assume a baud rate of at least 9600 baud.
You will need to fine-tune this code if you opt for a lower baud
rate.
-*-----*/
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;

    // Refresh the watchdog
    SCU_A_MASTER_Watchdog_Refresh();

    // Place system in 'safe state'
    SCU_A_MASTER_Enter_Safe_State();

    // Report error as we wait to start
    Network_error_pin = NETWORK_ERROR;
}

```

```
Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
SCH_Report_Status(); // Sch not yet running - do this manually

// Pause here (300 ms), to time-out all the slaves
// (This is the means by which we synchronize the network)
for (i = 0; i < 100; i++)
{
    Hardware_Delay_T0(30);
    SCU_A_MASTER_Watchdog_Refresh();
}

// Currently disconnected from all slaves
Num_active_slaves = 0;

// After the initial (long) delay, all (operational) slaves will
// have timed out.

// All operational slaves will now be in the 'READY TO START' state
// Send them a (single-byte) message to get them started
// DO THIS TWICE IN A ROW, WITH COMMAND BIT
// This cannot happen under normal circumstances
Slave_index = 0;
do {
    // Refresh the watchdog
    SCU_A_MASTER_Watchdog_Refresh();

    // Clear 'first byte' flag
    First_byte = 0;

    // Find the slave ID for this slave
    Slave_ID = (tByte) Current_Slave_IDS_G[Slave_index];

    // Send a 'Slave ID' message
    TI = 0;
    TB8 = 1; // Set command bit
    SBUF = Slave_ID;

    // Wait to give slave time to reply
    Hardware_Delay_T0(5);

    // Check we had a reply
    if (RI == 1)
    {
        // Get the reply data
        Id = (tByte) SBUF;
        RI = 0;

        // Check reply - with command bit
        if ((Id == Slave_ID) && (RB8 == 1))

```

```
{  
    First_byte = 1;  
}  
}  
  
// Send second byte  
TI = 0;  
TB8 = 1;  
SBUF = Slave_ID;  
  
// Wait to give slave time to reply  
Hardware_Delay_T0(5);  
  
// Check we had a reply  
Slave_replied_correctly = 0;  
if (RI != 0)  
{  
    // Get the reply data  
    Id = (tByte) SBUF;  
    RI = 0;  
  
    if ((Id == Slave_ID) && (RB8 == 1) && (First_byte == 1))  
    {  
        Slave_replied_correctly = 1;  
    }  
}  
  
if (Slave_replied_correctly)  
{  
    // Slave responded correctly  
    Num_active_slaves++;  
    Slave_index++;  
}  
else  
{  
    // Slave did not reply correctly  
    // - try to switch to backup device (if available)  
    if (Current_Slave_IDs_G[Slave_index] !=  
        BACKUP_SLAVE_IDS[Slave_index])  
    {  
        // There is a backup available: switch to backup and try  
        // again  
        Current_Slave_IDs_G[Slave_index] =  
        BACKUP_SLAVE_IDS[Slave_index];  
    }  
}
```

```

        else
        {
            // No backup available (or backup failed too) - have to
            // continue
            // NOTE: we don't abort the loop, to allow for more
            // flexible
            // error handling (below).
            Slave_index++;
        }
    }

} while (Slave_index < NUMBER_OF_SLAVES);

// DEAL WITH CASE OF MISSING SLAVE(S) HERE ...
if (Num_active_slaves < NUMBER_OF_SLAVES)
{
    // User-defined error handling here...
    // 1 or more slaves have not replied
    // NOTE: In some circumstances you may wish to abort if slaves
    // are missing
    // - or reconfigure the network.

    // Here we display error and shut down the network
    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;
}

// Get ready to send first tick message
Message_byte_G = 1;
First_ack_G = 1;
Current_slave_index_G = 0;

// Start the scheduler
EA = 1;
}

/*-----*
 * SCU_A_MASTER_Update_T2
 *
 * This is the scheduler ISR. It is called at a rate determined by
 * the timer settings in SCU_A_MASTER_Init_T2(). This version is
 * triggered by Timer 2 interrupts: timer is automatically reloaded.
-*-----*/

```

```
void SCU_A_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Task_index;
    tByte Previous_slave_index;

    TF2 = 0; // Must manually clear this.

    // Refresh the watchdog
    SCU_A_MASTER_Watchdog_Refresh();

    // Default
    Network_error_pin = NO_NETWORK_ERROR;

    // Keep track of the current slave
    // FIRST VALUE IS 0
    Previous_slave_index = Current_slave_index_G;

    // Assume 2-byte messages sent and received
    // - it takes two ticks to deliver each message
    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;
    }
}

// Check that the appropriate slave responded to the previous message:
// (if it did, store the data sent by this slave)
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
    // switch to a backup device (if one is available) as we reset
    // the network
    //
    // NOTE: We don't do this every tick (or the the net will be
    // constantly reset)
    //
    // Choose a value of SLAVE_RESET_INTERVAL to allow resets (say)
    // every 5 seconds
}
```

```

if (++Slave_reset_attempts_G[Previous_slave_index] >=
SLAVE_RESET_INTERVAL)
{
    SCU_A_MASTER_Reset_the_Network();
}
}

else
{
    // Reset this counter
    Slave_reset_attempts_G[Previous_slave_index] = 0
}

// Send 'tick' message to all connected slaves
// (sends one data byte to the current slave)
SCU_A_MASTER_Send_Tick_Message(Current_slave_index_G);

// NOTE: calculations are in *TICKS* (not milliseconds)
for (Task_index = 0; Task_index < SCH_MAX_TASKS; Task_index++)
{
    // Check if there is a task at this location
    if (SCH_tasks_G[Task_index].pTask)
    {
        if (SCH_tasks_G[Task_index].Delay == 0)
        {
            // The task is due to run
            SCH_tasks_G[Task_index].RunMe += 1; // Increment the RunMe
            flag

            if (SCH_tasks_G[Task_index].Period)
            {
                // Schedule periodic tasks to run again
                SCH_tasks_G[Task_index].Delay =
                SCH_tasks_G[Task_index].Period;
            }
        }
        else
        {
            // Not yet ready to run: just decrement the delay
            SCH_tasks_G[Task_index].Delay -= 1;
        }
    }
}

/*-----*
SCU_A_MASTER_Send_Tick_Message()
-----*/

```

This function sends a tick message, over the USART network. The receipt of this message will cause an interrupt to be generated in the slave(s): this invoke the scheduler 'update' function in the slave(s).

```
-----*/
void SCU_A_MASTER_Send_Tick_Message(const tByte SLAVE_INDEX)
{
    tLong Timeout;

    // Find the slave ID for this slave
    tByte Slave_ID = (tByte) Current_Slave_IDS_G[SLAVE_INDEX];

    // Sending one byte of data at a time, depending on index value
    // If Message_byte_G is 0, send first byte (the slave ID)
    if (Message_byte_G == 0)
    {
        Timeout = 0;
        while ((++Timeout) && (TI == 0));

        if (Timeout == 0)
        {
            // UART did not respond - error
            Error_code_G = ERROR_USART_TI;
            return;
        }

        TI = 0;
        TB8 = 1; // Set 'Command' bit
        SBUF = Slave_ID;
    }
    else
    {
        // Message_byte_G is 1, send the data byte
        Timeout = 0;
        while ((++Timeout) && (TI == 0));

        if (Timeout == 0)
        {
            // usart did not respond - error
            Error_code_G = ERROR_USART_TI;
            return;
        }

        TI = 0;
        TB8 = 0;
        SBUF = Tick_message_data_G[SLAVE_INDEX];
    }
}
```

```

    // Data sent - return
}

/*-----*
SCU_A_MASTER_Process_Ack()

Make sure the slave (SLAVE_ID) has acknowledged the previous
message that was sent. If it has, extract the message data
from the USART hardware: if not, call the appropriate error
handler.

Slave_index - The index of the slave.

RETURNS: RETURN_NORMAL - Ack received (data in Ack_message_data_G)
RETURN_ERROR - No ack received (-> no data)

-*-----*/
bit SCU_A_MASTER_Process_Ack(const tByte Slave_index)
{
    tByte Message_contents;
    tByte Slave_ID;

    // First time this is called there is no ack tick to check
    // - we simply return 'OK'
    if (First_ack_G)
    {
        First_ack_G = 0;
        return RETURN_NORMAL;
    }

    // Find the slave ID for this slave
    Slave_ID = (tByte) Current_Slave_IDS_G[Slave_index];

    // Data should already be in the buffer
    if (RI == 0)
    {
        // Slave has not replied to last tick message

        // Set error LED
        Network_error_pin = NETWORK_ERROR;

        return RETURN_ERROR;
    }

    // There is data - get it
    Message_contents = (tByte) SBUF;
    RI = 0;

    // This is the reply to the last message
    // - reverse the message byte interpretation

```

```

    if (Message_byte_G == 1)
    {
        // Check the 'command bit' is set
        if (RB8 == 1)
        {
            // Check that the ID is correct
            if (Slave_ID == (tByte) Message_contents)
            {
                // Required Ack message was received
                return RETURN_NORMAL;
            }
        }

        // Something is wrong...
        // Set error LED
        Network_error_pin = NETWORK_ERROR;

        return RETURN_ERROR;
    }
    else
    {
        // There *ARE* data available
        // Extract the data from the slave message
        //

        // NOTE: We *assume* these data are OK
        // - you may wish to send crucial data twice, etc.
        Ack_message_data_G[Slave_index] = Message_contents;

        return RETURN_NORMAL; // return the slave data
    }
}

/*-----*
SCU_A_MASTER_Reset_the_Network()

This function resets (that ism restarts) the whole network.
-*-----*/
void SCU_A_MASTER_Reset_the_Network(void)
{
    EA = 0; // Disable interrupts
    while(1); // Watchdog will time out
}

/*-----*
SCU_A_MASTER_Shut_Down_the_Network()

This function shuts down the whole network.
*/

```

```

-----*/
void SCU_A_MASTER_Shut_Down_the_Network(void)
{
    EA = 0; // Disable interrupts

    Network_error_pin = NETWORK_ERROR;
    SCH_Report_Status(); // Sch not running - do this manually

    while(1)
    {
        SCU_A_MASTER_Watchdog_Refresh();
    }
}

-----*/
SCU_A_MASTER_Enter_Safe_State()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Master node cannot detect a slave
(3) The network has an error

Try to ensure that the system is in a 'safe' state in these
circumstances.

-----*/
void SCU_A_MASTER_Enter_Safe_State(void)
{
    // USER DEFINED - Edit as required
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

-----*
SCU_A_MASTER_Watchdog_Init()

This function sets up the watchdog timer.

If the Master fails (or other error develop),
no tick messages will arrive, and the scheduler
will stop.

To detect this situation, we have a (hardware) watchdog
running in the slave. This watchdog - which should be set to
overflow at around 100 ms - is used to set the system into a
known (safe) state. The slave will then wait (indefinitely)
for the problem to be resolved.

NOTE: The slave will not be generating Ack messages in these
circumstances. The Master (if running) will therefore be aware
that there is a problem.

```

```

-*-----*/
void SCU_A_MASTER_Watchdog_Init(void)
{
    //
    // INIT NOT REQUIRED FOR 1232 EXTERNAL WATCHDOG
    // - May be required with different watchdog hardware
    // Edit as required
}
/*-----*/
SCU_A_MASTER_Watchdog_Refresh()
Feed the external (1232) watchdog.

Timeout is between ~60 and 250 ms (hardware dependent)

Assumes external 1232 watchdog
-*-----*/
void SCU_A_MASTER_Watchdog_Refresh(void) reentrant
{
    //
    // Change the state of the watchdog pin
    if (WATCHDOG_state_G == 1)
    {
        //
        WATCHDOG_state_G = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        //
        WATCHDOG_state_G = 1;
        WATCHDOG_pin = 1;
    }
}

/*-----*/
---- END OF FILE -----
-*-----*/

```

Listing 27.1 Part of a local UART scheduler library (Master node)

Slave software

```

/*-----*/
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"

// ----- Public variable definitions -----
// Data sent from the master to this slave
tByte Tick_message_data_G;

// Data sent from this slave to the master
// - data may be sent on, by the master, to another slave
tByte Ack_message_data_G = 'S';

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// ----- Private function prototypes -----
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;

// ----- Private constants -----
// Each slave must have a unique (non-zero) ID
#define SLAVE_ID 0x31

#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)

// ----- Private variables -----
```

```

static bit Message_byte_G;
static bit WATCHDOG_state_G = 0
static tByte Message_ID_G = 0

/* -----
SCU_A_SLAVE_Init_T1()

Scheduler initialization function. Prepares scheduler
data structures and sets up timer interrupts at required rate.
Must call this function before using the scheduler.

BAUD_RATE - The required baud rate

*/
void SCU_A_SLAVE_Init_T1(const tWord BAUD_RATE)
{
    tByte i;

    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // Set the network error pin (reset when tick message received)
    Network_error_pin = NETWORK_ERROR;

    // Ready for first tick message
    Message_byte_G = 1;

    // ----- Set the baud rate (begin) -----
    PCON &= 0x7F;    // Set SMOD bit to 0 (don't double baud rates)

    // receiver enabled
    // 9-bit data, 1 start bit, 1 stop bit, variable baud rate
    // (asynchronous)
    SCON = 0xD2;

    TMOD |= 0x20;    // T1 in mode 2, 8-bit auto reload

    TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
                           / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));

    TL1 = TH1;
    TR1 = 1;    // Run the timer
    TI = 1;    // Send first character (dummy)
}

```

```
// ----- Set the baud rate (end) -----
// Interrupt enabled
// (Both receiving and SENDING a byte will generate a serial interrupt)
// Global interrupts not yet enabled.
ES = 1;

// Start the watchdog
SCU_A_SLAVE_Watchdog_Init();
}

/*-----*
SCU_A_SLAVE_Start()

Starts the slave scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

*/
void SCU_A_SLAVE_Start(void)
{
    tByte Command = 0;
    tByte Message_byte;
    tByte Count = 0;
    bit Slave_started = 0;

    // Disable interrupts
    EA = 0;

    // We can be at this point because:
    // 1. The network has just been powered up
    // 2. An error has occurred in the Master, and it is not generating
    // ticks
    // 3. The network has been damaged and no ticks are being received
    // by this slave
    //
    // Try to make sure the system is in a safe state...
    SCU_A_SLAVE_Enter_Safe_State();

    // NOTE: Interrupts are disabled here
    Count = 0;

    Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
    SCH_Report_Status(); // Sch not yet running - do this manually

    // Now wait (indefinitely) for appropriate signals from the master
    do {
        // Wait for tick messages (byte 1), all bits set to 0, to be
        // received
```

```

do {
    SCU_A_SLAVE_Watchdog_Refresh(); // Must keep feeding the
                                    watchdog
} while (RI == 0);

Message_byte = (tByte) SBUF;
RI = 0;
// Must get two ID messages in a row...
// (with command bit)
// Ack each one
if ((Message_byte == (tByte) SLAVE_ID) && (RB8 == 1))
{
    Count++;

    // Received message for this slave - send ack
    TI = 0;
    TB8 = 1; // Set command bit
    SBUF = (tByte) SLAVE_ID;
}
else
{
    Count = 0;
}
} while (Count < 2);

// Start the scheduler
EA = 1;
}

/*-----*
SCU_A_SLAVE_Update

This is the scheduler ISR. It is called at a rate
determined by the timer settings in SCU_A_SLAVE_Init().

This Slave is triggered by USART interrupts.

-*-----*/
void SCU_A_SLAVE_Update(void) interrupt INTERRUPT_UART_Rx_Tx
{
    tByte Index;

    if (RI == 1) // Must check this.
    {
        // Default
        Network_error_pin = NO_NETWORK_ERROR;

        // Two-byte messages are sent (Ack) and received (Tick)
        // - it takes two sched ticks to process each message
    }
}

```

```

// 
// Keep track of the current byte
if (Message_byte_G == 0)
{
    Message_byte_G = 1;
}
else
{
    Message_byte_G = 0;
}

// Check tick data - send ack if necessary
// NOTE: 'START' message will only be sent after a 'time out'
if (SCU_A_SLAVE_Process_Tick_Message() == SLAVE_ID)
{
    SCU_A_SLAVE_Send_Ack_Message_To_Master();

    // Feed the watchdog ONLY when a *relevant* message is
    // received
    // (noise on the bus, etc, will not stop the watchdog...)
    //
    // START messages will NOT refresh the slave
    // - Must talk to every slave at regular intervals
    SCU_A_SLAVE_Watchdog_Refresh();
}

// NOTE: calculations are in *TICKS* (not milliseconds)
for (Index = 0; Index < SCH_MAX_TASKS; Index++)
{
    // Check if there is a task at this location
    if (SCH_tasks_G[Index].pTask)
    {
        if (SCH_tasks_G[Index].Delay == 0)
        {
            // The task is due to run
            SCH_tasks_G[Index].RunMe = 1; // Set the run flag

            if (SCH_tasks_G[Index].Period)
            {
                // Schedule periodic tasks to run again
                SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
            }
        }
    else
    {
        // Not yet ready to run: just decrement the delay

```

```

        SCH_tasks_G[Index].Delay -= 1;
    }
}
}
RI = 0; // Reset the RI flag
}
else
{
// ISR call was triggered by TI flag, after last character was sent
// Must clear the TI flag
TI = 0;
}
}

/*-----*/

```

SCU_A_SLAVE_Send_Ack_Message_To_Master()

Slave must send and 'Acknowledge' message to the master, after tick messages are received. NOTE: Only tick messages specifically addressed to this slave should be acknowledged.

The acknowledge message serves two purposes:

- [1] It confirms to the master that this slave is alive & well.
- [2] It provides a means of sending data to the master and - hence - to other slaves.

NOTE: Direct data transfer between slaves is NOT possible.

```

-----*/
void SCU_A_SLAVE_Send_Ack_Message_To_Master(void)
{
// Sending one byte of data at a time, depending on index value
// If Message_byte_G is 0, send first byte (the slave ID)
if (Message_byte_G == 0)
{
TI = 0;
TB8 = 1; // Set 'Command' bit
SBUF = SLAVE_ID;
}
else
{
// Message_byte_G is 1, send the data byte
TI = 0;
TB8 = 0;
SBUF = Ack_message_data_G;
}
}
```

```

        // Data sent - return
    }

/*-----*/
SCU_A_SLAVE_Process_Tick_Message()

The ticks messages are crucial to the operation of this shared-clock
scheduler: the arrival of a tick message (at regular intervals)
invokes the 'Update' ISR, that drives the scheduler.

The tick messages themselves may contain data. These data are
extracted in this function.

/*-----*/
tByte SCU_A_SLAVE_Process_Tick_Message(void)
{
    tByte Data;

    // Try to get data byte
    if (RI == 0)
    {
        // No data - something is wrong

        // Set the error flag bit
        Network_error_pin = NETWORK_ERROR;

        // Return slave ID 0
        return 0x00;
    }

    // There *ARE* data available
    Data = (tByte) SBUF;
    RI = 0; // Clear RI flag

    // What we do with this message depends if it is a first or second
    byte
    if (Message_byte_G == 0)
    {
        // This is (should be) an ID byte
        Message_ID_G = Data;

        if (RB8 == 0)
        {
            Message_ID_G = 0; // Command bit should be set
        }
    }
    else
    {
        // This is (should be) a data byte
        // - Command bit should not be set
    }
}

```

```

    if ((Message_ID_G == SLAVE_ID) && (RB8 == 0))
    {
        Tick_message_data_G = Data;
    }
    else
    {
        // Something is wrong - set Message_ID to 0
        Message_ID_G = 0;

        // Set the error flag bit
        Network_error_pin = NETWORK_ERROR;
    }
}

return Message_ID_G;
}

/*-----*
SCU_A_SLAVE_Watchdog_Init()
This function sets up the watchdog timer.

If the Master fails (or other error develop),
no tick messages will arrive, and the scheduler
will stop.

To detect this situation, we have a (hardware) watchdog
running in the slave. This watchdog - which should be set to
overflow at around 100 ms - is used to set the system into a
known (safe) state. The slave will then wait (indefinitely)
for the problem to be resolved.

NOTE: The slave will not be generating Ack messages in these
circumstances. The Master (if running) will therefore be aware
that there is a problem.

*-----*/
void SCU_A_SLAVE_Watchdog_Init(void)
{
    // INIT NOT REQUIRED FOR 1232 EXTERNAL WATCHDOG
    // - May be required with different watchdog hardware
    //
    // Edit as required
}

/*-----*
SCU_A_SLAVE_Watchdog_Refresh()

```

```

Feed the external (1232) watchdog.

Timeout is between ~60 and 250 ms (hardware dependent)

Assumes external 1232 watchdog

----- */
void SCU_A_SLAVE_Watchdog_Refresh(void) reentrant
{
    // Change the state of the watchdog pin
    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()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Master node fails, and no working backup is available
(3) The network has an error
(4) Tick messages are delayed for any other reason

Try to ensure that the system is in a 'safe' state in these
circumstances.

----- */
void SCU_A_SLAVE_Enter_Safe_State(void)
{
    // USER DEFINED - Edit as required
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

----- END OF FILE -----
----- */

```

Listing 27.2 Part of a local UART scheduler library (Slave node)

Example: A 3-node local scheduler

See CD for details.

Example: A 2-node local scheduler with backup Slave

See CD for details.

Example: Adding an additional UART

As we saw in Chapter 18, using scheduled ‘RS-232’ comms to transfer data to and from a PC is straightforward (Figure 27.8).

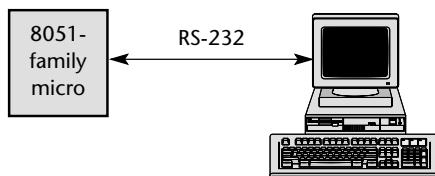


FIGURE 27.8 Using an RS-232 link to transfer data to and from a desktop PC

Suppose, however, that you are using a UART-based scheduler to link two microcontrollers and you also need to create a link to a desktop PC: in most cases, you will not have a second UART to set up the link to the PC (Figure 27.9).

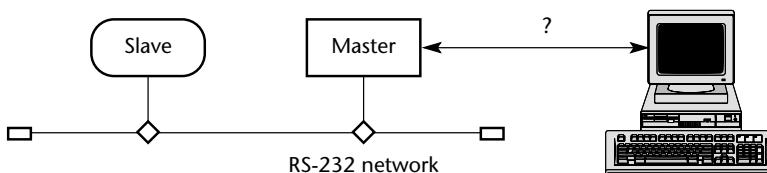


FIGURE 27.9 How do we use a microcontroller with a single UART to communicate with another microcontroller using a UART-based scheduler and to transfer data to and from a desktop PC over an RS-232 link?

There are several possibilities:

- Use an 8051 with more than one UART: for example, several of the Dallas microcontrollers have this facility, as do the Infineon c517 and c509.
- Add an external UART, such as the Maxim Max3110E (which has an SPI interface).
- Use CAN to link the nodes together and reserve the UART for communication with the PC: see **scc SCHEDULER** [page 677] for details.

- Use a combination of a UART scheduler and an interrupt scheduler (linked to one board): see Figure 27.10.

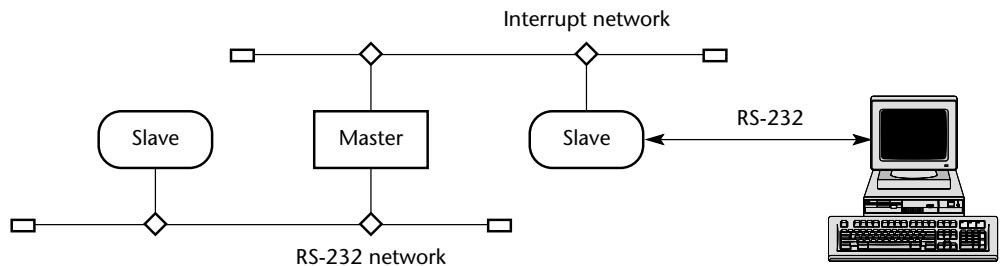


FIGURE 27.10 Using a combination of a UART scheduler and an interrupt scheduler

Further reading

SCU SCHEDULER (RS-232)

Context

- You are developing an embedded application using more than one member of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a scheduler.

Problem

How do you schedule tasks on (and transfer data over) a distributed network of two 8051 microcontrollers communicating using the RS-232 protocol?

Background

See Chapter 18 for background information on the RS-232 protocol.

See **SCU SCHEDULER (LOCAL)** [page 609] for details of the required software.

Solution

This pattern extends **SCU SCHEDULER (LOCAL)** [page 609]; the key difference between the two patterns is that **SCU SCHEDULER (RS-232)** allows the creation of two-node, distributed networks, by making use of RS-232-compatible transceiver hardware. Figure 27.11 shows one possible node configuration, where an external 1232 watchdog timer is employed.

Figure 27.12 shows a suitable node configuration where you have an on-board watchdog timer.

Hardware resource implications

Clearly, the main (microcontroller) hardware resource required is the UART; refer to the second example at the end of **SCU SCHEDULER (LOCAL)** [page 609] for techniques that may allow you to use this pattern in circumstances where a UART is also required for data transfers to a PC.

Note that this pattern does not necessarily require the use of a timer for baud-rate generation; see the ‘Solution’ section in **SCU SCHEDULER (LOCAL)** for details.

Reliability and safety implications

Many of the reliability and safety considerations raised in connection with **SCI SCHEDULER (TICK)** [page 554] also apply here.

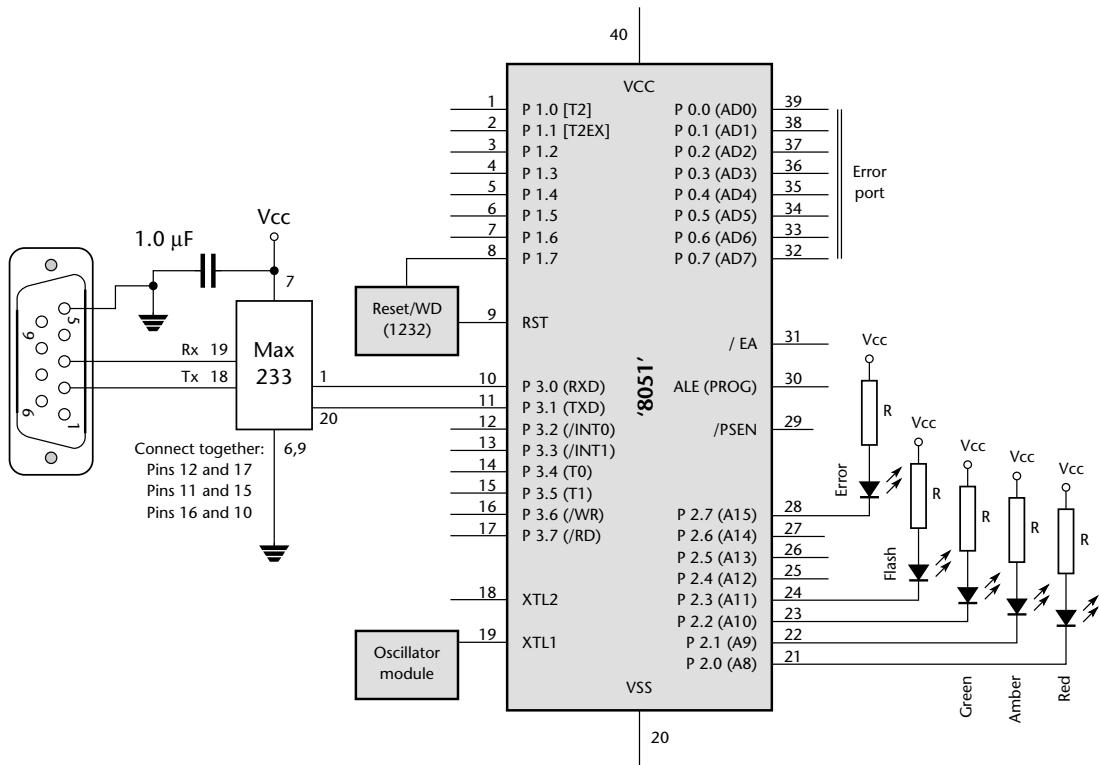


FIGURE 27.11 An example node configuration for UART-based scheduler, where an external 1232 watchdog timer is employed

Portability

Can be used with the whole 8051 family and many other microcontrollers / microprocessor / DSP devices.

Overall strengths and weaknesses

- ☺ A simple scheduler for distributed systems of two 8051 microcontrollers.
- ☺ All necessary hardware (apart from transceivers) is part of the 8051 core: as a result, the technique is very portable within this family.
- ☺ Easy to implement with minimal CPU and memory overheads.
- ☹ The UART supports byte-based communications only: data transfer between Master and Slaves (and vice versa) is limited to 0.5 bytes per clock tick.

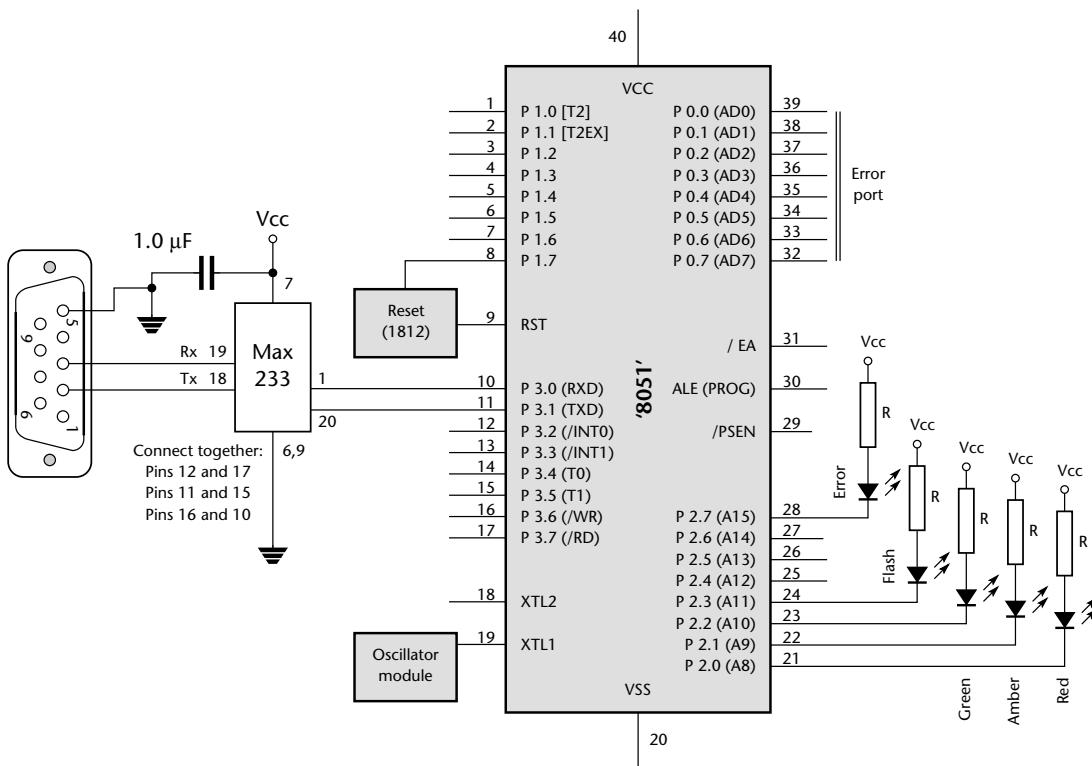


FIGURE 27.12 An example node configuration for UART-based scheduler, where an internal watchdog timer is employed

- (⌚) Uses an important hardware resource (the UART).
- (⌚) Most error detection / correction must be carried out in software.

Related patterns and alternative solutions

See **SCU SCHEDULER (RS-485)** [page 646] for a similar scheduler suitable for use in multi-drop networks.

See **SCC SCHEDULER** [page 677] for a scheduler suitable for use in multi-drop networks and with the ability to transfer larger amounts of data between network nodes.

Example: Traffic lights

Any of the two-node examples from **SCU SCHEDULER (LOCAL)** [page 609] may be applied here, using a hardware platform based on Figure 27.11 or Figure 27.12.

Further reading

Axelson, J. (1998) *Serial Port Complete*, Lakeview Research.

SCU SCHEDULER (RS-485)

Context

- You are developing an embedded application using more than one member of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a scheduler.

Problem

How do you schedule tasks on (and transfer data over) a distributed network of two (or more) 8051 microcontrollers communicating using the RS-485 protocol?

Background

The key characteristics of a shared-clock scheduler were discussed in Chapter 25.

Solution

The communications standard generally referred to as 'RS-485' is an electrical specification for what are often referred to as 'multi-point' or 'multi-drop' communication systems; for our purposes, this means applications that involve at least three nodes, each containing a microcontroller.

Note that the specification document (EIA/TIA-485-A) defines the electrical characteristics of the line and its drivers and receivers: this is the limit of the standard. Thus, unlike 'RS-232', there is no discussion of software protocols or of connectors.

We discussed the RS-232 protocol in connection with **scu SCHEDULER (rs-232)** [page 642] and **PC LINK (rs-232)** [page 362]. There are many similarities between RS-232 and RS-485 communication protocols:

- Both are serial standards.
- Both are in widespread use.
- Both involve – for our purposes – the use of an appropriate transceiver chip connected to a UART.
- Both involve very similar software libraries.

There are also some key differences:

- RS-232 is a single-wire standard (one signal line, per channel, plus ground). Electrical noise in the environment can lead to data corruption. This restricts the communication range to a maximum of around 30 metres and the data rate to around 115 kbaud (with recent drivers).

- RS-485 is a two-wire or differential communication standard. This means that, for each channel, two lines carry (1) the required signal and (2) the inverse of the signal. The receiver then detects the voltage *difference* between the two lines. Electrical noise will impact on both lines, and will cancel out when the difference is calculated at the receiver. As a result, an RS-485 network can extend as far as 1 km, at a data rate of 90 kbaud. Faster data rates (up to 10 Mbaud) are possible at shorter distances (up to 15 metres).
- RS-232 is a peer-to-peer communications standard. For our purposes, this means that it is suitable for applications that involve two nodes, each containing a microcontroller (or, as we saw in Chapter 18, for applications where one node is a desktop, or similar, PC).
- RS-485 is a ‘multi-point’ or ‘multi-drop’ communications standard. Larger RS-485 networks can have up to 32 ‘unit loads’: by using high-impedance receivers, you can have as many as 256 nodes on the network.
- RS-232 requires low-cost ‘straight’ cables, with three wires for fully duplex communications (Tx, Rx, ground).
- For full performance, RS-485 requires twisted-pair cables, with two twisted pairs, plus ground (and usually a screen). This cabling is more bulky and more expensive than the RS-232 equivalent.
- RS-232 cables do not require terminating resistors.
- RS-485 cables are usually used with 120Ω terminating resistors (assuming 24-AWG twisted pair cables) connected in parallel, at or just beyond the final node at *both* ends of the network (Figure 27.13). The terminations reduce voltage reflections that can otherwise cause the receiver to misread logic levels.
- RS-232 transceivers are simple and standard.
- Choice of RS-485 transceivers depends on the application. A common choice for basic systems is the Maxim⁵ Max489 family (see Figures 27.13 and 27.14). For increased reliability, the Linear Technology⁶ LTC1482, National Semiconductors⁷ DS36276 and the Maxim Max3080–89 series all have internal circuitry to protect against cable short-circuits. Also, the Maxim Max MAX1480 contains its own transformer-isolated supply and opto-isolated signal path: this can help avoid interaction between power lines and network cables destroying your microcontroller.

5. www.maxim-ic.com

6. www.national.com

7. www.linear-tech.com

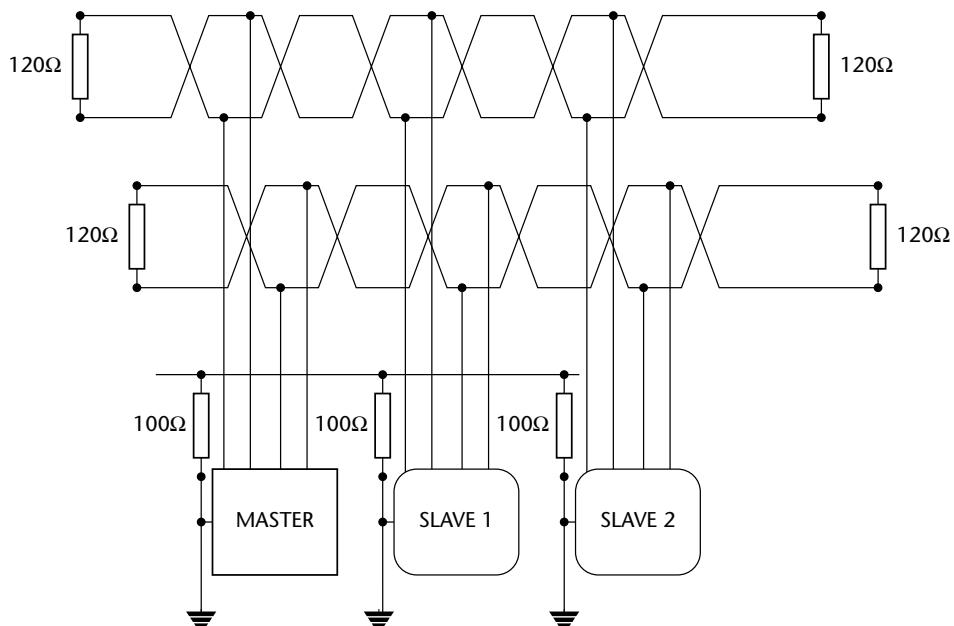


FIGURE 27.13 Possible (twisted-pair) cabling for an RS-485 network

[Note: that the RS-485 specification recommends connecting a 100W resistor (0.5 W or more) in series between each node's signal ground and the network's ground wire. This helps to ensure that if the ground potentials on the nodes vary, current flow in the ground wire is limited.]

Software considerations: enable inputs

The software required in this pattern is, in almost all respects, identical to that presented in **scu SCHEDULER (LOCAL)** [page 609]. The only exception is the need, in this multi-node system, to control the 'enable' inputs on the RS-485 transceivers; this is done because only one such device can be active on the network at any time.

The time-triggered nature of the shared-clock scheduler makes the controlled activation of the various transceivers straightforward: refer to the code examples that follow for details.

Hardware resource implications

Clearly, the main (microcontroller) hardware resource required is the UART; refer to the second example at the end of **scu SCHEDULER (LOCAL)** [page 609] for techniques that may allow you to use this pattern in circumstances where a UART is also required for data transfers to a PC.

Note that this pattern does not necessarily require the use of a timer for baud-rate generation; see the 'Solution' section in **scu SCHEDULER (LOCAL)** for details.

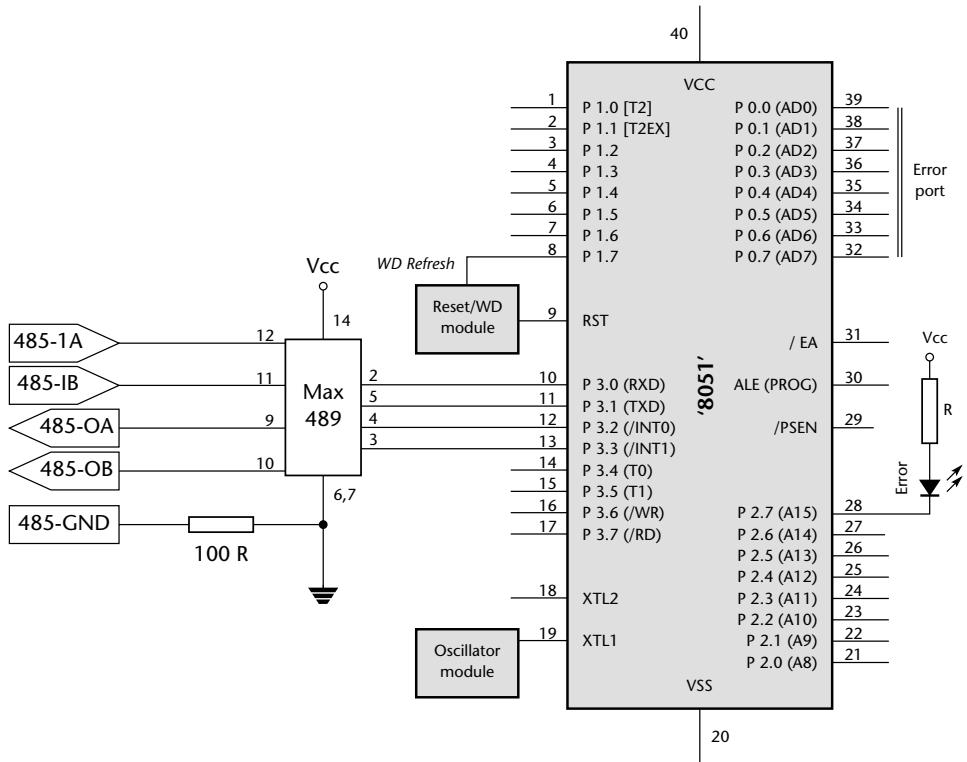


FIGURE 27.14 A simple node for an RS-485-based network

Reliability and safety implications

Refer to page 552 for a discussion of the reasons why use of multiple microcontrollers may decrease the system reliability.

However, if a distributed solution is required, the fact that RS-485 is a differential communication standard means that applications based on this pattern will tend to have a good level of noise immunity.

Portability

This pattern can be used with the whole 8051 family and many other microcontrollers / microprocessor / DSP devices.

Overall strengths and weaknesses

- ☺ A simple scheduler for distributed systems consisting of multiple 8051 microcontrollers.

- ☺ Easy to implement with low CPU and memory overheads.
- ☺ Twisted-pair cabling and differential signals make this more robust than RS-232-based alternatives.
- (☹) UART supports byte-based communications only: data transfer between Master and Slaves (and vice versa) is limited to 0.5 bytes per clock tick.
- (☹) Uses an important hardware resource (the UART).
- (☹) The hardware still has a very limited ability to detect errors: most error detection / correction must be carried out in software.

Related patterns and alternative solutions

See **scc SCHEDULER** [page 677] for an alternative scheduler suitable for use in multi-drop networks and with the ability to transfer larger amounts of data between network nodes.

Example: Network with Max489 transceivers

In this example, we present a code framework for an **scu SCHEDULER (RS-485)** suitable for use with network nodes like that illustrated in Figure 27.15 (Listings 27.3 and 27.4).

Master software

```
/* -----
   SCU_Bm.c (v1.00)

-----
This is an implementation of SCU Scheduler (RS-485) for 8051/52.

*** MASTER NODE ***
*** CHECKS FOR SLAVE ACKNOWLEDGEMENTS ***
*** Includes support for transceiver enables ***

--- Assumes 12.00 MHz crystal -> 5ms tick rate ---
--- Master and slave(s) share same tick rate ---

--- Assumes '1232' watchdog on Master and Slave ---

*/
#include "Main.h"
#include "Port.h"

#include "SCU_Bm.H"
#include "Delay_T0.h"
#include "TLight_B.h"
```

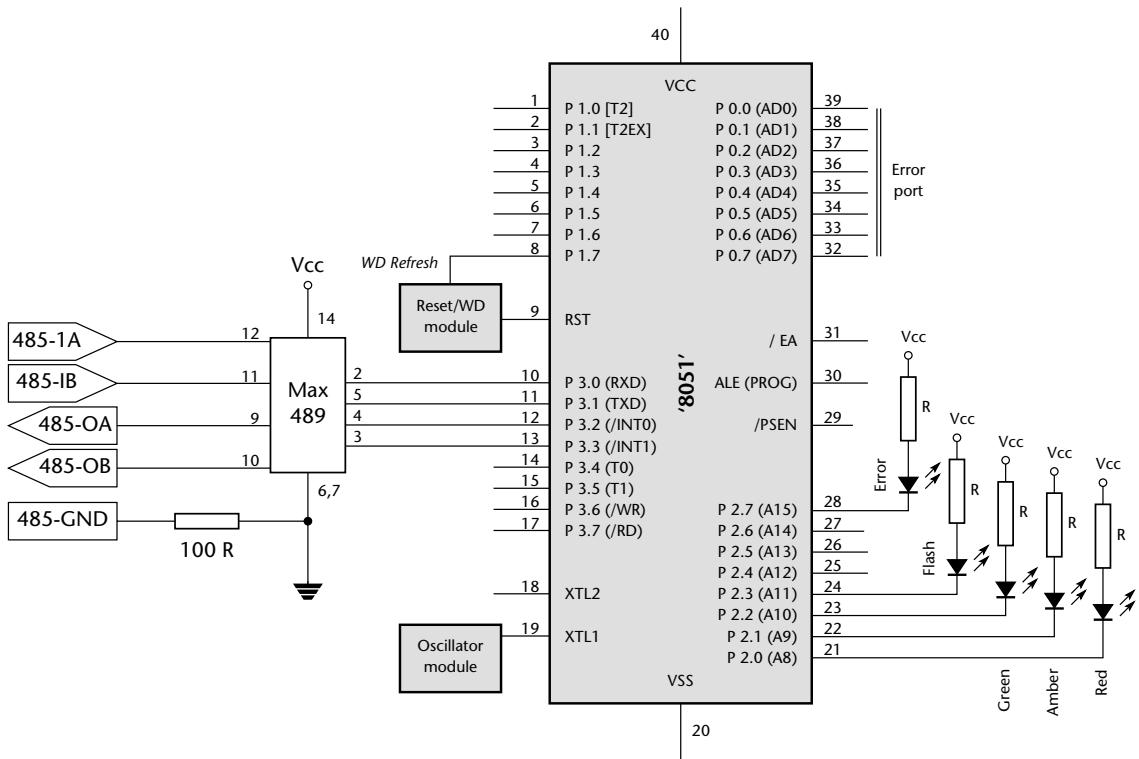


FIGURE 27.15 A node for an RS-485-based network

```

// ----- Public variable definitions -----
tByte Tick_message_data_G[NODE_OF_SLAVES] = {'M'};
tByte Ack_message_data_G[NODE_OF_SLAVES];

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// ----- Private variable definitions -----
static tByte Current_slave_index_G = 0;
static bit First_ack_G = 1;
static bit WATCHDOG_state_G = 0;

// ----- Private function prototypes -----
static viod 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;

// ----- Private constants -----
// Slave IDs may be any NON-ZERO tByte value (all must be different)
// NOTE: ID 0x00 is for error handling.
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)

// Adjust (if required) to increase interval between network resets
// (in the event of sustained network error)
#define SLAVE_RESET_INTERVAL 0U

// ----- Private variables -----
static tWord Slave_reset_attempts_G[NUMBER_OF_SLAVES];

// See MAIN_SLAVE_IDs[] above
static tByte Current_Slave_IDs_G[NUMBER_OF_SLAVES] = {0};

static bit Message_byte_G = 1;

/*-----*
SCU_B_MASTER_Init_T1_T2()

Scheduler initialisation function. Prepares scheduler data
structures and sets up timer interrupts at required rate.
You must call this function before using the scheduler.

BAUD_RATE - The required baud rate.
-*-----*/
void SCU_B_MASTER_Init_T1_T2(const tWord BAUD_RATE)
{
    tByte Task_index;
    tByte Slave_index;

    // No interrupts (yet)
    EA = 0;

    // Start the watchdog
    SCU_B_MASTER_Watchdog_Init();

    Network_error_pin = NO_NETWORK_ERROR;

    // Set up RS-485 transceiver
    RS485_Rx_NOT_Enable = 0; // Master Rec is constantly enabled
    RS485_Tx_Enable = 1; // Master Tran is constantly enabled
```

```

// ----- Set up the scheduler -----
// Sort out the tasks
for (Task_index = 0; Task_index < SCH_MAX_TASKS; Task_index++)
{
    SCH_Delete_Task(Task_index);
}

// Reset the global error variable
// - SCH_Delete_Task() will generate an error code,
//   (because the task array is empty)
Error_code_G = 0;

// We allow any combination of ID numbers in slaves
// - but reserve ID 0x00
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];
}

// ----- Set the baud rate (begin) -----
PCON &= 0x7F; // Set SMOD bit to 0 (don't double baud rates)

// Receiver enabled
// *9-bit data*, 1 start bit, 1 stop bit, variable baud rate
//(asynchronous)
SCON = 0xD2;

TMOD |= 0x20; // T1 in mode 2, 8-bit auto reload

TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
    / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));
TL1 = TH1;
TR1 = 1; // Run the timer
TI = 1; // Send first character (dummy)
// ----- Set the baud rate (end) -----

// Serial interrupt *NOT* enabled
ES = 0;
// ----- Set up the serial link (end) -----

// ----- Set up Timer 2 (begin) -----
// Crystal is assumed to be 12 MHz, 12 osc/increment
//
// The Timer 2 resolution is 0.000001 seconds (1 µs)
// The required Timer 2 overflow is 0.005 seconds (5 ms)
// - this takes 5000 timer ticks
// Reload value is 65536 - 5000 = 60536 (dec) = 0xEC78

```

```

T2CON = 0x04; // load Timer 2 control register
T2MOD = 0x00; // load Timer 2 mode register

TH2    = 0xEC; // load timer 2 high byte
RCAP2H = 0xEC; // load timer 2 reload capture reg, high byte
TL2    = 0x00; // load timer 2 low byte
RCAP2L = 0x00; // load timer 2 reload capture reg, low byte

ET2    = 1; // Timer 2 interrupt is enabled

TR2    = 1; // Start Timer 2
// ----- Set up Timer 2 (end) -----
}

/*-----*/
SCU_B_MASTER_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

NOTE: Delays here (2 ms) assume a baud rate of at least 9600 baud.
You will need to fine-tune this code if you opt for a lower baud
rate.
*/
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;

    // Refresh the watchdog
    SCU_B_MASTER_Watchdog_Refresh();

    // Place system in 'safe state'
    SCU_B_MASTER_Enter_Safe_State();

    // Report error as we wait to start
    Network_error_pin = NETWORK_ERROR;

    Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
    SCH_Report_Status(); // Sch not yet running - do this manually

    // Pause here (3000 ms), to time-out all the slaves
    // (This is the means by which we synchronize the network)
    for (i = 0; i < 100; i++)
}

```

```

{
    Hardware_Delay_T0(30);
    SCU_B_MASTER_Watchdog_Refresh();
}

// Currently disconnected from all slaves
Num_active_slaves = 0;

// After the initial (long) delay, all (operational) slaves will
// have timed out.
// All operational slaves will now be in the 'READY TO START' state
// Send them a (single-byte) message to get them started
// DO THIS TWICE IN A ROW, WITH COMMAND BIT
// This cannot happen under normal circumstances
Slave_index = 0;
do {
    // Refresh the watchdog
    SCU_B_MASTER_Watchdog_Refresh();

    // Clear 'first byte' flag
    First_byte = 0;

    // Find the slave ID for this slave
    Slave_ID = (tByte) Current_Slave_IDs_G[Slave_index];

    // Send a 'Slave ID' message
    TI = 0;
    TB8 = 1; // Set command bit
    SBUF = Slave_ID;

    // Wait to give slave time to reply
    Hardware_Delay_T0(5);

    // Check we had a reply
    if (RI == 1)
    {
        // Get the reply data
        Id = (tByte) SBUF;
        RI = 0;

        // Check reply - with command bit
        if ((Id == Slave_ID) && (RB8 == 1))
        {
            First_byte = 1;
        }
    }

    // Send second byte
    TI = 0;
}

```

```
TB8 = 1;
SBUF = Slave_ID;

// Wait to give slave time to reply
Hardware_Delay_T0(5);

// Check we had a reply
Slave_replied_correctly = 0;
if (RI != 0)
{
    //
    // Get the reply data
    Id = (tByte) SBUF;
    RI = 0;

    if ((Id == Slave_ID) && (RB8 == 1) && (First_byte == 1))
    {
        Slave_replied_correctly = 1;
    }
}

if (Slave_replied_correctly)
{
    //
    // Slave responded correctly
    Num_active_slaves++;
    Slave_index++;
}
else
{
    //
    // Slave did not reply correctly
    // - try to switch to backup device (if available)
    if (Current_Slave_IDS_G[Slave_index] != BACKUP_SLAVE_IDS[Slave_index])
    {
        //
        // There is a backup available: switch to backup and try
        // again
        Current_Slave_IDS_G[Slave_index] =
        BACKUP_SLAVE_IDS[Slave_index];
    }
    else
    {
        //
        // No backup available (or backup failed too) - have to
        // continue
        // NOTE: we don't abort the loop, to allow for more flexible
        // error handling (below)
        Slave_index++;
    }
}
```

```

        }
    } while (Slave_index < NUMBER_OF_SLAVES);

    // DEAL WITH CASE OF MISSING SLAVE(S) HERE ...
    if (Num_active_slaves < NUMBER_OF_SLAVES)
    {
        // User-defined error handling here...
        // 1 or more slaves have not replied
        // NOTE: In some circumstances you may wish to abort if slaves
        // are missing
        // - or reconfigure the network.

        // Here we display error and shut down the network
        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;
    }

    // Get ready to send first tick message
    Message_byte_G = 1
    First_ack_G = 1;
    Current_slave_index_G = 0;

    // Start the scheduler
    EA = 1;
}

/*-----*
SCU_B_MASTER_Update_T2

This is the scheduler ISR. It is called at a rate determined by
the timer settings in SCU_B_MASTER_Init_T2(). This version is
triggered by Timer 2 interrupts: timer is automatically reloaded.
*-----*/
void SCU_B_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Task_index;
    tByte Previous_slave_index;

    TF2 = 0; // Must manually clear this.

    // Refresh the watchdog
    SCU_B_MASTER_Watchdog_Refresh();
}

```

```
// Default
Network_error_pin = NO_NETWORK_ERROR;

// Keep track of the current slave
// FIRST VALUE IS 0
Previous_slave_index = Current_slave_index_G;

// Assume 2-byte messages sent and received
// - it takes two ticks to deliver each message
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;
}
}

// Check that the appropriate slave responded to the previous message:
// (if it did, store the data sent by this slave)
if (SCU_B_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
    // switch to a backup device (if one is available) as we reset
    // the network
    //
    // NOTE: We don't do this every tick (or the the network will be
    // constantly reset)
    //
    // Choose a value of SLAVE_RESET_INTERVAL to allow resets (say)
    // every 5 seconds
    if (++Slave_reset_attempts_G[Previous_slave_index] >=
SLAVE_RESET_INTERVAL)
    {
{
        // Now reset the network
        SCU_B_MASTER_Reset_the_Network();
}
}
}
else
```

```

{
// Reset this counter
Slave_reset_attempts_G[Previous_slave_index] = 0
}

// Send 'tick' message to all connected slaves
// (sends one data byte to the current slave)
SCU_B_MASTER_Send_Tick_Message(Current_slave_index_G);

// NOTE: calculations are in *TICKS* (not milliseconds)
for (Task_index = 0; Task_index < SCH_MAX_TASKS; Task_index++)
{
    // Check if there is a task at this location
    if (SCH_tasks_G[Task_index].pTask)
    {
        if (SCH_tasks_G[Task_index].Delay == 0)
        {
            // The task is due to run
            SCH_tasks_G[Task_index].RunMe += 1; // Increment the run
                                            // flag

            if (SCH_tasks_G[Task_index].Period)
            {
                // Schedule periodic tasks to run again
                SCH_tasks_G[Task_index].Delay =
                SCH_tasks_G[Task_index].Period;
            }
        }
        else
        {
            // Not yet ready to run: just decrement the delay
            SCH_tasks_G[Task_index].Delay -= 1;
        }
    }
}
}

/*-----*
 * SCU_B_MASTER_Send_Tick_Message()

This function sends a tick message, over the USART network.
The receipt of this message will cause an interrupt to be generated
in the slave(s): this invokes the scheduler 'update' function
in the slave(s).
*-----*/

```

```

void SCU_B_MASTER_Send_Tick_Message(const tByte SLAVE_INDEX)
{
    tLong Timeout;

    // Find the slave ID for this slave
    tByte Slave_ID = (tByte) Current_Slave_IDS_G[SLAVE_INDEX];

    // Sending one byte of data at a time, depending on index value
    // If Message_byte_G is 0, send first byte (the slave ID)
    if (Message_byte_G == 0)
    {
        Timeout = 0;
        while ((++Timeout) && (TI == 0));

        if (Timeout == 0)
        {
            // UART did not respond - error
            Error_code_G = ERROR_USART_TI;
            return;
        }

        TI = 0;
        TB8 = 1; // Set 'Command' bit
        SBUF = Slave_ID;
    }
    else
    {
        // Message_byte_G is 1, send the data byte
        Timeout = 0;
        while ((++Timeout) && (TI == 0));

        if (Timeout == 0)
        {
            // usart did not respond - error
            Error_code_G = ERROR_USART_TI;
            return;
        }

        TI = 0;
        TB8 = 0;
        SBUF = Tick_message_data_G[SLAVE_INDEX];
    }

    // Data sent - return
}

```

/*-----*.-

SCU_B_MASTER_Process_Ack()

Make sure the slave (SLAVE_ID) has acknowledged the previous message that was sent. If it has, extract the message data from the USART hardware: if not, call the appropriate error handler.

Slave_index - The index of the slave.

RETURNS: RETURN_NORMAL - Ack received (data in Ack_message_data_G)
RETURN_ERROR - No ack received (-> no data)

```
-----*/
bit SCU_B_MASTER_Process_Ack(const tByte Slave_index)
{
    tByte Message_contents;
    tByte Slave_ID;

    // First time this is called there is no ack tick to check
    // - we simply return 'OK'
    if (First_ack_G)
    {
        First_ack_G = 0;
        return RETURN_NORMAL;
    }

    // Find the slave ID for this slave
    Slave_ID = (tByte) Current_Slave_IDS_G[Slave_index];

    // Data should already be in the buffer
    if (RI == 0)
    {
        // Slave has not replied to last tick message

        // Set error LED
        Network_error_pin = NETWORK_ERROR;

        return RETURN_ERROR;
    }

    // There is data - get it
    Message_contents = (tByte) SBUF;
    RI = 0;

    // This is the reply to the last message
    // - reverse the message byte interpretation
    if (Message_byte_G == 1)
    {
        // Check the 'command bit' is set
        if (RB8 == 1)
        {
```

```

        // Check that the ID is correct
        if (Slave_ID == (tByte) Message_contents)
        {
            // Required Ack message was received
            return RETURN_NORMAL;
        }
    }

    // Something is wrong...

    // Set error LED
    Network_error_pin = NETWORK_ERROR;

    return RETURN_ERROR;
}
else
{
    // There *ARE* data available
    // Extract the data from the slave message
    //

    // NOTE: We *assume* these data are OK
    // - you may wish to send crucial data twice, etc.
    Ack_message_data_G[Slave_index] = Message_contents;

    return RETURN_NORMAL; // return the slave data
}
}

/*-----*
SCU_B_MASTER_Reset_the_Network()

This function resets (that is, restarts) the whole network.

-*-----*/
void SCU_B_MASTER_Reset_the_Network(void)
{
    EA = 0;
    while(1); // Watchdog will time out
}

/*-----*
SCU_B_MASTER_Shut_Down_the_Network()

This function shuts down the whole network.

-*-----*/
void SCU_B_MASTER_Shut_Down_the_Network(void)
{
    EA = 0; // Disable interrupts
}

```

```

Network_error_pin = NETWORK_ERROR;
SCH_Report_Status(); // Sch not running - do this manually

while(1)
{
    SCU_B_MASTER_Watchdog_Refresh();
}
}

/*-----*
SCU_B_MASTER_Enter_Safe_State()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Master node cannot detect a slave
(3) The network has an error

Try to ensure that the system is in a 'safe' state in these
circumstances.

*-----*/
void SCU_B_MASTER_Enter_Safe_State(void)
{
// USER DEFINED - Edit as required
TRAFFIC_LIGHTS_Display_Safe_Output();
}

/*-----*
SCU_B_MASTER_Watchdog_Init()

This function sets up the watchdog timer.

If the Master fails (or other error develops),
no tick messages will arrive, and the scheduler
will stop.

To detect this situation, we have a (hardware) watchdog
running in the slave. This watchdog - which should be set to
overflow at around 100 ms - is used to set the system into a
known (safe) state. The slave will then wait (indefinitely)
for the problem to be resolved.

NOTE: The slave will not be generating Ack messages in these
circumstances. The Master (if running) will therefore be aware
that there is a problem.

*-----*/
void SCU_B_MASTER_Watchdog_Init(void)
{

```

```

    // INIT NOT REQUIRED FOR 1232 EXTERNAL WATCHDOG
    // - May be required with different watchdog hardware
    // Edit as required
}

/*-----*
SCU_B_MASTER_Watchdog_Refresh()

Feed the external (1232) watchdog.

Timeout is between ~60 and 250 ms (hardware dependent)

Assumes external 1232 watchdog
-*-----*/
void SCU_B_MASTER_Watchdog_Refresh(void) reentrant
{
    // Change the state of the watchdog pin
    if (WATCHDOG_state == 1)
    {
        WATCHDOG_state_G = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state_G = 1;
        WATCHDOG_pin = 1;
    }
}

/*-----*
----- END OF FILE -----
-*-----*/

```

Listing 27.3 Part of the software for shared-clock (UART) scheduler demonstration system (Master node)

[Note: Network supports Max489 transceivers and uses the RS-485 protocol.]

Slave software

```

/*-----*
SCU_Bs.c (v1.00)

-----*
This is an implementation of SCU SCHEDULER (RS-485) for 8051/52.

```

```
*** 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"

// ----- Public variable definitions -----
// Data sent from the master to this slave
tByte Tick_message_data_G;

// Data sent from this slave to the master
// - data may be sent on, by the master, to another slave
tByte Ack_message_data_G = 'S';

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// ----- Private function prototypes -----
static void SCU_B_SLAVE_Enter_Safe_State(void);

static void SCU_B_SLAVE_Send_Ack_Message_To_Master(void);
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;

// ----- Private constants -----
// Each slave must have a unique ID
#define SLAVE_ID 0x31

#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)
```

```

// ----- Private variables -----
static bit Message_byte_G;
static bit WATCHDOG_state_G = 0
static tByte Message_ID_G = 0

/*-----*
SCU_B_SLAVE_Init_T1()

Scheduler initialisation function. Prepares scheduler
data structures and sets up timer interrupts at required rate.
Must call this function before using the scheduler.

BAUD_RATE - The required baud rate
-*-----*/
void SCU_B_SLAVE_Init_T1(const tWord BAUD_RATE)
{
    tByte i;

    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // Set the network error pin (reset when tick message received)
    Network_error_pin = NETWORK_ERROR;

    // Set up RS-485 transceiver
    RS485_Rx_NOT_Enable = 0; // Receiver is (here) constantly enabled
                            // (NOTE - negative logic!)

    RS485_Tx_Enable = 0;      // Transmitter (in slave) is enabled
                            // only when data are to be transmitted
                            // by this slave

    // Ready for first tick message
    Message_byte_G = 1;

    // ----- Set the baud rate (begin) -----
    PCON &= 0x7F; // Set SMOD bit to 0 (don't double baud rates)

    // receiver enabled
    // 9-bit data, 1 start bit, 1 stop bit, variable baud rate
    // (asynchronous)
    SCON = 0xD2;

```

```

TMOD |= 0x20; // T1 in mode 2, 8-bit auto reload

TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
    / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));

TL1 = TH1;
TR1 = 1; // Run the timer
TI = 1; // Send first character (dummy)

// ----- Set the baud rate (end) -----

// Interrupt enabled
// (Both receiving and SENDING a byte will generate a serial interrupt)
// Global interrupts not yet enabled.
ES = 1;

// Start the watchdog
SCU_B_SLAVE_Watchdog_Init();
}

/*-----*
SCU_B_SLAVE_Start()

Starts the slave scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

*-----*/
void SCU_B_SLAVE_Start(void)
{
    tByte Command = 0;
    tByte Message_byte;
    tByte Count = 0;
    bit Slave_started = 0;

    // Disable interrupts
    EA = 0;

    // We can be at this point because:
    // 1. The network has just been powered up
    // 2. An error has occurred in the Master, and it is not generating
    //     ticks
    // 3. The network has been damaged and no ticks are being received
    //     by this slave
    //

    // Try to make sure the system is in a safe state...
    SCU_B_SLAVE_Enter_Safe_State();
}

```

```
// NOTE: Interrupts are disabled here
Count = 0;

Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
SCH_Report_Status(); // Sch not yet running - do this manually

// Now wait (indefinitely) for appropriate signals from the master
do {
    // Wait for tick messages (byte 1), all bits set to 0, to be
    // received
    do {
        SCU_B_SLAVE_Watchdog_Refresh(); // Must keep feeding the
                                         // watchdog
        } while (RI == 0);

    Message_byte = (tByte) SBUF;
    RI = 0;

    // Must get two ID messages in a row...
    // (with command bit)
    // Ack each one
    if ((Message_byte == (tByte) SLAVE_ID) && (RB8 == 1))
    {
        Count++;

        // Received message for this slave - send ack
        // Must enable the slave RS-485 (Max489) hardware (Tx)
        RS485_Tx_Enable = 1;

        TI = 0;
        TB8 = 1; // Set command bit
        SBUF = (tByte) SLAVE_ID;

        // Wait while data are sent
        // (watchdog will trap UART failure...)
        while (TI == 0)

            // Now clear Tx enable pin
            RS485_Tx_Enable = 0;
        }

    else
    {
        Count = 0;
    }
} while (Count < 2);
// Start the scheduler
EA = 1;
}
```

```

/* -----
   SCU_B_SLAVE_Update

   This is the scheduler ISR. It is called at a rate
   determined by the timer settings in SCU_B_SLAVE_Init().

   This Slave is triggered by USART interrupts.

----- */

void SCU_B_SLAVE_Update(void) interrupt INTERRUPT_UART_Rx_Tx
{
    tByte Index;

    if (RI == 1) // Must check this.
    {
        // Default
        Network_error_pin = NO_NETWORK_ERROR;

        // Two-byte messages are sent (Ack) and received (Tick)
        // - it takes two sched ticks to process each message
        //
        // Keep track of the current byte
        if (Message_byte_G == 0)
        {
            Message_byte_G = 1;
        }
        else
        {
            Message_byte_G = 0;
        }

        // Check tick data - send ack if necessary
        // NOTE: 'START' message will only be sent after a 'time out'
        if (SCU_B_SLAVE_Process_Tick_Message() == SLAVE_ID)
        {
            SCU_B_SLAVE_Send_Ack_Message_To_Master();

            // Feed the watchdog ONLY when a *relevant* message is
            // received
            // (noise on the bus, etc, will not stop the watchdog...)
            //
            // START messages will NOT refresh the slave
            // - Must talk to every slave at regular intervals
            SCU_B_SLAVE_Watchdog_Refresh();
        }

        // NOTE: calculations are in *TICKS* (not milliseconds)
        for (Index = 0; Index < SCH_MAX_TASKS; Index++)
        {

```

```

        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                SCH_tasks_G[Index].RunMe = 1; // Set the run flag

                if (SCH_tasks_G[Index].Period)
                {
                    // Schedule periodic tasks to run again
                    SCH_tasks_G[Index].Delay =
                    SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }

    RI = 0; // Reset the RI flag
}

else
{
    // ISR call was triggered by TI flag, after last character was sent

    // RS485_Tx_Enable flag is reset here
    RS485_Tx_Enable = 0;

    // Must clear the TI flag
    TI = 0;
}
}

/*-----*/

```

SCU_B_SLAVE_Send_Ack_Message_To_Master()

Slave must send and 'Acknowledge' message to the master, after tick messages are received. NOTE: Only tick messages specifically addressed to this slave should be acknowledged.

The acknowledge message serves two purposes:

- [1] It confirms to the master that this slave is alive & well.
- [2] It provides a means of sending data to the master and - hence - to other slaves.

NOTE: Direct data transfer between slaves is NOT possible.

```
-----*/
void SCU_B_SLAVE_Send_Ack_Message_To_Master(void)
{
    // Enable the slave RS-485 hardware (Tx)
    // NOTE: This flag will be reset in the 'Update' ISR
    RS485_Tx_Enable = 1;

    // Sending one byte of data at a time, depending on index value
    // If Message_byte_G is 0, send first byte (the slave ID)
    if (Message_byte_G == 0)
    {
        TI = 0;
        TB8 = 1; // Set 'Command' bit
        SBUF = SLAVE_ID;
    }
    else
    {
        // Message_byte_G is 1, send the data byte
        TI = 0;
        TB8 = 0;
        SBUF = Ack_message_data_G;
    }

    // Data sent - return
}
```

```
-----*/
```

`SCU_B_SLAVE_Process_Tick_Message()`

The tick messages are crucial to the operation of this shared-clock scheduler: the arrival of a tick message (at regular intervals) invokes the 'Update' ISR, that drives the scheduler.

The tick messages themselves may contain data. These data are extracted in this function.

```
-----*/
tByte SCU_B_SLAVE_Process_Tick_Message(void)
{
    tByte Data;

    // Try to get data byte
    if (RI == 0)
    {
        // No data - something is wrong
```

```

    // Set the error flag bit
    Network_error_pin = NETWORK_ERROR;

    // Return slave ID 0
    return 0x00;
}

// There *ARE* data available
Data = (tByte) SBUF;
RI = 0; // Clear RI flag

// What we do with this message depends if it a first or second byte
if (Message_byte_G == 0)
{
    //
    // This is (should be) an ID byte
    Message_ID_G = Data;

    if (RB8 == 0)
    {
        Message_ID_G = 0; // Command bit should be set
    }
}
else
{
    //
    // This is (should be) a data byte
    // - Command bit should not be set
    if ((Message_ID_G == SLAVE_ID) && (RB8 == 0))
    {
        Tick_message_data_G = Data;
    }
}
else
{
    //
    // Something is wrong - set Message_ID to 0
    Message_ID = 0;

    // Set the error flag bit
    Network_error_pin = NETWORK_ERROR;
}
}

return Message_ID_G;
}
/*-----*-SCU_B_SLAVE_Watchdog_Init()-----*/
SCU_B_SLAVE_Watchdog_Init()

This function sets up the watchdog timer.

```

If the Master fails (or other error develops), no tick messages will arrive, and the scheduler will stop.

To detect this situation, we have a (hardware) watchdog running in the slave. This watchdog - which should be set to overflow at around 100 ms - is used to set the system into a known (safe) state. The slave will then wait (indefinitely) for the problem to be resolved.

NOTE: The slave will not be generating Ack messages in these circumstances. The Master (if running) will therefore be aware that there is a problem.

```
-----*/
void SCU_B_SLAVE_Watchdog_Init(void)
{
    // INIT NOT REQUIRED FOR 1232 EXTERNAL WATCHDOG
    // - May be required with different watchdog hardware
    //
    // Edit as required
}

/*
SCU_B_SLAVE_Watchdog_Refresh()
Feed the external (1232) watchdog.
Timeout is between ~60 and 250 ms (hardware dependent)
Assumes external 1232 watchdog
-----*/
void SCU_B_SLAVE_Watchdog_Refresh(void) reentrant
{
    // Change the state of the watchdog pin
    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()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Master node fails, and no working backup is available
(3) The network has an error
(4) Tick messages are delayed for any other reason

Try to ensure that the system is in a 'safe' state in these
circumstances.

----- */
void SCU_B_SLAVE_Enter_Safe_State(void)
{
    // USER DEFINED - Edit as required
    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/* -----
----- END OF FILE -----
----- */

```

Listing 27.4 Part of the software for shared-clock (UART) scheduler demonstration system (Slave node)

[Note: Network supports Max489 transceivers, and uses the RS-485 protocol.]

Further reading

- 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]

chapter **28**

Shared-clock schedulers using CAN

Introduction

In the previous chapter we introduced methods, based on 'RS-232' and 'RS-485' communication protocols, to allow us to extend the co-operative scheduler discussed in Part C for use in a distributed, multiprocessor environment.

One key advantage of using such environments is that the essential (UART) hardware is available in all members of the 8051 family, as well as the great majority of other microcontroller devices (8-bit, 16-bit, 32-bit or greater). As a result, creating point-to-point and multiprocessor distributed (and local) systems in this way is a cost-effective solution that may be widely applied.

However, UART-based communication protocols suffer from two significant weaknesses in a shared-clock environment:

- (⌚) The UART supports byte-based communications only: data transfer between Master and Slaves (and vice versa) is limited to 0.5 bytes per clock tick.
- (⌚) The hardware has a very limited ability to detect errors: most error detection / correction must be carried out in software.

In the late 1970s the automotive industry began to tackle very similar problems. The motivation was largely that the wiring looms in passenger cars were becoming larger, heavier and more expensive. The industry was looking for a form of cheap, multi-drop serial bus to which numerous different system components could be connected (see Lawrenz, 1997, for further details).

From this need arose the 'controller area network' (CAN). CAN is now becoming commonplace in vehicles produced by most major automotive manufacturers. It is also widely used in process control and other industrial areas.

We can summarize some of the features of CAN as follows:

- ☺ CAN is message based and messages can be up to eight bytes in length. Used in a shared-clock scheduler, the data transfer between Master and Slaves (and vice versa) is up to seven bytes per clock tick. This is adequate for most applications.
- ☺ A number of 8051 devices have on-chip support for CAN, allowing the protocol to be used with minimal overheads.
- ☺ The hardware has advanced error-detection (and correction) facilities built in, further reducing the software load.
- ☺ CAN may be used for both 'local' and 'distributed' systems.

In this chapter, we will consider how we can use the CAN bus as an efficient means of creating reliable, shared-clock, multiprocessor systems.

SCC SCHEDULER

Context

- You are developing an embedded application using more than one member of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a scheduler.

Problem

How do you schedule tasks on (and transfer data over) a network of two (or more) 8051 microcontrollers communicating using the CAN protocol?

Background

We consider some relevant background material in this section

What is a shared-clock scheduler?

See Chapter 25 for general background material on shared-clock schedulers.

What is CAN?

We begin our discussion of the controller area network (CAN) protocol by highlighting some important features of this standard:

- CAN supports high-speed (1 Mbits/s) data transmission over short distances (40 m) and low-speed (5 kbytes/s) transmissions at lengths of up to 10,000 m.
- CAN is message based. The data in each message may vary in length between 0 and eight bytes. This data length is ideal for many embedded applications.
- The receipt of a message can be used to generate an interrupt. The interrupt will be generated only when a complete message (up to eight bytes of data) has been received: this is unlike a UART (for example) which will respond to every character.
- CAN is a shared broadcast bus: all messages are sent to all nodes. However, each message has an identifier: this can be used to ‘filter’ messages. This means that – by using a ‘Full CAN’ controller (see later) – we can ensure that a particular node will only respond to ‘relevant’ messages: that is, messages with a particular ID.

This is very powerful. What it means in practice is, for example, that a Slave node can be set to ignore all messages directed from a different Slave to the Master.

- CAN is usually implemented on a simple, low-cost, two-wire differential serial bus system. Other physical media may be used, such as fibre optics (but this is comparatively rare).

- The maximum number of nodes on a CAN bus is 32.
- Messages can be given an individual priority. This means, for example, that 'Tick messages' can be given a higher priority than 'Acknowledge messages'.
- CAN is highly fault tolerant, with powerful error-detection and handling mechanisms built into the controller.
- Last, but not least, microcontrollers with built-in CAN controllers are available from a range of companies. For example, 8051 devices with CAN controllers are available from Infineon (c505c, c515c), Philips (8xC592, 8xC598) and Dallas (80C390).

Overall, as we will see in the course of this chapter, the CAN bus provides an excellent foundation for reliable distributed scheduled applications.

We now consider some of the important features of CAN in more detail.

CAN 1.0 vs. CAN 2.0

The CAN protocol comes in two versions: CAN 1.0 and CAN 2.0. CAN 2.0 is backwardly compatible with CAN 1.0 and most new controllers are CAN 2.0.

In addition, there are two parts to the CAN 2.0 standard: Part A and Part B. With CAN 1.0 and CAN 2.0A, identifiers must be 11 bits long. With CAN 2.0B identifiers can be 11 bits (a 'standard' identifier) or 29 bits (an 'extended' identifier).

The following basic compatibility rules apply:

- CAN 2.0B *active* controllers are able to send and receive both standard and extended messages.
- CAN 2.0B *passive* controllers are able to send and receive standard messages. In addition, they will discard (and ignore) extended frames. They will not generate an error when they 'see' extended messages.
- CAN 1.0 controllers generate bus errors when they see extended frames: they cannot be used on networks where extended identifiers are used.

Basic CAN vs. Full CAN

There are two main classes of CAN controller available. Note that these classes are not covered by the standard, so there is some variation. As noted already, the difference is that Full CAN controllers provide an acceptance filter that allows a node to ignore irrelevant messages.

Which microcontrollers have support for CAN?

At the time of writing, the following 8051 devices provide on-chip CAN support:

- Dallas 80c390. Two on-chip CAN modules, each supporting CAN 2.0B.
- Infineon C505C. Supports CAN2.0B.
- Infineon C515C. Supports CAN2.0B.

- Philips 8xC591. Supports CAN2.0B.
- Philips 8x592. Supports CAN2.0A.
- Philips 8x598. Supports CAN2.0A.
- Temic T89C51CC01. Supports CAN2.0B.

What about CAN without on-chip hardware support?

Using a microcontroller with on-chip CAN support is, in most circumstances, likely to be easier, more reliable and more cost-effective. If you are unable to use a microcontroller with such support and need to use CAN, then you will require an external CAN controller chip. Here we will briefly mention two options.

Scott (1995) discusses in detail how to connect an Intel 82527 CAN controller chip to an 8051-family microcontroller.

Hank and Jöhnk (1997) discuss in detail how to connect a Philips SJA1000 CAN controller to various microcontrollers.

Solution

We consider how a CAN-based, shared-clock scheduler can be designed in this section.

The basis of the approach

All the shared-clock schedulers we discuss in this book have the same underlying architecture (Figure 28.1).

Here we have one, accurate clock on the Master node in the network. This clock is used to drive the scheduler in the Master node in exactly the manner discussed in Part C.

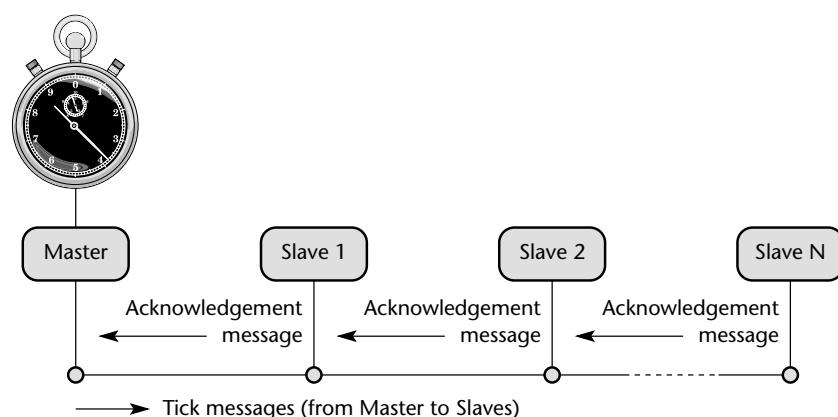


FIGURE 28.1 The architecture of all the shared-clock schedulers discussed in this book

The Slave nodes also have schedulers: however, the interrupts used to drive these schedulers are derived from ‘tick messages’ generated by the Master. Thus, in the CAN-based network we are concerned with in this pattern, the Slave node(s) will have a S-C scheduler driven by the ‘receive’ interrupts generated through the receipt of a byte of data sent by the Master.

Note that the ability to generate an interrupt (in a Slave) on receipt of a data byte by the CAN module makes the underlying scheduler operation an extremely simple two-stage process:

- 1 Timer overflow in the Master causes the scheduler ‘Update’ function to be invoked. This, in turn, causes a byte of data to be sent (via the CAN bus) to all Slaves:

```
void MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    ...
    ...
    MASTER_Send_Tick_Message(...);
    ...
}
```

- 2 When these data have been received all Slaves generate an interrupt; this invokes the ‘Update’ function in the Slave schedulers. This, in turn, causes one Slave to send an ‘Acknowledge’ message back to the Master (again via the CAN bus).

```
void SLAVE_Update(void) interrupt INTERRUPT_CAN
{
    ...
    ...
    SLAVE_Send_Ack_Message_To_Master();
    ...
}
```

The message structure

As we saw in **SCU SCHEDULER (LOCAL)** [page 609] the design of the message structure to be used in a UART-based network requires some care, since the data bandwidth is very limited (eight bits per tick). A key advantage of the CAN bus is that it supports messages with a size of up to eight-bytes. This makes the design of the message structure much more straightforward.

Here we adopt the following solution:

- Up to 31 Slave nodes (and one Master node) may be used in a CAN network. Each Slave is given a unique ID (0x01 to 0xFF).

- Each tick message from the Master is between one and eight bytes long; *all the bytes are sent in a single tick interval*. In all messages, the first byte is the ID of the Slave to which the message is addressed; the remaining bytes (if any) are the message data.
- All Slaves generate interrupts in response to *every* tick message (whether or not they are the addressee of the message).
- Only the Slave to which a tick message is addressed will reply to the Master; this reply takes the form of an acknowledge message.
- Each acknowledge message from a Slave is between one and eight bytes long; all of the bytes are sent in the same tick interval in which the tick message is received. The first byte of the acknowledge message is the ID of the Slave from which the message was sent; the remaining bytes (if any) are the message data (Figure 28.2).

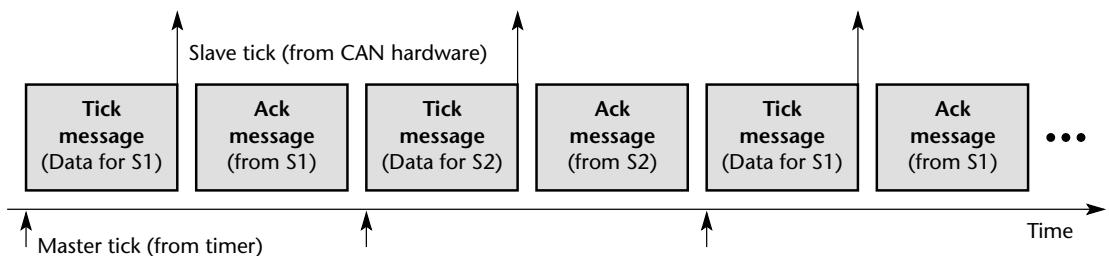


FIGURE 28.2 The communication between the Master and (two) Slave nodes in a CAN-based S-C network

Determining the required baud rate

The timing of timer ticks in the Master is set to a duration such that a tick message can be sent and an acknowledge message received between ticks. Clearly, this duration depends on the network baud rate.

Determining the required baud rate for a CAN bus is more involved, because the message structure is more complex (Table 28.1).

We require two messages per tick: with 1 ms ticks, we require at least 308,000 baud: allowing 350,000 baud gives a good margin for error. This is achievable with CAN, at distances up to around 100 metres. Should you require larger distances, the tick interval must either be lengthened or repeater nodes should be added in the network at 100-metre intervals.

Note also that, as with any shared-clock network, there is a delay between the timer on the Master and the UART-based interrupt on the Slave (Figure 28.3). This delay arises because the Slave interrupt is generated at the end of the tick message. In the absence of network errors, this delay is fixed and derives largely from the time taken to transmit a message via the CAN bus; that is, it varies with the baud rate. As discussed earlier, a baud rate of at least 350,000 bits/second is appropriate for a 1 ms scheduler; this gives a tick latency (assuming 154 bits / message: see Table 28.1) of approximately 0.5 ms.

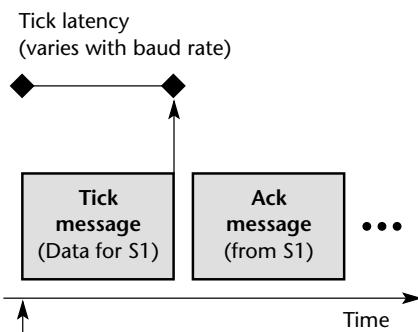


FIGURE 28.3 The latency between tick generation on the Master and on the Slave

TABLE 28.1 The structure of a CAN message

Description	Size (bits)
Data	64
Start bit	1
Identifier bits	11
SRR bit	1
IDE bit	1
Identifier bits	18
RTR bit	1
Control bits	6
CRC bits	15
Stuff bits (maximum)	23
CRC delimiter	1
ACK slot	1
ACK delimiter	1
EOF bits	7
IFS bits	3
Total	154 bits / message

[Note: this is a worst-case message duration and includes 8 bytes of data.]

As before, note that this latency is fixed and can be accurately predicted on paper and then confirmed through simulation and testing. If precise synchronization of Master and Slave processing is required, note that:

- All the Slaves are – within the limits of measurement – precisely in step.
- To bring the Master in step with the Slaves, it is necessary only to add a short delay in the Master ‘Update’ function.

Transceivers for distributed networks

Although several CAN transceivers are available, the Philips⁸ PCA82c250 remains the most widely used. This may be easily connected to any of the CAN-based 8051 devices, using a circuit similar to that shown in Figure 28.4.

Note that a range of different connections are possible; for example greater isolation of the microcontroller from damage to the CAN network can be achieved through the use of opto-couplers between the transceiver and the microcontroller: refer to the Philips PCA82c250 data sheet for further details.

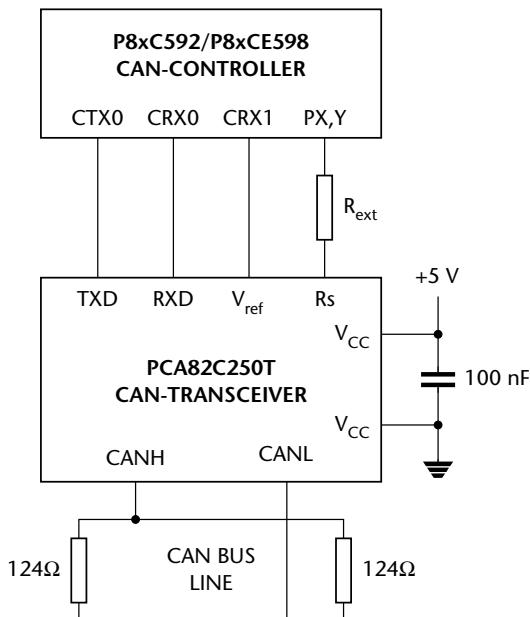


FIGURE 28.4 Connecting a CAN transceiver (Philips PCA82C250) to a CAN-based microcontroller (reproduced, courtesy of Philips Semiconductors)

Node wiring for distributed networks

The CAN standard does not dictate that a particular wiring scheme is used. However, the most common means of linking together CAN nodes is through the use of a two-wire, twisted pair arrangement: this is very similar to a single channel of the RS-485 wiring arrangement we discussed in **SCU SCHEDULER (RS-485)** [page 648].

In the CAN bus, the two signal lines are termed ‘CAN High’ and ‘CAN Low’. In the quiescent state, both lines sit at 2.5V. A ‘1’ is transmitted by raising the voltage of the High line above that of Low line: this is termed a ‘dominant’ bit. A ‘0’ is represented by raising the voltage of the Low line above that of the High line: this is termed a ‘recessive’ bit.

Using twisted-pair wiring, the differential CAN inputs successfully cancel out noise. In addition, the CAN networks connected in this way continue to function even when one of the lines is severed.

Note that, as with the RS-485 cabling, a 120Ω terminating resistor is connected at each end of the bus (Figure 28.5).

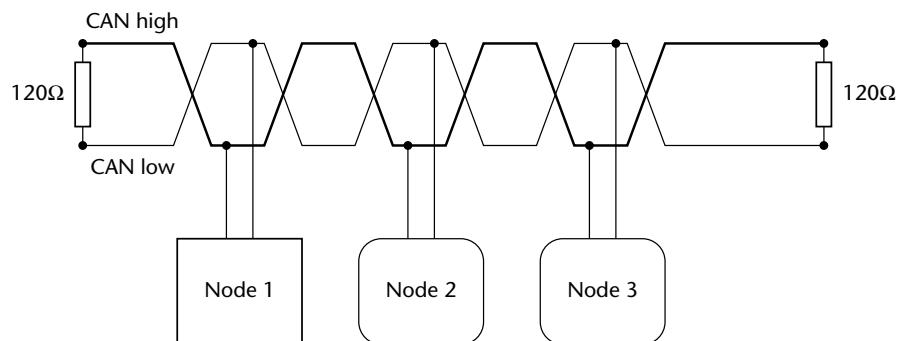


FIGURE 28.5 Connecting three nodes (with transceivers) to a twisted-pair CAN bus

Hardware and wiring for local networks

Use of a ‘local’ CAN network does not require the use of transceiver chips. For early prototypes, connecting together the Tx and Rx lines from a number of CAN-based microcontrollers will allow you to link the devices (that is, connect all the Tx lines together, and – separately – collect all the Rx lines together). No terminating resistors will be required.

A better solution (proposed by Barrensheen, 1996) is based on a wired-OR structure (see Figure 28.6). Here all Tx lines are connected to a single data line via fast diodes (in order to avoid short circuits at the output pins). The Rx inputs are directly connected to this single data line, which is pulled to +5V by a pull-up resistor (to generate the required passive ‘1’-level). The maximum wire length is limited to approximately one metre.

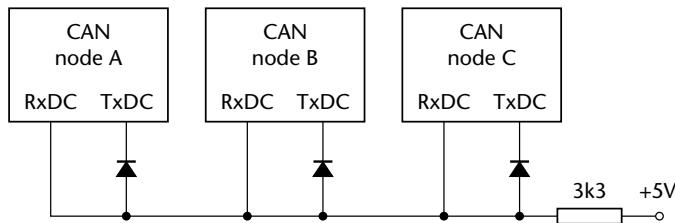


FIGURE 28.6 Connecting together CAN-based microcontrollers without the use of transceivers
(adapted from Barrensheen, 1996)

Software for the shared-clock CAN scheduler

As you would expect, the software required to create the shared-clock CAN scheduler is very similar in structure to the code presented in the previous two chapters. (Refer to the code listings that follow for detailed implementation details.)

One important difference between the scheduler presented here and those discussed in previous chapters is the mechanism of error handling used here. As we discussed in Chapter 25 (page 549), three different error-handling techniques for S-C schedulers are presented in this book. Here we use technique three. This operates as follows. If a Slave fails, then – rather than restarting the whole network – we attempt to start the corresponding backup unit.

The strengths and weaknesses of this approach are as follows:

- 😊 It allows full use to be made of backup nodes.
- 😊 In most circumstances it takes comparatively little time to engage the backup unit.
- 😢 The underlying coding is more complicated than the other alternatives discussed in this book.

Note that if the network fails to start the backup unit successfully, one of the other recovery strategies introduced in Chapter 25 may be attempted.

Hardware resource implications

Use of CAN will be most cost-effective using a controller with built-in CAN support.

Note that some controllers are available with dual CAN connections.

Reliability and safety implications

Refer to page 550 for a discussion of the reasons why use of multiple microcontrollers may decrease the system reliability.

Nonetheless, CAN is highly fault tolerant, with powerful error-detection and handling mechanisms built into the controller: some of these facilities are demonstrated in the sample code that follow.

Portability

As relatively few 8051 microcontrollers have built-in CAN support, this pattern is less easy to port than some of the other patterns presented in this book.

Overall strengths and weaknesses

- ☺ CAN is message based and messages can be up to eight bytes in length. Used in a shared-clock scheduler, the data transfer between Master and Slaves (and vice versa) is up to seven bytes per clock tick. This is more than adequate for the great majority of applications.
- ☺ A number of 8051 devices have on-chip support for CAN, allowing the protocol to be used with minimal overheads.
- ☺ The hardware has advanced error-detection (and correction) facilities built in, further reducing the software load
- ☺ CAN may be used for both 'local' and 'distributed' systems.
- ☹ 8051 devices with CAN support tend to be more expensive than 'Standard' 8051s.

Related patterns and alternative solutions

The nearest alternative discussed in this book is **SCU SCHEDULER (RS-485)** [page 646].

Example: Creating a CAN-based scheduler using the Infineon C515c

This example illustrates the use of the Infineon c515C microcontroller. This popular device has on-chip CAN hardware (Listings 28.1 and 28.2).

The code may be used in either a distributed or local network, with the hardware discussed earlier.

Master software

```
/* ----- *-
SCC_M515.c (v1.00)

-----
*** THIS IS A SHARED-CLOCK (CAN) SCHEDULER (MASTER) ***
*** FOR 80C515C (etc.) ***

*** Uses T2 for timing, 16-bit auto reload ***

*** This version assumes 10 MHz crystal on 515c ***
*** 6 ms (precise) tick interval ***
*** Both Master and Slave(s) share the same tick rate ***
```

```
- *-----*/  
  
#include "Main.h"  
#include "Port.h"  
  
#include "Delay_T0.h"  
#include "TLight_B.h"  
#include "SCC_M515.h"  
  
// ----- Public variable definitions -----  
  
// Four bytes of data (plus ID information) are sent  
tByte Tick_message_data_G[NUMBER_OF_SLAVES][4] = {'M'};  
tByte Ack_message_data_G[NUMBER_OF_SLAVES][4];  
  
// ----- Public variable declarations -----  
  
// The array of tasks (see Sch51.c)  
extern sTask SCH_tasks_G[SCH_MAX_TASKS];  
  
// The error code variable (see Sch51.c)  
extern tByte Error_code_G;  
  
// ----- Private variable definitions -----  
  
static tByte Slave_index_G = 0;  
static bit First_ack_G = 1;  
  
// ----- Private function prototypes -----  
  
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;  
  
// ----- Private constants -----  
  
// Do not use ID 0x00 (used to start slaves)  
static const tByte MAIN_SLAVE_IDs[NUMBER_OF_SLAVES] = {0x01};  
static const tByte BACKUP_SLAVE_IDs[NUMBER_OF_SLAVES] = {0x02};  
  
// Number of ticks between reset attempts  
#define SLAVE_RESET_INTERVAL 0U  
  
#define NO_NETWORK_ERROR (1)
```

```

#define NETWORK_ERROR (0)

// ----- Private variables -----
static tWord Slave_reset_attempts_G[NUMBER_OF_SLAVES];

// Slave IDs may be any non-zero tByte value (but all must be different)
static tByte Current_Slave_IDs_G[NUMBER_OF_SLAVES] = {0};

/*-----*/
SCC_A_MASTER_Init_T2_CAN()

Scheduler initialization function. Prepares scheduler data
structures and sets up timer interrupts at required rate.
Must call this function before using the scheduler.

/*-----*/
void SCC_A_MASTER_Init_T2_CAN(void)
{
    tByte i;
    tByte Message;
    tByte Slave_index;

    // No interrupts (yet)
    EA = 0;

    // Start the watchdog
    SCC_A_MASTER_Watchdog_Init();

    Network_error_pin = NO_NETWORK_ERROR;

    // ----- Set up the scheduler -----
    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // We allow any combination of ID numbers in slaves
    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];
    }
}

```

```
}

// Get ready to send first tick message
First_ack_G = 1;
Slave_index_G = 0;

// ----- Set up the CAN link (begin) -----
// ----- SYSCON Register -----
// The access to XRAM and CAN controller is enabled.
// The signals !RD and !WR are not activated during accesses
// to the XRAM/CAN controller.
// ALE generation is enabled.
SYSCON = 0x20;

// ----- CAN Control/Status Register -----
// Start to init the CAN module
CAN_cr = 0x41; // INIT and CCE

// ----- Bit Timing Register -----
// Baudrate = 333.333 kbaud
// - Need 308+ kbaud plus for 1 ms ticks, 8 data bytes
// - See text for details
//
// There are 5 time quanta before sample point
// There are 4 time quanta after sample point
// The (re)synchronization jump width is 2 time quanta
CAN_btr1 = 0x34; // Bit Timing Register
CAN_btr0 = 0x42;

CAN_gms1 = 0xFF; // Global Mask Short Register 1
CAN_gms0 = 0xFF; // Global Mask Short Register 0

CAN_ugm11 = 0xFF; // Upper Global Mask Long Register 1
CAN_ugm10 = 0xFF; // Upper Global Mask Long Register 0

CAN_lgm11 = 0xF8; // Lower Global Mask Long Register 1
CAN_lgm10 = 0xFF; // Lower Global Mask Long Register 0

// --- Configure the 'Tick' Message Object ---
// 'Message Object 1' is valid
CAN_messages[0].MCR1 = 0x55; // Message Control Register 1
CAN_messages[0].MCR0 = 0x95; // Message Control Register 0

// Message direction is transmit
// Extended 29-bit identifier
// These have ID 0x000000 and 5 valid data bytes
```

```

CAN_messages[0].MCFG = 0x5C;           // Message Configuration Register

CAN_messages[0].UAR1 = 0x00;           // Upper Arbit. Reg. 1
CAN_messages[0].UAR0 = 0x00;           // Upper Arbit. Reg. 0
CAN_messages[0].LAR1 = 0x00;           // Lower Arbit. Reg. 1
CAN_messages[0].LAR0 = 0x00;           // Lower Arbit. Reg. 0

CAN_messages[0].Data[0] = 0x00;         // data byte 0
CAN_messages[0].Data[1] = 0x00;         // data byte 1
CAN_messages[0].Data[2] = 0x00;         // data byte 2
CAN_messages[0].Data[3] = 0x00;         // data byte 3
CAN_messages[0].Data[4] = 0x00;         // data byte 4

// --- Configure the 'Ack' Message Object ---

// 'Message Object 2' is valid
// NOTE: Object 2 receives *ALL* ack messages
CAN_messages[1].MCR1 = 0x55;           // Message Control Register 1
CAN_messages[1].MCR0 = 0x95;           // Message Control Register 0

// Message direction is receive
// Extended 29-bit identifier
// These all have ID: 0x000000FF (5 valid data bytes)
CAN_messages[1].MCFG = 0x04;           // Message Configuration Register

CAN_messages[1].UAR1 = 0x00;           // Upper Arbit. Reg. 1
CAN_messages[1].UAR0 = 0x00;           // Upper Arbit. Reg. 0
CAN_messages[1].LAR1 = 0xF8;           // Lower Arbit. Reg. 1
CAN_messages[1].LAR0 = 0x07;           // Lower Arbit. Reg. 0

// Configure remaining message objects - none are valid
for (Message = 2; Message <= 14; ++Message)
{
    CAN_messages[Message].MCR1 = 0x55; // Message Control Register 1
    CAN_messages[Message].MCR0 = 0x55; // Message Control Register 0
}

// ----- CAN Control Register -----
// reset CCE and INIT
CAN_cr = 0x00;

// ----- Set up the CAN link (end) -----

// ----- Set up Timer 2 (begin) -----
// 80c515c, 10 MHz
// Timer 2 is set to overflow every 6 ms - see text
// Mode 1 = Timerfunction

// Prescaler: Fcpu/12

```

```
T2PS = 1;

// Mode 0 = auto-reload upon timer overflow
// Preset the timer register with autoreload value
// NOTE: Timing is same as standard (8052) T2 timing
// - if T2PS = 1 (otherwise twice as fast as 8052)
TL2 = 0x78;
TH2 = 0xEC;

// Mode 0 for all channels
T2CON |= 0x11;

// timer 2 overflow interrupt is enabled
ET2 = 1;
// timer 2 external reload interrupt is disabled
EXEN2 = 0;

// Compare/capture Channel 0
// Disabled
// Compare Register CRC on: 0x0000;
CRCL = 0x78;
CRCH = 0xEC;

// CC0/ext3 interrupt is disabled
EX3 = 0;

// Compare/capture Channel 1-3
// Disabled
CCL1 = 0x00;
CCH1 = 0x00;
CCL2 = 0x00;
CCH2 = 0x00;
CCL3 = 0x00;
CCH3 = 0x00;

// Interrupts Channel 1-3
// Disabled
EX4 = 0;
EX5 = 0;
EX6 = 0;

// all above mentioned modes for Channel 0 to Channel 3
CCEN = 0x00;
// ----- Set up Timer 2 (end) -----
}

/*-----*-
```

```
SCC_A_MASTER_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

-*-----*/
void SCC_A_MASTER_Start(void)
{
    tByte Num_active_slaves;
    tByte i;
    bit Slave_replied_correctly;
    tByte Slave_index, Slave_ID;

    // Refresh the watchdog
    SCC_A_MASTER_Watchdog_Refresh();

    // Place system in 'safe state'
    SCC_A_MASTER_Enter_Safe_State();

    // Report error as we wait to start
    Network_error_pin = NETWORK_ERROR;

    Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
    SCH_Report_Status(); // Sch not yet running - do this manually

    // Pause here (300 ms), to timeout all the slaves
    // (This is the means by which we synchronize the network)
    for (i = 0; i < 10; i++)
    {
        Hardware_Delay_T0(30);
        SCC_A_MASTER_Watchdog_Refresh();
    }

    // Currently disconnected from all slaves
    Num_active_slaves = 0;

    // After the initial (long) delay, all (operational) slaves will
    // have timed out.
    // All operational slaves will now be in the 'READY TO START' state
    // Send them a 'slave id' message to get them started
    Slave_index = 0;
    do {
        // Refresh the watchdog
        SCC_A_MASTER_Watchdog_Refresh();
```

```

// Find the slave ID for this slave
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
{
    // Slave did not reply correctly
    // - try to switch to backup device (if available)
    if (Current_Slave_IDs_G[Slave_index] != BACKUP_SLAVE_IDS[Slave_index])
    {
        // There is a backup available: switch to backup and try again
        Current_Slave_IDs_G[Slave_index] =
            BACKUP_SLAVE_IDS[Slave_index];
    }
    else
    {
        // No backup available (or backup failed too) - have to
        // continue
        Slave_index++;
    }
}
} while (Slave_index < NUMBER_OF_SLAVES);

// DEAL WITH CASE OF MISSING SLAVE(S) HERE ...
if (Num_active_slaves < NUMBER_OF_SLAVES)
{
    // User-defined error handling here...
    // 1 or more slaves have not replied
    // NOTE: In some circumstances you may wish to abort if slaves
    // are missing
    // - or reconfigure the network.

    // Simplest solution is to display an error and carry on
    // (this is what we do here)
    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;
    }

    // Start the scheduler
    IRCON = 0;
    EA = 1;
}

/*-----*
 * SCC_A_MASTER_Update_T2
 *
 * This is the scheduler ISR. It is called at a rate determined by
 * the timer settings in SCC_A_MASTER_Init_T2(). This version is
 * triggered by Timer 2 interrupts: timer is automatically reloaded.
 *-----*/
void SCC_A_MASTER_Update_T2(void) interrupt INTERRUPT_Timer_2_Overflow
{
    tByte Index;
    tByte Previous_slave_index;
    bit Slave_replied_correctly;

    TF2 = 0; // Must clear this.

    // Refresh the watchdog
    SCC_A_MASTER_Watchdog_Refresh();

    // Default
    Network_error_pin = NO_NETWORK_ERROR;

    // Keep track of the current slave
    Previous_slave_index = Slave_index_G; // First value of prev slave
                                            // is 0...

    if (++Slave_index_G >= NUMBER_OF_SLAVES)
    {
        Slave_index_G = 0;
    }

    // Check that the appropriate slave responded to the previous message:
    // (if it did, store the data sent by this slave)
    if (SCC_A_MASTER_Process_Ack(Previous_slave_index) == RETURN_ERROR)
    {
        Error_code_G = ERROR_SCH_LOST_SLAVE;
        Network_error_pin = NETWORK_ERROR;
    }
}

```

```

// If we have lost contact with a slave, we attempt to
// switch to a backup device (if one is available)
if (Current_Slave_IDs_G[Slave_index_G] !=  

    BACKUP_SLAVE_IDS[Slave_index_G])
{
    // There is a backup available: switch to backup and try again
    Current_Slave_IDs_G[Slave_index_G] =  

        BACKUP_SLAVE_IDS[Slave_index_G];
}
else
{
    // There is no backup available (or we are already using it)
    // Try main device.
    Current_Slave_IDs_G[Slave_index_G] =  

        MAIN_SLAVE_IDS[Slave_index_G];
}

// Try to connect to the slave
Slave_replied_correctly =
SCC_A_MASTER_Start_Slave(Current_Slave_IDs_G[Slave_index_G]);

if (!Slave_replied_correctly)
{
    // No backup available (or backup failed too) - we shut down
    // OTHER BEHAVIOUR MAY BE MORE APPROPRIATE IN YOUR APPLICATION
    SCC_A_MASTER_Shut_Down_the_Network();
}
}

// Send 'tick' message to all connected slaves
// (sends one data byte to the current slave)
SCC_A_MASTER_Send_Tick_Message(Slave_index_G);

// Check the last error codes on the CAN bus via the status register
if ((CAN_sr & 0x07) != 0)
{
    Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
    Network_error_pin = NETWORK_ERROR;

    // See Infineon c515c manual for error code details
    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;
    }

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            if (SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Incr. the 'Run Me' flag

                if (SCH_tasks_G[Index].Period)
                {
                    // Schedule periodic tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }

    /*-----*
     SCC_A_MASTER_Send_Tick_Message()

     This function sends a tick message, over the CAN network.
     The receipt of this message will cause an interrupt to be generated
     in the slave(s): this invoke the scheduler 'update' function
     in the slave(s).

     *-----*/
void SCC_A_MASTER_Send_Tick_Message(const tByte SLAVE_INDEX)
{
    // Find the slave ID for this slave
    // ALL SLAVES MUST HAVE A UNIQUE (non-zero) ID
}

```

```

tByte Slave_ID = (tByte) Current_Slave_IDS_G[SLAVE_INDEX];
CAN_messages[0].Data[0] = Slave_ID;

// Fill the data fields
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];

// Send the message on the CAN bus
CAN_messages[0].MCR1 = 0xE7; // TXRQ, reset CPUUPD
}

/*-----*
SCC_A_MASTER_Process_Ack()

Make sure the slave (SLAVE_ID) has acknowledged the previous
message that was sent. If it has, extract the message data
from the USART hardware: if not, call the appropriate error
handler.

PARAMS: The index of the slave.

RETURNS: RETURN_NORMAL - Ack received (data in Ack_message_data_G)
         RETURN_ERROR - No ack received (-> no data)

*-----*/
bit SCC_A_MASTER_Process_Ack(const tByte SLAVE_INDEX)
{
    tByte Ack_ID, Slave_ID;

    // First time this is called there is no ack tick to check
    // - we simply return 'OK'
    if (First_ack_G)
    {
        First_ack_G = 0;
        return RETURN_NORMAL;
    }

    if ((CAN_messages[1].MCR1 & 0x03) == 0x02) // if NEWDAT
    {
        // An ack message was received
        //
        // Extract the data
        Ack_ID = CAN_messages[1].Data[0]; // Get data byte 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; // reset NEWDAT, INTPND
CAN_messages[1].MCR1 = 0xfd;

// Find the slave ID for this slave
Slave_ID = (tByte) Current_Slave_IDs_G[SLAVE_INDEX];

if (Ack_ID == Slave_ID)
{
    return RETURN_NORMAL;
}
}

// No message, or ID incorrect
return RETURN_ERROR;
}

/*-----*/
SCC_A_MASTER_Shut_Down_the_Network()

This function will be called when a slave fails to
acknowledge a tick message.

/*-----*/
void SCC_A_MASTER_Shut_Down_the_Network(void)
{
EA = 0;

while(1)
{
    SCC_A_MASTER_Watchdog_Refresh();
}
}

/*-----*/
SCC_A_MASTER_Enter_Safe_State()

This is the state entered by the system when:
(1) The node is powered up or reset
(2) The Master node cannot detect a slave
(3) The network has an error

Try to ensure that the system is in a 'safe' state in these
circumstances.

/*-----*/

```

```

void SCC_A_MASTER_Enter_Safe_State(void)
{
    // USER DEFINED - Edit as required

    TRAFFIC_LIGHTS_Display_Safe_Output();
}

/* -----
   SCC_A_MASTER_Watchdog_Init()

This function sets up the watchdog timer.

- */
void SCC_A_MASTER_Watchdog_Init(void)
{
    // Watchdog timer prescaler (1/16) enabled
    // Watchdog timer reload value is 0x6B
    // Oscillator is 10 MHz -> watchdog period is ~103 ms
    WDTREL = 0xEB;

    // Start watchdog timer
    WDT = 1;
    SWDT = 1;
}

/* -----
   SCC_A_MASTER_Watchdog_Refresh()

Feed the internal c515c watchdog.

- */
void SCC_A_MASTER_Watchdog_Refresh(void) reentrant
{
    WDT = 1;
    SWDT = 1;
}

/* -----
   SCC_A_MASTER_Start_Slave()

Try to connect to a slave device.

- */
tByte SCC_A_MASTER_Start_Slave(const tByte SLAVE_ID) reentrant
{
    tByte Slave_replied_correctly = 0;

    // tByte Slave_ID;
    tByte Ack_ID, Ack_00;
}

```

```

    // Send a 'Slave ID' message
    CAN_messages[0].Data[0] = 0x00; // Not a valid slave ID
    CAN_messages[0].Data[1] = SLAVE_ID;
    CAN_messages[0].MCR1 = 0xE7; // Send it

    // Wait to give slave time to reply
    Hardware_Delay_T0(5);

    // Check we had a reply
    if ((CAN_messages[1].MCR1 & 0x03) == 0x02) // if NEWDAT
    {
        // An ack message was received - extract the data
        Ack_00 = (tByte) CAN_messages[1].Data[0]; // Get data byte 0
        Ack_ID = (tByte) CAN_messages[1].Data[1]; // Get data byte 1

        CAN_messages[1].MCR0 = 0xfd; // reset NEWDAT, INTPND
        CAN_messages[1].MCR1 = 0xfd;

        if ((Ack_00 == 0x00) && (Ack_ID == SLAVE_ID))
        {
            Slave_replied_correctly = 1;
        }
    }

    return Slave_replied_correctly;
}

/* -----
----- END OF FILE
----- */

```

Listing 28.1 Part of the software for a shared-clock CAN scheduler (Master node)

Slave software

```

/* -----
SCC_S515.c (v1.00)

-----
THIS IS A SHARED-SCHEDULER [CAN BASED] FOR 80C515C (etc.)

*** Both Master and Slave share the same tick rate ***
*** - See Master code for details ***

----- */

```

```
#include "Main.h"
#include "Port.h"
#include "SCC_S515.h"
#include "TLight_B.h"

// ----- Public variable definitions -----
// Data sent from the master to this slave
tByte Tick_message_data_G[4];

// Data sent from this slave to the master
// - data may be sent on, by the master, to another slave
tByte Ack_message_data_G[4];

// ----- Public variable declarations -----
// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

// ----- Private function prototypes -----
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;

// ----- Private constants -----
// Each slave (and backup) must have a unique (non-zero) ID
#define SLAVE_ID 0x01

#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)

/*-----*
 * SCC_A_SLAVE_Init_CAN()
 *
 * Scheduler initialization function. Prepares scheduler
 * data structures and sets up timer interrupts at required rate.
 * Must call this function before using the scheduler.
 */
```

```

-----*/
void SCC_A_SLAVE_Init_CAN(void)
{
    tByte i;
    tByte Message;

    // Sort out the tasks
    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // Set the network error pin (reset when tick message received)
    Network_error_pin = NETWORK_ERROR;

    // ----- SYSCON Register
    // The access to XRAM and CAN controller is enabled.
    // The signals !RD and !WR are not activated during accesses
    // to the XRAM/CAN controller.
    // ALE generation is enabled.
    SYSCON = 0x20;

    // ----- CAN Control/Status Register -----
    // Start to init the CAN module
    CAN_cr = 0x41; // INIT and CCE

    // ----- Bit Timing Register -----
    // Baudrate = 333.333 kbaud
    // - Need 308+ kbaud plus for 1 ms ticks, 8 data bytes
    // - See text for details
    //

    // There are 5 time quanta before sample point
    // There are 4 time quanta after sample point
    // The (re)synchronization jump width is 2 time quanta
    CAN_btr1 = 0x34; // Bit Timing Register
    CAN_btr0 = 0x42;

    CAN_gms1 = 0xFF; // Global Mask Short Register 1
    CAN_gms0 = 0xFF; // Global Mask Short Register 0

    CAN_ugm11 = 0xFF; // Upper Global Mask Long Register 1
    CAN_ugm10 = 0xFF; // Upper Global Mask Long Register 0

```

```
CAN_lgm11 = 0xF8; // Lower Global Mask Long Register 1
CAN_lgm10 = 0xFF; // Lower Global Mask Long Register 0

// ----- Configure 'Tick' Message Object
// Message object 1 is valid
// enable receive interrupt
CAN_messages[0].MCR1 = 0x55; // Message Ctrl. Reg. 1
CAN_messages[0].MCR0 = 0x99; // Message Ctrl. Reg. 0

// Message direction is receive
// extended 29-bit identifier
CAN_messages[0].MCFG = 0x04; // Message Config. Reg.

CAN_messages[0].UAR1 = 0x00; // Upper Arbit. Reg. 1
CAN_messages[0].UAR0 = 0x00; // Upper Arbit. Reg. 0
CAN_messages[0].LAR1 = 0x00; // Lower Arbit. Reg. 1
CAN_messages[0].LAR0 = 0x00; // Lower Arbit. Reg. 0

// ----- Configure 'Ack' Message Object
CAN_messages[1].MCR1 = 0x55; // Message Ctrl. Reg. 1
CAN_messages[1].MCR0 = 0x95; // Message Ctrl. Reg. 0

// Message direction is transmit
// extended 29-bit identifier
// 5 valid data bytes
CAN_messages[1].MCFG = 0x5C; // Message Config. Reg.

CAN_messages[1].UAR1 = 0x00; // Upper Arbit. Reg. 1
CAN_messages[1].UAR0 = 0x00; // Upper Arbit. Reg. 0
CAN_messages[1].LAR1 = 0xF8; // Lower Arbit. Reg. 1
CAN_messages[1].LAR0 = 0x07; // Lower Arbit. Reg. 0

CAN_messages[1].Data[0] = 0x00; // data byte 0
CAN_messages[1].Data[1] = 0x00; // data byte 1
CAN_messages[1].Data[2] = 0x00; // data byte 2
CAN_messages[1].Data[3] = 0x00; // data byte 3
CAN_messages[1].Data[4] = 0x00; // data byte 4

// ----- Configure other objects -----
// Configure remaining message objects (2-14) - none are valid
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 Ctrl. Reg. -----
// reset CCE and INIT
// enable interrupt generation from CAN Modul
// enable CAN-interrupt of Controller
CAN_cr = 0x02;
IEN2 |= 0x02;

// Start the watchdog
SCC_A_SLAVE_Watchdog_Init();
}

/*-----*/
SCC_A_SLAVE_Start()

Starts the slave scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

/*
void SCC_A_SLAVE_Start(void)
{
tByte Tick_00, Tick_ID;
bit Start_slave;

// Disable interrupts
EA = 0;

// We can be at this point because:
// 1. The network has just been powered up
// 2. An error has occurred in the Master, and it is not generating
// ticks
// 3. The network has been damaged and no ticks are being received
// by this slave
//
// Try to make sure the system is in a safe state...
// NOTE: Interrupts are disabled here
SCC_A_SLAVE_Enter_Safe_State();

Start_slave = 0;
Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
SCH_Report_Status(); // Sch not yet running - do this manually

// Now wait (indefinitely) for appropriate signal from the master
do {
```

```

// Wait for 'Slave ID' message to be received
do {
    SCC_A_SLAVE_Watchdog_Refresh(); // Must keep feeding the
                                    // watchdog
} while ((CAN_messages[0].MCR1 & 0x03) != 0x02);

// Got a message - extract the data
if ((CAN_messages[0].MCR1 & 0x0c) == 0x08) // if MSGLST set
{
    // Ignore lost message
    CAN_messages[0].MCR1 = 0xf7; // reset MSGLST
}

Tick_00 = (tByte) CAN_messages[0].Data[0]; // Get data byte 0
Tick_ID = (tByte) CAN_messages[0].Data[1]; // Get data byte 1

CAN_messages[0].MCR0 = 0xfd; // reset NEWDAT, INTPND
CAN_messages[0].MCR1 = 0xfd;

if ((Tick_00 == 0x00) && (Tick_ID == SLAVE_ID))
{
    // Message is correct
    Start_slave = 1;

    // Send ack
    CAN_messages[1].Data[0] = 0x00; // Set data byte 0
    CAN_messages[1].Data[1] = SLAVE_ID; // Set data byte 1
    CAN_messages[1].MCR1 = 0xE7; // Send it
}
else
{
    // Not yet received correct message - wait
    Start_slave = 0;
}
} while (!Start_slave);

// Start the scheduler
IRCON = 0;
EA = 1;
}

```

/*-----*

SCC_A_SLAVE_Update

This is the scheduler ISR. It is called at a rate
determined by the timer settings in SCC_A_SLAVE_Init().

This Slave is triggered by USART interrupts.

```
- *-----*/  
void SCC_A_SLAVE_Update(void) interrupt INTERRUPT_CAN_c515c  
{  
    tByte Index;  
  
    // Reset this when tick is received  
    Network_error_pin = NO_NETWORK_ERROR;  
  
    // Check tick data - send ack if necessary  
    // NOTE: 'START' message will only be sent after a 'timeout'  
    if (SCC_A_SLAVE_Process_Tick_Message() == SLAVE_ID)  
    {  
        SCC_A_SLAVE_Send_Ack_Message_To_Master();  
  
        // Feed the watchdog ONLY when a *relevant* message is received  
        // (noise on the bus, etc, will not stop the watchdog...)  
        //  
        // START messages will NOT refresh the slave  
        // - Must talk to every slave at regular intervals  
        SCC_A_SLAVE_Watchdog_Refresh();  
    }  
  
    // Check the last error codes on the CAN bus via the status register  
    if ((CAN_sr & 0x07) != 0)  
    {  
        Error_code_G = ERROR_SCH_CAN_BUS_ERROR;  
        Network_error_pin = NETWORK_ERROR;  
  
        // See Infineon c515c manual for error code details  
        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;  
    }  
  
    // NOTE: calculations are in *TICKS* (not milliseconds)  
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)  
    {  
        // Check if there is a task at this location
```

```

    if (SCH_tasks_G[Index].pTask)
    {
        if (SCH_tasks_G[Index].Delay == 0)
        {
            // The task is due to run
            SCH_tasks_G[Index].RunMe = 1; // Increment the RunMe flag

            if (SCH_tasks_G[Index].Period)
            {
                // Schedule periodic tasks to run again
                SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
            }
        }
        else
        {
            // Not yet ready to run: just decrement the delay
            SCH_tasks_G[Index].Delay -= 1;
        }
    }
}

/*
SCC_A_SLAVE_Process_Tick_Message()

The ticks messages are crucial to the operation of this shared-clock
scheduler: the arrival of a tick message (at regular intervals)
invokes the 'Update' ISR, that drives the scheduler.

The tick messages themselves may contain data. These data are
extracted in this function.
*/

```

```

tByte SCC_A_SLAVE_Process_Tick_Message(void)
{
    tByte Tick_ID;

    if ((CAN_messages[0].MCR1 & 0x0c) == 0x08) // if MSGLST set
    {
        // Indicates that the CAN controller has stored a new
        // message into this object, while NEWDAT was still set,
        // i.e. the previously stored message is lost.

        // We simply IGNORE this here and reset the flag
        CAN_messages[0].MCR1 = 0xf7; // reset MSGLST
    }
}

```

```

// The first byte is the ID of the slave for which the data are
// intended
Tick_ID = CAN_messages[0].Data[0]; // Get data byte 0 (Slave ID)

if (Tick_ID == SLAVE_ID)
{
    // Only if there is a match do we need to copy these fields
    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; // reset NEWDAT, INTPND
CAN_messages[0].MCR1 = 0xfd;

return Tick_ID;
}

/*-----*
 * SCC_A_SLAVE_Send_Ack_Message_To_Master()
 *
 Slave must send an 'Acknowledge' message to the master, after
 tick messages are received. NOTE: Only tick messages specifically
 addressed to this slave should be acknowledged.

 The acknowledge message serves two purposes:
 [1] It confirms to the master that this slave is alive & well.
 [2] It provides a means of sending data to the master and - hence
     - to other slaves.
 *
 NOTE: Data transfer between slaves is NOT permitted!
 *-----*/
void SCC_A_SLAVE_Send_Ack_Message_To_Master(void)
{
    // First byte of message must be slave ID
    CAN_messages[1].Data[0] = SLAVE_ID; // data byte 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];

    // Send the message on the CAN bus
    CAN_messages[1].MCR1 = 0xE7; // TXRQ, reset CPUUPD
}

```

```
/*-----*  
SCC_A_SLAVE_Watchdog_Init()  
This function sets up the watchdog timer.  
If the Master fails (or other error develop),  
no tick messages will arrive, and the scheduler  
will stop.  
To detect this situation, we have a (hardware) watchdog  
running in the slave. This watchdog - which should be set to  
overflow at around 100 ms - is used to set the system into a  
known (safe) state. The slave will then wait (indefinitely)  
for the problem to be resolved.  
NOTE: The slave will not be generating Ack messages in these  
circumstances. The Master (if running) will therefore be aware  
that there is a problem.  
-----*/  
void SCC_A_SLAVE_Watchdog_Init(void)  
{  
    // Watchdog timer prescaler (1/16) enabled  
    // Watchdog timer reload value is 0x6B  
    // Watchdog period is 103.2 ms (10.0 MHz xtal, c515c)  
    WDTREL = 0xEB;  
  
    // Start watchdog timer  
    WDT = 1;  
    SWDT = 1;  
}  
  
/*-----*  
SCC_A_SLAVE_Watchdog_Refresh()  
Feed the watchdog.  
-----*/  
void SCC_A_SLAVE_Watchdog_Refresh(void) reentrant  
{  
    WDT = 1;  
    SWDT = 1;  
}  
  
/*-----*  
SCC_A_SLAVE_Enter_Safe_State()
```

This is the state entered by the system when:

- (1) The node is powered up or reset
- (2) The Master node fails, and no working backup is available
- (3) The network has an error
- (4) Tick messages are delayed for any other reason

Try to ensure that the system is in a 'safe' state in these circumstances.

```
- *-----*/  
void SCC_A_SLAVE_Enter_Safe_State(void)  
{  
    // USER DEFINED  
    TRAFFIC_LIGHTS_Display_Safe_Output();  
}  
  
/*-----  
--- END OF FILE -----  
-----*/
```

Listing 28.2 Part of the software for a shared-clock CAN scheduler (Slave node)

Backup slave software

See the CD for details of the code for the backup Slave.

Further reading

- 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]

Designing multiprocessor applications

Introduction

There are a number of important design problems which must be addressed when using shared-clock schedulers. We address some of the key problems in this chapter. Specifically, we consider the following:

- In **DATA UNION** [page 712], we consider the problem of transferring (possibly large amounts of) data between Master and Slave nodes, using a communication channel of limited bandwidth.
- In **LONG TASK** [page 716], we consider a perennial problem in scheduled systems; the need to handle both long, infrequently invoked tasks and short, frequently invoked tasks. In this case, we consider techniques for migrating one or more long task(s) onto a Slave, thereby freeing up the Master node to handle shorter tasks.
- In **DOMINO TASK** [page 720], we consider the problem of distributing processing between tasks operating on two or more network nodes; specifically, we consider the situation where the tasks are of similar duration and must be invoked in sequence.

DATA UNION

Context

- You are developing an embedded application using more than one member of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a shared-clock scheduler.

Problem

How do you transfer data between Master and Slave nodes, using a communication channel of limited bandwidth?

Background

Suppose we are using a shared-clock, UART-based scheduler. The communication channel is eight bits wide; that is, we can send only eight bits of data between Master and a Slave (or vice versa) in any network message. How do we transfer, say, integer or floating-point data over this limited bandwidth channel? This is the type of problem we are seeking to address in the present pattern.

Solution

To implement **DATA UNION**, we will use the C `union` keyword. In C, a `union`-based variable can hold, at different times, values of different types and sizes, with the compiler ensuring that size and alignment are maintained. Different types of data can therefore be accommodated in a single area of storage.

To create a variable using a `union` we use the same syntax as the widely used `struct`, but whereas a `struct`-based variable might hold an `int`, a `double` and a `char`, a `union` can hold an `int` or a `double` or a `char`.

For example, consider the code in Listing 29.1.

```
// An example using the C/C++ union keyword

#include <stdio.h>

// A user-defined data type based on a union
typedef union
{
    int    Integer;
    float  Float;
} uNumber;
```

```
int main(void)
{
    uNumber Value;

    Value.Integer = 100;
    printf("%s\n%s\n%s%d\n%s%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%s%d\n%s%f\n",
           "Put a value in the floating member",
           "and print both members.",
           "int:   ", Value.Integer,
           "float: ", Value.Float);

    return 0;
}
```

Listing 29.1 An example using the C/C++ union keyword

An output from the program in Listing 29.1 is shown in Figure 29.1.

Careful use of the `union` keyword can allow the efficient transfer of data between nodes in a shared-clock network, as we illustrate in the example that follows.

```
Put a value in the integer member
and print both members.
int: 100
float: 0.000000

Put a value in the floating member
and print both members.
int: 0
float: 100.000000
```

FIGURE 29.1 An output from the program in Listing 29.1

Hardware resource implications

There are no significant hardware resource implications.

Reliability and safety implications

If used as intended here – to transfer data between two 8051 microcontrollers, with code on each microcontroller created using the same compiler – these techniques may be used safely and reliably, without difficulty.

However, if you attempt to use this technique to transfer data to a different microcontroller or microprocessor family, then problems can arise: see 'Portability' for further details.

Portability

This *technique* is portable, in that it can be applied to any microcontroller or microprocessor.

That said, the *data themselves* may not be portable. For example, if you attempt to use this approach to transfer data between an 8-bit and 16-bit microcontroller or an 8-bit microcontroller and a (32-bit) PC, then care must be taken, since the code assumes that the data (particularly the floating-point data) are of a particular size and stored in a particular byte order; in fact, the size and order of the data varies between compilers and operating environments.

For example, while a float may be represented in four bytes on an 8051 device, an 8-byte or 16-byte representation may be used in other environments. If you attempt to directly interpret an encoded 4-byte float in an 8-byte environment, the results will be meaningless.

Care must therefore be taken if this approach is used to transfer data between different environments.

Overall strengths and weaknesses

- ☺ Simple and effective as a way of sending data between 8051 microcontrollers over a byte-wide communication channel.
- ☹ Care must be taken if the technique is used to transfer data between 8- and 16- or 32-bit environments.

Related patterns and alternative solutions

DATA UNION is a form of MULTI-STAGE TASK [page 317].

Example: Transferring floats between microcontrollers

We provide a simple example illustrating the transfer of (32-bit) floats across an 8-bit communication network here.

```
// Breaking up floats (etc) for transfer over a serial link
// or via one or more parallel ports

// Assume float is 4 bytes
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);

    // Simulate transfer of floats over byte-wide communication link
    for (byte = 0; byte < 4; byte++)
    {
        Y.Bytes[byte] = X.Bytes[byte];
    }

    // Now display 'transferred data'
    printf("Original data is %f\n", X.Float);
}
```

Further reading

Schildt, H. (1997) *Teach Yourself C*, 3rd edn, McGraw-Hill, Maidenhead.

LONG TASK

Context

- You are developing an embedded application using more than one member of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a shared-clock scheduler.

Problem

How can you handle both long (infrequent) tasks and short (frequent) tasks in a multiprocessor application?

Background

As we have seen, co-operative, time-triggered architectures have many advantages when we wish to develop embedded applications; indeed, throughout this book we have argued that we would generally wish to use such an architecture when it is feasible to do so. However, we have also taken a realistic view of the limitations of the co-operative scheduler; in particular we have acknowledged that in some circumstances there can be a need to run both long, infrequent tasks (e.g. 100 ms duration every 1,000 ms) and one or more short, frequent task (e.g. 0.1 ms duration every 1 ms); these two requirements can conflict in a co-operative system, where – for all tasks, under all circumstances – the worst case execution time (WCET) for a task, that is, the maximum task duration, must satisfy the condition:

$$WCET_{Task} < \text{tick interval}$$

In this pattern collection we have already considered various ways of meeting the need for both ‘frequent’ and ‘long’ tasks. For example, by using a faster processor (see Chapter 3) or a faster oscillator (Chapter 4) we can shorten the task execution times. Alternatively, we can make use of timeouts (see Chapter 15), develop one or more multi-stage tasks (see Chapter 16) or use a hybrid scheduler (see Chapter 17).

In this pattern, we consider a multiprocessor solution to this problem.

Solution

In this pattern, we discuss the design and use of a system architecture which is based on a shared-clock scheduler and which – at its simplest – takes the following form (see Figures 29.2 and 29.3):

- A single short task (e.g. WCET 0.1 ms) runs frequently (e.g. every millisecond) on the Master node. This task may, for example, be used to check for errors or handle some other activity that requires a rapid response.

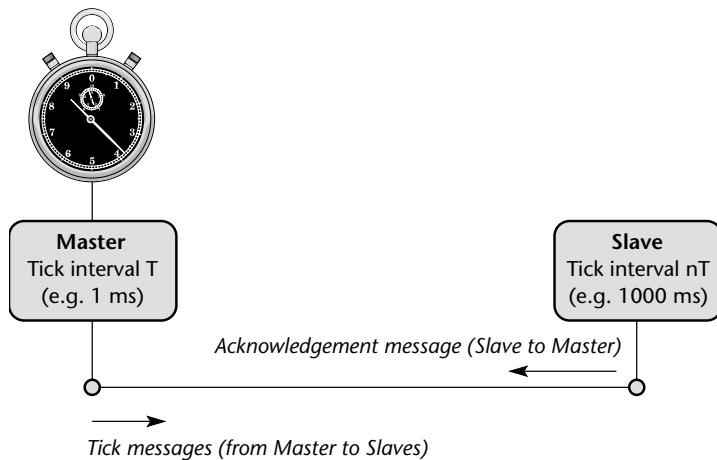


FIGURE 29.2 The 'long task' architecture

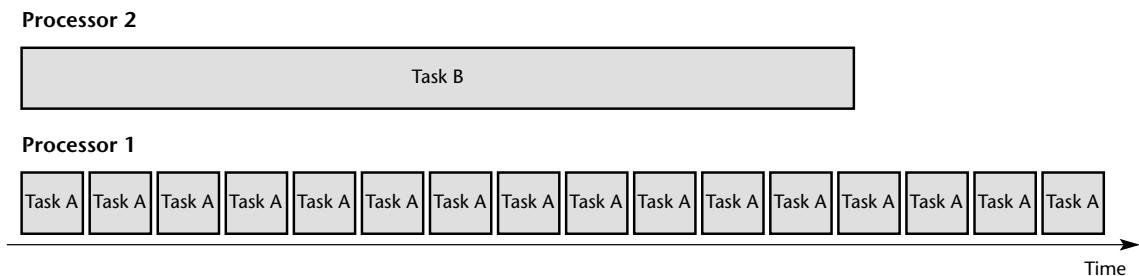


FIGURE 29.3 The task structure in the 'long task' architecture

- A single long task (e.g. WCET 500 ms) runs infrequently (e.g. every second) on a single Slave node. This might be implementing, for example, a data-processing or data-compression algorithm.
- In most cases, the Master node and Slave node will have different tick intervals. Specifically, the Master will send tick messages to the Slave at a rate that matches the long task duration. For example, in this case, the Master might have a 1 ms tick interval, while the Slave has a 1,000 ms tick interval; thus, the Master will send ticks to the Slave every 1,000 calls of the (Master) 'Update' function.

Note that:

- Multiple Slave nodes may be used.
- All Slave nodes will generally share the same (usually slow) tick rate.
- As described, there is no direct means for the Master to abort the Slave processing.

Hardware resource implications

This pattern requires the use of at least two microcontrollers and associated hardware (for example, reset, oscillator, memory and power supplies).

Reliability and safety implications

The correct operation of **LONG TASK** depends on all the microcontrollers involved operating normally. This requirement can, under some circumstances, reduce the overall reliability of your system: see page 550 for a discussion of this issue.

Portability

This general design pattern may be applied with any microcontroller or microprocessor family.

Overall strengths and weaknesses

- ☺ An effective way of supporting the execution of both long, infrequent tasks and short, frequent tasks.
- ☹ Like any multiprocessor design, it must be used with care if overall system reliability is not to be compromised.

Related patterns and alternative solutions

- See **HYBRID SCHEDULER** [page 333].
- See **DATA UNION** [page 712].
- See **DOMINO TASK** [page 720].

Example: Data acquisition and FFT

In many condition-monitoring applications, we need to:

- Acquire data, via an ADC
- Perform a fast Fourier transform (FFT) on these data, prior to subsequent processing

The data-acquisition process will be very short, typically requiring a fraction of a millisecond to complete. By contrast, the FFT operation may be carried out every time we have amassed, say, 256 data samples. It is a computationally intense process (e.g. see Press *et al.*, 1992; Lynn and Fuerst, 1998; Smith, 1999) and must usually be completed before the next set of 256 samples has been acquired.

Use of the **LONG TASK** architecture is ideal in these circumstances.

Further reading

- 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]

DOMINO TASK

Context

- You are developing an embedded application using more than one member of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a shared-clock scheduler.

Problem

How do you make best use of an architecture involving more than one microcontroller when you must use tasks that are scheduled consecutively?

Background

The improved performance achieved through the use of more than one microcontroller is superficially obvious: if one microcontroller has a performance of 1 MIP and we need 2 MIPs, we can use two processors.

In reality, the situation is rather more complicated. Because of the need for communication between the two processors, we will not necessarily get a doubling in performance: indeed, we may see a performance *decrease* in some circumstances.

Consider, for example, that we are required to perform three consecutive tasks (Task A, Task B, Task C) on some data obtained from a ‘data source’, as indicated in Figure 29.4.

If we use one processor to perform these tasks, the time taken to perform the three tasks and generate ‘Data C’ will be:

$$\text{Duration}_1 = A + B + C + \text{Transfer}_{A-B} + \text{Transfer}_{B-C}$$

That is, the total duration is the duration of Task A, plus the duration of Task B and Task C, plus the time taken to transfer the intermediate data between various tasks.

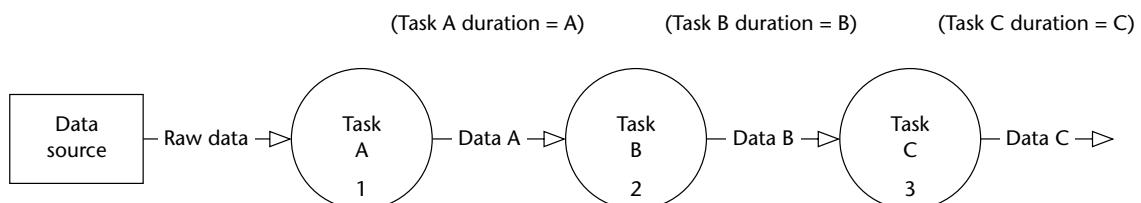


FIGURE 29.4 Performing three consecutive tasks

Now, suppose we carry out the same activity on three different microprocessors, each with a performance level the same as that assumed in the first example. The time taken will be:

$$\text{Duration}_3 = A + B + C + \text{Transfer}_{A-B} + \text{Transfer}_{B-C}$$

In this case, the second processor must wait for the Data A to become available before it can begin processing and the third processor must wait for Data B before it can begin processing: as a result, we see no immediate performance improvement. Worse, if we make the reasonable assumption that the time taken to transfer the data between tasks is greater where the two tasks are running on different processors, then the overall duration will be *greater* in the three-processor situation.

However, this is not the only way of assessing the performance of this application. If, instead of the task duration, we focus on the *data coverage*, we see a different picture. Consider Figure 29.5.

In Figure 29.5 we illustrate, in a simplified form, the single-processor version of the three-task system. Here, during the execution of Task B, we are unable to obtain data. As a result, if Task A and Task B are of similar duration, then for approximately half the time, no data capture is being carried out.

Compare this with Figure 29.6.

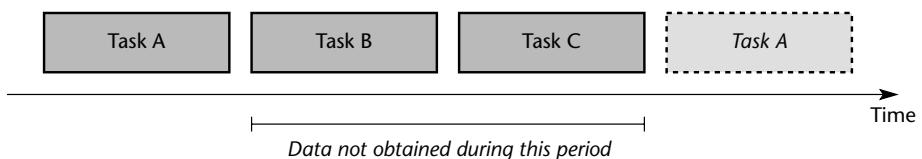


FIGURE 29.5 Running three consecutive tasks on a single processor can lead to loss of data

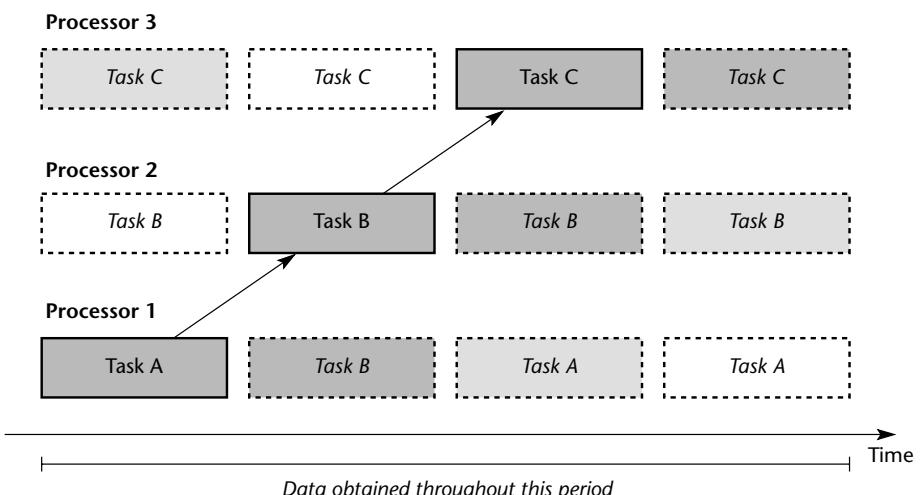


FIGURE 29.6 Running three consecutive tasks on a multiprocessor architecture can avoid loss of data

In Figure 29.6, we ‘domino schedule’ (or pipeline) Task A, Task B and Task C. At each interval of time, Task A processes new data and Task B processes the results of the previous Task A run.

In this case, we refer to Task B and Task C as *domino tasks*.

Solution

The key factor involved in determining whether a domino task architecture is appropriate for your application is the worst case execution time for the various tasks.

For example, consider Figure 29.6. For maximum efficiency, Task A, Task B and Task C must be of similar duration. Where this condition is not satisfied the ‘domino’ architecture is inappropriate and the gains in performance obtained through the use of the additional processor will be greatly reduced.

Where we have tasks of different duration, we can still obtain benefits from the use of multiple processors, but we need to use a different software architecture: see **LONG TASK** [page 716] for details.

Note that, if we require complete data coverage, Task A must have a duration greater than or equal to the duration of Task B and Task B must have a duration greater than or equal to that of Task C. If these conditions are not met, we have three main options:

- 1 Use a faster processor to reduce the execution time of Task B and / or Task C.
- 2 Split one or more of the tasks in such a way that they run as two *successive* (shorter) tasks; for example, split Task B into Task B1 and Task B2 and run each, successively, on two processors.
- 3 Split one or more of the tasks in such a way that they run as two *parallel* (shorter) tasks; for example, split Task B into Task B1 and Task B2 and run each, consecutively, on two processors.

Hardware resource implications

This pattern requires the use of at least two microcontrollers and associated hardware (for example, reset, oscillator, memory and power supplies).

Reliability and safety implications

The correct operation of **DOMINO TASK** depends on all the microcontrollers involved operating normally. This requirement can, under some circumstances, reduce the overall reliability of your system: see page 550 for a discussion of this issue.

Portability

This general design pattern may be applied with any microcontroller or microprocessor family.

Overall strengths and weaknesses

- ☺ An effective way of ‘pipelining’ tasks.
- ☹ Like any multiprocessor design, it must be used with care if overall system reliability is not to be compromised.

Related patterns and alternative solutions

See **HYBRID SCHEDULER** [page 333].

See **LONG TASK** [page 716].

See **DATA UNION** [page 720].

Example: Condition monitoring and control

Use of **DOMINO TASK** is particularly common in condition monitoring and control applications, of the type discussed in Part G.

For example, various condition monitoring systems we have worked on have taken the form illustrated in Figure 29.7.

These applications have three main phases:

- A data-acquisition and pre-processing phase, typically involving interaction with an ADC.
- A time-frequency transform phase (typically involving a fast Fourier transform: FFT).
- A classification phase, possibly involving a neural network.

In each case, the phases have a similar duration. The domino task architecture is therefore highly appropriate.

See Li *et al.* (1999, 2000) and Parikh *et al.* (2001) for further details of this type of application.

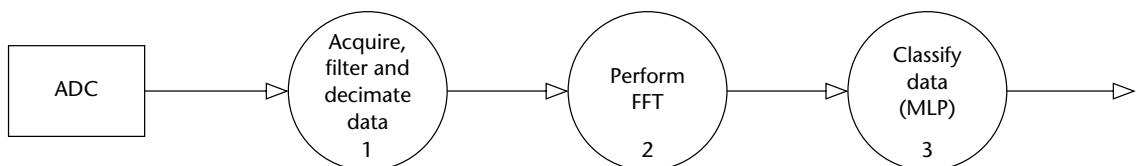


FIGURE 29.7 The architecture of a typical condition-monitoring application

Further reading

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. 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.

Monitoring and control components

The penultimate group of patterns presented in this book are intended for use in data acquisition, condition monitoring and control systems. As in previous parts, we are concerned with techniques which are appropriate for use in systems employing a time-triggered software architecture.

We begin in Chapter 30 by considering the counting of pulses. These techniques are in widespread use in many industrial applications and throughout the automotive industry. **HARDWARE PULSE COUNT** [page 728] and **SOFTWARE PULSE COUNT** [page 736] present, respectively, hardware-based and software-only techniques for pulse counting.

In Chapter 31, we address pulse-rate modulation: that is, the generation of square-wave signals of a specified frequency. Again, two patterns are presented: **HARDWARE PRM** [page 742] and **SOFTWARE PRM** [page 748].

In Chapter 32 we consider how we can use an 8051 microcontroller to measure analogue voltage or current signals. These may, for example, represent the battery voltage in a charger application: more generally, such signals can come from an enormous range of sensors, such as potentiometers or temperature probes. The patterns **ONE-SHOT ADC** [page 757] and **SEQUENTIAL ADC** [page 782s] discuss how to make effective use of on-chip and off-chip analogue-to-digital converters (ADCs). Also in this chapter, the patterns **ADC PRE-AMP** [page 777] and **A-A FILTER** [page 794] consider important pre-processing stages that may be required before an analogue signal is measured. Finally, the pattern **CURRENT SENSOR** [page 802] considers how we can use analogue measurements for current-sensing applications, such as detecting blown bulbs or stalled DC motors.

In Chapter 33 we explore how analogue outputs from the microcontroller – generated in the form of pulse-width modulated signals – can be used in applications such as setting the speed of motors or controlling the brightness of bulbs. Again, both hardware-

and software-based techniques are considered, in the patterns **HARDWARE PWM** [page 808] and **SOFTWARE PWM** [page 831]. We also consider the post-processing that may be required to filter PWM-based signals, through the pattern **PWM SMOOTHER** [page 818] and the creation of a high-frequency PWM output without using specialized hardware in the pattern **3-LEVEL PWM** [page 822].

In Chapter 34 we consider how analogue outputs from the microcontroller may be generated using digital-to-analogue converter (DAC) hardware (see **DAC OUTPUT** [page 841]). We consider the post-processing that may be required to filter and / or amplify DAC-derived signals, through the patterns **DAC SMOOTHER** [page 853] and **DAC DRIVER** [page 857].

Finally, in Chapter 35, we turn our attention to proportional-integral-differential (PID) control. PID is both simple and effective: as a consequence it is the most widely used control algorithm. The focus in this chapter is on techniques for designing and implementing PID controllers for use in embedded, time-triggered applications.

chapter 30

Pulse-rate sensing

Introduction

Suppose we wish to measure the speed of a rotating shaft. This could be part, for example, of an automotive or industrial application.

An effective way of measuring the speed is to attach an optically or magnetically based rotary encoder to the shaft (Figure 30.1), and to count the number of pulses that occur over a fixed period of time (say 100 ms or 1 second). From the count, and having details of the rotary encoder, we can calculate the average speed of rotation.

In this chapter we consider how pulses may be counted. The technique we discuss may be used not just for the measurement of rotational speed but also, for example, in liquid flow-rate measurement and in the sensing of vibration.

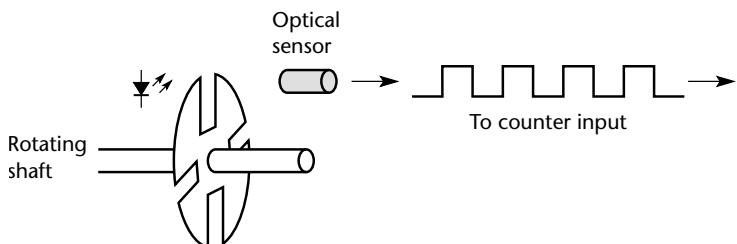


FIGURE 30.1 Counting pulses from an optical encoder in order to measure the speed of rotation of a shaft

HARDWARE PULSE COUNT

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you count rectangular pulses arriving from an external peripheral?

Background

Solution

Consider the train of pulses obtained from the rotary encoder discussed in the introduction to this chapter.

Where we have sufficient hardware resources available, we can often determine the average pulse rate from this stream largely without software intervention. Timer 0 or Timer 1¹ will count pulses (more specifically, the falling edges of pulses) on external pins, without generating interrupts and without interfering with any other processing.

The key to **HARDWARE PULSE COUNT** is the setting of the TMOD SFR (Table 30.1).

TABLE 30.1 The TMOD SFR, used to control Timer 0 and Timer 1

Bit	7 (MSB)	6	5	4	3	2	1	0 (LSB)
Name	Gate	C/T	M1	M0	Gate	C/T	M1	M0
<i>Timer 1</i>					<i>Timer 0</i>			

In Table 30.1, most of the features of TMOD have been discussed previously (see Chapter 11). Here we are particularly concerned with the ‘counter’ bits (6 and 2). If either of these bits is set to 0, then the corresponding timer is set to counter mode.

1. In many cases, as we saw in Chapter 3, modern ‘8051’ family devices are based on the slightly later 8052 architecture: such devices include an extra, more powerful timer (Timer 2). Timer 2 can also be used to count pulses; however, as we discussed in Chapters 13 and 14, we prefer to use this timer – where available – as a source of scheduler ticks. We will therefore not discuss the use of Timer 2 for pulse counting here.

For example:

```
// Timer 0 used as 16-bit timer, counting pulses
// (falling edges) on Pin 3.4 (T0 pin)
TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
TMOD |= 0x05; // Set required T0 bits (T1 left unchanged)
```

In this case, the Timer 0 registers (TL0 and TH0) are incremented in response to a transition (1-to-0) at its corresponding external input pin (Figure 30.2). The pin is sampled every machine cycle. When the samples show a high in one cycle and a low in the next cycle, the count is incremented. The new count value then appears in the register in the cycle following the one in which the transition was detected.

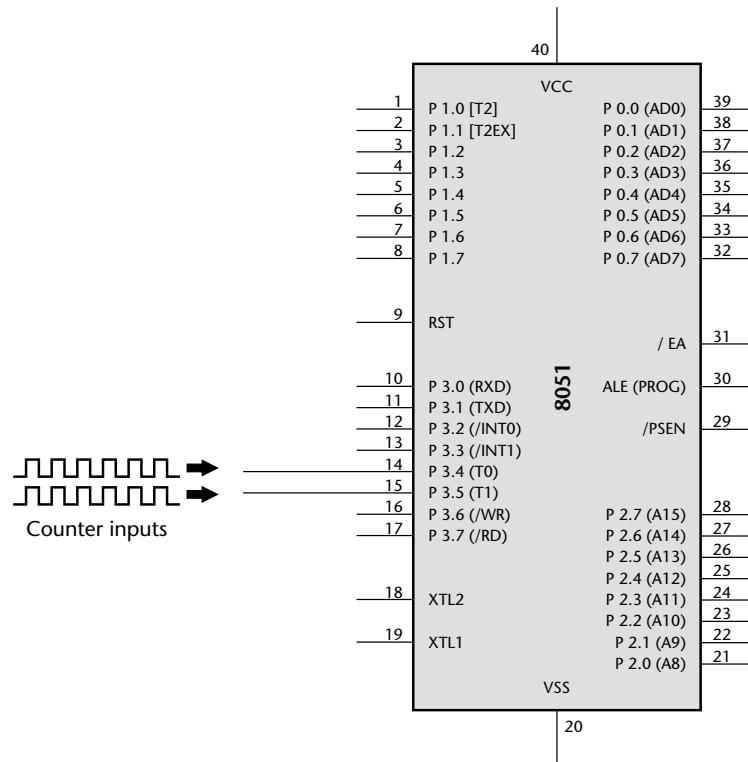


FIGURE 30.2 Reading pulse counts

There are no restrictions on the duty cycle of the waveform that we can monitor using this approach, but – to ensure accurate measurement – the ‘high’ and ‘low’ levels should be held for at least one full machine cycle (12 oscillator cycles in an original 8051; 6, 4 or 1 oscillator cycle(s) in some more recent variants). As a result, it takes at

least two machine cycles (24, 12, 8 or 2 oscillator cycles) to detect a 1-to-0 transition. This determines the maximum pulse rate that you can measure: for example, on a basic 12 MHz / 12 oscillator cycle 8051, the maximum pulse rate (square wave) that we can measure has a period of 24 oscillator cycles (2 µs), and a frequency of 500 kHz.

We can use this information efficiently to measure the rotational speed in the previous example by creating a task that operates as follows:

- 1 Read the current hardware pulse count.
- 2 Store the result in a global variable.
- 3 Reset the count to 0.

We need to schedule this task for repeated execution (say every 100 ms).

We give a code example that illustrates this approach in a following example.

Hardware resource implications

HARDWARE PULSE COUNT requires exclusive access to a timer. In many cases, Timer 2 will be used for the scheduler and Timer 1 may be used for baud-rate generation. This will only leave Timer 0 for this purpose. Where this is not available (or you need to measure multiple pulse trains) you will need to consider software-based techniques: see **SOFTWARE PULSE COUNT** [page 736] or the use of an additional microcontroller (see Part F).

Reliability and safety implications

There are no specific reliability and safety implications, provided you do not attempt to count pulses beyond the limits of your hardware (see 'Solution' for details).

Portability

This pattern uses only core 8051 features: there are no specific portability implications.

Overall strengths and weaknesses

- 😊 **HARDWARE PULSE COUNT** imposes a minimal software overhead.
- 😢 **HARDWARE PULSE COUNT** requires exclusive access to a timer for each pulse train you wish to count.

Related patterns and alternative solutions

There are two main alternatives to **HARDWARE PULSE COUNT**.

The closest match is **SOFTWARE PULSE COUNT** [page 736]: this performs in a similar way, but does not require the use of timer hardware. Inevitably, this imposes a larger software load than the present pattern.

Alternatively, you can use external hardware to convert the pulse train to an analog voltage, which may then be measured using an on-chip or external analogue-to-digital converter (ADC). Various ‘frequency-to-voltage’ conversion chips are available to assist with this process, such as the National Semiconductor² LM2907 and LM2917. The use of an ADC is discussed in **ONE-SHOT ADC** [page 757].

Note that the use of an ADC (and associated hardware) will not allow you to count, precisely, the number of pulses that occur: this approach would probably not be appropriate, for example, for counting visitors passing through a turnstile. However, if you require a measure of the average pulse count – as in some forms of speed measurement – then this solution may be adequate. Note, however, that there may be significant hardware costs involved.

Example: Generic pulse-count (hardware) library

A simple pulse-count (hardware) library is presented in Listings 30.1 to 30.3.

This library counts pulses on Pin 3.4 and displays the results in the form of a bargraph on Port 1.

```
/* -----
Port.H (v1.00)

-----
'Port Header' (see Chap 10) for the project PC_Hard (see Chap 30)

----- */
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P2

#endif

// ----- Bargraph.C -----
-----
```

2. www.national.com

```

// Connect LED from +5V (etc) to these pins, via appropriate resistor
// [see Chapter 7 for details]
// The 8 port pins may be distributed over several ports if required

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 -----
// Counts pulses on P3.4 (using Timer 0)

/*-----*
*----- END OF FILE -----*
*-----*/

```

Listing 30.1 Part of a generic pulse-count (hardware) example

```

/*-----*
*----- Main.c (v1.00)
-----*

Demo application for Hardware Pulse Count (Chapter 30).

Required linker options (see Chapter 14 for details):
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();           // Set up the scheduler
    PC_HARD_Init_T0();       // Prepare to count pulses
    BARGRAPH_Init();         // Prepare a bargraph-type display

    // Add a 'pulse count' task
    SCH_Add_Task(PC_HARD_Get_Count_T0, 1000, 1000);
    // Simply display the count here (bargraph display)
    SCH_Add_Task(BARGRAPH_Update, 1200, 1000);

    // All tasks added: start running the scheduler
    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

/*-----*
 *----- END OF FILE -----*
 */
```

Listing 30.2 Part of a generic pulse-count (hardware) example

```

/*-----*
 *----- PulCnt_H.C (v1.00)
 *-----*/

Hardware pulse count library (see Chapter 30).

/*-----*/
#include "Main.h"
#include "Port.h"

#include "Bargraph.h"
#include "PulCnt_H.h"

// ----- Public variable declarations -----
// Global count variable - stores the latest count value
extern tBargraph Count_G;

/*-----*/

```

```

PC_HARD_Init_T0()

    Prepare for 'Hardware Pulse Count' using Timer 0.
    -----
void PC_HARD_Init_T0(void)
{
    // Timer 0 used as 16-bit timer, counting pulses
    // (falling edges) on Pin 3.4 (T0 pin)
    TMOD &= 0xF0;          // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x05;          // Set required T0 bits (T1 left unchanged)
    TH0 = 0; TL0 = 0;       // Set timer count to 0
    Count_G = 0;            // Set global count to 0
    TR0 = 1;                // Start the counter
}

PC_HARD_Get_Count_T0()

    Schedule this function at regular intervals.

    Remember: max count is 65536 (16-bit counter)
    - it is your responsibility to ensure this count
    is not exceeded. Choose an appropriate schedule
    interval and allow a margin for error.

    For high-frequency pulses, you need to take account of
    the fact that the count is stopped for a (very brief) period,
    to read the counter.

    Note: the delay before the first count is taken should
    generally be the same as the inter-count interval,
    to ensure that the first count is as accurate as possible.

    For example, this is OK:

        Sch_Add_Task(PC_HARD_Get_Count_T0, 1000, 1000);

    While this will give a very low first count:

        Sch_Add_Task(PC_HARD_Get_Count_T0, 10, 1000);

    -----
void PC_HARD_Get_Count_T0(void)
{
    TR0 = 0; // Stop counter

    Count_G = (TH0 << 8) + TL0; // Read count
    TH0 = 0; TL0 = 0;           // Reset count
}

```

```
if (TF0 == 1)
{
    // Timer has overflowed
    // - pulse frequency too high
    // - or schedule rate too low

    // We code this error as a 'max count'
    // - could also set a global error flag, if required
    Count_G = 65536;
    TF0 = 0;
}
TR0 = 1; // Restart counter
}

/* -----
--- END OF FILE ---
----- */
```

Listing 30.3 Part of a generic pulse-count (hardware) example

Further reading

SOFTWARE PULSE COUNT

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you count pulses arriving from an external peripheral?

Background

See **HARDWARE PULSE COUNT** [page 582] for background details.

Solution

HARDWARE PULSE COUNT requires exclusive access to a timer.

Where this timer is not available and comparatively low pulse rates are to be measured, software-only pulse counting is possible.

Software-based pulse counting involves a periodic task that keeps track of the state of one or more input pins. When a pin changes state in the interval between task invocations, we increment the corresponding count by 1. The main drawback is the maximum pulse rate: with a 1 ms scheduler, the maximum pulse rate we can measure is 500 Hz: this is around 1,000 times less than can be measured using **HARDWARE PULSE COUNT** [page 730].

Hardware resource implications

This pattern uses no on-chip hardware components.

Reliability and safety implications

There are no specific reliability and safety implications.

Portability

This code has no hardware requirements and may be readily ported to a different microcontroller environment, provided that the new environment also uses a scheduler operating at the same tick rate.

Overall strengths and weaknesses

😊 **SOFTWARE PULSE COUNT** does not require a timer.

-  The main drawback is the maximum pulse rate. With a 1 ms scheduler, the maximum pulse rate we can measure is 500 Hz: this is around 1,000 times less than can be measured using hardware.

Related patterns and alternative solutions

The main alternative to SOFTWARE PULSE COUNT is HARDWARE PULSE COUNT [page 728].

Example: Generic pulse-count (software) library

A simple pulse-count (software) library is presented in Listings 30.4 to 30.6.

This library counts pulses on Pin 3.0 and displays the results in the form of a bar-graph on Port 1.

```
/*-----*-
Port.H (v1.00)
-----*
'Port Header' (see Chap 10) for the project PC_Soft

-*-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P2

#endif

// ----- Bargraph.C -----
// Connect LED from +5V (etc) to these pins, via appropriate resistor
// [see Chapter 7 for details]
// The 8 port pins may be distributed over several ports if required
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;

/* -----
--- END OF FILE -----
*/
```

Listing 30.4 Part of a generic pulse-count (software) example

```
/* -----
Main.c (v1.00)

-----

Demo application for Software Pulse Count

Required linker options (see Chapter 14 for details):
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(); // Set up the scheduler
    PC_SOFT_Init(); // Prepare to count pulses
    BARGRAPH_Init(); // Prepare a bargraph-type display (P4)

    // TIMING IS IN TICKS (1 ms interval)

    // Add a 'pulse count poll' task
    // Every 40 milliseconds
    SCH_Add_Task(PC_SOFT_Poll_Count, 0, 20);

    // Add a 'pulse get count' task
    // Every 20 seconds
    SCH_Add_Task(PC_SOFT_Get_Count, 0, 20000);

    // Simply display the count here (bargraph display)
    // Max count is ~250
    SCH_Add_Task(BARGRAPH_Update, 50, 10000);
```

```

// All tasks added: start running the scheduler
SCH_Start();

while(1)
{
    SCH_Dispatch_Tasks();
}

/* -----
--- END OF FILE ---
*/
```

Listing 30.5 Part of a generic pulse-count (software) example

```

/* -----
----- PulCnt_S.C (v1.00)

-----
Software pulse count library (see Chapter 30).

*/
#include "Main.h"
#include "Port.h"
#include "Bargraph.h"
#include "PulCnt_S.h"

// ----- Public variable definitions -----
// Stores the average count value
extern tBargraph Count_G;

// ----- Private constants -----
#define PULSE_HIGH (1)
#define PULSE_LOW (0)

// ----- Private variable definitions-----
// Stores the instantaneous count value
static tWord Count_local_G;

// The previous state of the pulse-count pin
static bit Previous_state_G;
/* -----
PC_SOFT_Init()
Prepare for software pulse counts.

*/

```

```

void PC_SOFT_Init(void)
{
    Count_local_G = 0;
    Count_G = 0;
}

/*-----*/
PC_SOFT_Poll_Count()

Using software to count falling edges on a specified pin
- T0 is *NOT* used here.

/*-----*/
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()

Schedule this function at regular intervals.

Copies 'polled' count to global variable.

/*-----*/
void PC_SOFT_Get_Count(void)
{
    Count_G = Count_local_G;
    Count_local_G = 0;
}

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 30.6 Part of a generic pulse-count (software) example

Further reading

chapter 31

Pulse-rate modulation

Introduction

In this chapter we consider how you can create pulse streams (sometimes called 'clock-outs') of different frequencies.

Such streams have various applications, from the driving of simple flashing lights and sirens (Figure 31.1), controlling the brightness of lamps, controlling the speed of DC motors, through to test circuits for pulse-rate sensors.

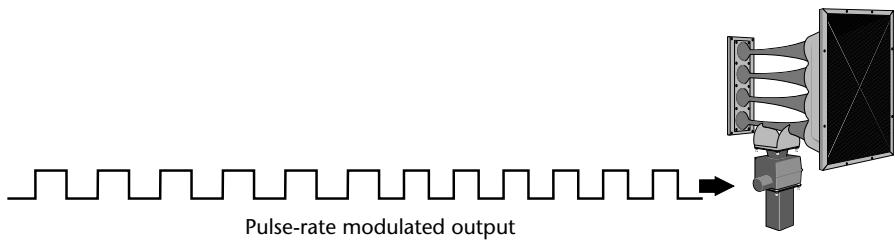


FIGURE 31.1 A pulse-rate modulated output used to control the frequency of a siren

HARDWARE PRM

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you create a pulse-rate modulated output (a ‘clock-out’) of variable frequency, while imposing a minimal software load?

Background

We consider some basic background material in this section.

Duty cycles

Pulse-rate modulation is carried out by setting a port pin to Logic 1 for a period (x) and then to Logic 0 for another period (y). We then repeat this process (Figure 31.2).

The duty cycle is defined as follows:

$$\text{Duty cycle (\%)} = \frac{x}{x + y} \times 100$$

In this pattern, we are concerned only with signals which have a 50% duty cycle: that is, where x and y are equal.

The pulse-rate frequency is, in these circumstances, given by:

$$\text{Frequency} = \frac{1}{2x}, \text{ where } x \text{ is in seconds.}$$

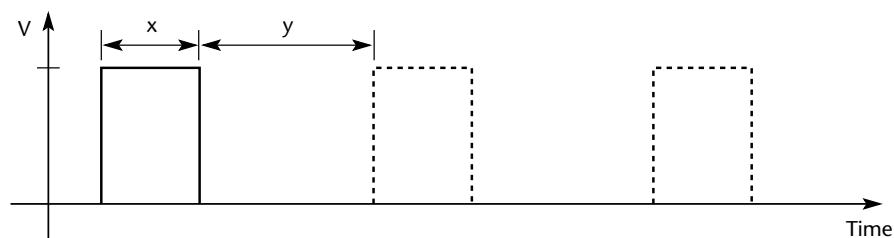


FIGURE 31.2 The underlying principles of pulse-rate modulation

Timer 2

As we saw in Chapter 3, the original 8052 and most modern ‘8051’ devices contain an additional timer, known as Timer 2. This pattern requires the use of Timer 2.

Refer to Chapter 3 for basic background on Timer 2 and to Chapter 11 for basic information on timers.

Solution

Using Timer 2 (see ‘Background’), Pin 1.0 can be programmed to output a 50% duty cycle clock.

To configure this timer³ as a clock generator, bit C/T2 (T2CON.1) must be cleared and bit T2OE (T2MOD.1) must be set. Bit TR2 (T2CON.2) starts and stops the timer. The clock-out frequency depends on the oscillator frequency and the reload value of Timer 2 capture registers (RCAP2H, RCAP2L), as shown in the following equation.

$$\text{Frequency}_{\text{pulse}} = \frac{\text{Frequency}_{\text{Oscillator}}}{4 \times (65536 - \text{RCAP2H}, \text{RCAP2L})}$$

For example, the minimum frequency with a 12 MHz crystal is:

$$\text{Frequency}_{\text{min}} = \frac{12000000}{4 \times (65536)} = 45.78 \text{ Hz}$$

Similarly, the maximum frequency is:

$$\text{Frequency}_{\text{max}} = \frac{12000000}{4} = 3.0 \text{ MHz}$$

In the clock-out mode, Timer 2 rollovers will not generate an interrupt.

This behaviour is similar to when Timer 2 is used as a baud-rate generator and it is possible to use Timer 2 as a baud-rate generator and a clock generator simultaneously. Note, however, that the baud rate and clock-out frequencies cannot be determined independently from one another since they both use RCAP2H and RCAP2L.

Hardware resource implications

This pattern requires the use of Timer 2. In single-processor applications, Timer 2 is often the most suitable timer to use to drive the scheduler itself; as a result, use of this technique may have an impact on other parts of the application.

Note that, if a two-processor solution is possible (using a UART- or interrupt-based scheduler, for example), then Timer 2 can be used to drive the scheduler in the Master

3. As we discussed in Chapters 3 and 11, different implementations of Timer 2 are used in various 8051 family members; the names of some of the registers can therefore vary. This description applies specifically to the Atmel 89x52 / 89x53 devices but may be applied with minor modifications to other family members.

node, leaving Timer 2 in the Slave node free for use as a PRM generator: see Part F for details of various multiprocessor solutions that may be appropriate here.

Reliability and safety implications

This technique is very reliable.

Portability

This technique may only be used with 8052-based devices: that is, those with an implementation of Timer 2 available. While most modern ‘8051s’ have such a timer, some popular devices, such as the Atmel Small 8051s, do not.

Overall strengths and weaknesses

- ☺ A simple, effective technique for creating PRM signals over a very wide frequency range.
- ☺ Imposes no measurable software or CPU load.
- ☺ Requires exclusive use of Timer 2.

Related patterns and alternative solutions

The most directly comparable pattern is **SOFTWARE PRM** [page 748]. In addition, **SOFTWARE PWM** [page 831], **HARDWARE PWM** [page 808] and particularly **3-LEVEL PWM** [page 822] can form the basis of alternative solutions under some circumstances.

As we noted in ‘Hardware resource implications’, it may be appropriate to use this technique in a Slave node in a multiprocessor application. Refer to the various patterns in Part F for further information on this topic.

Example: Hardware PRM on the ‘8052’

An example of software to control hardware-based pulse-rate modulation is given in this section.

In Listings 31.1 and 31.2, we use a 16-bit (unsigned) integer variable (`PRM_reload_G`) to control the pulse frequency. The resulting frequency, as discussed earlier, is as follows:

$$\text{Frequency}_{\text{pulse}} = \frac{\text{Frequency}_{\text{Oscillator}}}{4 \times (65536 - \text{RCAP2H}, \text{RCAP2L})}$$

Thus, with a 12 MHz oscillator, the pulse rate will increase slowly from 45Hz to 3MHz (approximately), repeatedly.

```

/* -----
Main.c (v1.00)

-----

Demo of hardware PRM.

----- */

#include "Main.h"
#include "PRM_Hard.h"

// ----- Public variable declarations -----
extern tWord PRM_reload_G;

/* ..... */
/* .. */

void main(void)
{
    tLong Count = 0;

    PRM_Hardware_Init();

    while(1)
    {
        if (++Count > 10000UL)
        {
            // Slowly change the PRM frequency
            PRM_reload_G++;

            PRM_Hardware_Update();

            Count = 0;
        }
    }
}

/* -----
--- END OF FILE ---
*/

```

Listing 31.1 Part of a generic pulse-rate modulation (hardware-based) example

```

/* -----
PRM_Hard.C (v1.00)

```

```
-----  
      Simple library demonstrating hardware (T2) pulse-rate modulation.  
      See Chapter 31 for details.  
- *----- */  
  
#include "Main.h"  
  
// ----- Public variable definitions -----  
  
tWord PRM_reload_G = 0;  
  
/*----- */  
  
PRM_Hardware_Init()  
  
    Start PRM.  
  
- *----- */  
void PRM_Hardware_Init(void)  
{  
    T2CON &= 0xFD;    // Clear *only* C /T2 bit  
    T2MOD |= 0x02;    // Set T20E bit (omit in basic 8052 clone)  
  
    // Start at lowest frequency (~45Hz with 12MHz xtal)  
    TL2      = 0x00;    // Timer 2 low byte  
    TH2      = 0x00;    // Timer 2 high byte  
    RCAP2L   = 0x00;    // Timer 2 reload capture register, low byte  
    RCAP2H   = 0x00;    // Timer 2 reload capture register, high byte  
  
    ET2      = 0; // No interrupt.  
  
    TR2      = 1; // Start Timer 2  
}  
  
/*----- */  
  
PRM_Hardware_Update()  
  
Call this function only when you need to change the pulse rate.  
  
See text for details of resulting PRM frequency.  
  
- *----- */  
void PRM_Hardware_Update(void)  
{  
    TR2 = 0;  
  
    TL2      = PRM_reload_G % 256;  
    RCAP2L   = TL2;  
    TH2      = PRM_reload_G / 256;
```

```
RCAP2H = TH2;  
TR2 = 1;  
}  
  
/* ----- * -  
----- END OF FILE -----  
-* ----- * /
```

Listing 31.2 Part of a generic pulse-rate modulation (hardware-based) example

Further reading

SOFTWARE PRM

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you create a square- or rectangular-wave output pulse with particular pulse-width and frequency characteristics without using on-chip hardware?

Background

Refer to **HARDWARE PRM** [page 742] for basic background information on pulse-rate modulation.

Solution

Creating an output that is pulse-rate modulated using a scheduler is easy to do: in fact, all our introductory examples did just this. For example, consider Listing 31.3, first presented in Chapter 14.

```
void main(void)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Prepare for the 'Flash_LED' task
    LED_Flash_Init();

    // Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
    // - timings are in ticks (1 ms tick interval)
    // (Max interval / delay is 65535 ticks)
    SCH_Add_Task(LED_Flash_Update, 0, 1000);

    // Start the scheduler
    SCH_Start();

    while(1)
    {
```

```

        SCH_Dispatch_Tasks();
    }
}
}

```

Listing 31.3 A simple form of pulse-rate modulation (software based)

The ‘flash LED’ task itself may be implemented as shown in Listing 31.4.

```

void LED_Flash_Update(void)
{
    // Change the LED from OFF to ON (or vice versa)
    if (LED_state_G == 1)
    {
        LED_state_G = 0;
        LED_pin = 0;
    }
    else
    {
        LED_state_G = 1;
        LED_pin = 1;
    }
}

```

Listing 31.4 A ‘flash LED’ task

This code flashes an LED at a 0.5Hz rate: it could, of course, drive various other devices, if we use an appropriate hardware interface.

The same technique may be readily adapted to allow us to perform low-frequency software PRM, at variable frequencies: Listing 31.5 illustrates this.

```

void PRM_Soft_Update(void)
{
    // Increment the 'position' variable
    if (++PRM_position_G >= PRM_period_G)
    {
        PRM_position_G = 0;

        PRM_period_G = PRM_period_new_G;

        PRM_pin = 0;

        return;
    }

    // Generate the PRM output
    if (PRM_position_G < (PRM_period_G / 2))

```

```

    {
        PRM_pin = 1;
    }
else
{
    PRM_pin = 0;
}
}

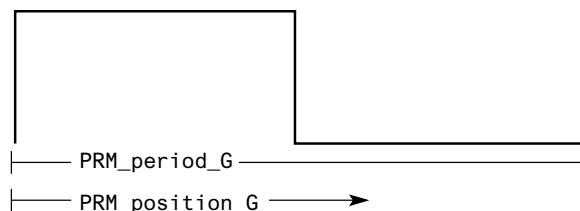
```

Listing 31.5 Implementing variable-frequency software PRM

The key to Listing 31.5 is the use of the variables `PRM_position_G`, `PRM_period_G` and `PRM_period_new_G`:

- **PRM_period_G** is the current PRM period. It is not varied by the user. Note that if the update function is scheduled every millisecond, this period is in milliseconds.
 - **PRM_period_new_G** is the next PRM period. This period may be varied by the user, as required. Note that the ‘new’ value is only copied to **PRM_period_G** at the end of a PRM cycle, to avoid noise. Again, the units are milliseconds, if the update function is scheduled every millisecond.
 - **PRM_position_G** is the current position in the PRM cycle. This is incremented by the update function. Again, the units are milliseconds if the same conditions apply.

The link between PRM_period_G, PRM_position_G and the PRM output is illustrated in Figure 31.3.



```
if (PRM_position_G < (PRM_period_G / 2))
{
    PRM_pin = 1;
}
else
{
    PRM_pin = 0;
}
```

FIGURE 31.3 Implementing pulse-rate modulation in software

Hardware resource implications

This pattern imposes a minimal software or hardware load.

Reliability and safety implications

This approach is reliable.

Portability

This software is based entirely on the core scheduler code: as a result, like the scheduler itself, it is inherently portable.

Overall strengths and weaknesses

- ☺ Simple and easy to use, with minimal CPU or memory overheads.
- ☺ Can produce very low frequency signals with ease.
- ☹ The maximum frequency is limited by the scheduler itself: if the scheduler tick period is T (seconds), then the maximum pulse rate is 1/2T (Hz).

Related patterns and alternative solutions

See **HARDWARE PRM** [page 742] for an alternative solution.

Example: Software PRM on the 8051 (generic code)

A simple example of a complete software PRM library is given in Listings 31.6 to 31.8.

```
/*-----*  
Port.H (v1.00)  
-----*  
  
'Port Header' (see Chap 10) for the project PRM_Soft  
-----*/  
// ----- Sch51.C -----  
// Comment this line out if error reporting is NOT required  
#define SCH_REPORT_ERRORS  
  
#ifdef SCH_REPORT_ERRORS  
// The port on which error codes will be displayed  
// ONLY USED IF ERRORS ARE REPORTED  
#define Error_port P2
```

```
#endif

// ----- PRM_Soft.c -----
// The PRM output pin
sbit PRM_pin = P2^0;

/*-----*
*----- END OF FILE -----
*-----*/
```

Listing 31.6 Part of an example demonstrating software-based pulse-rate modulation

```
/*-----*
*----- 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();

    // Call every millisecond to update PRM output
    SCH_Add_Task(PRM_Soft_Update, 10, 1);

    // Call every minute to change PRM control value
    SCH_Add_Task(PRM_Soft_Test, 0, 60000);

    SCH_Start();

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

```
/*-----*  
---- END OF FILE -----  
*-----*/
```

Listing 31.7 Part of an example demonstrating software-based pulse-rate modulation

```
/*-----*  
PRM_Soft.c (v1.00)  
-----*  
  
Simple Software PRM library.  
*-----*/  
  
#include "Main.h"  
#include "Port.h"  
  
#include "2_01_12g.h"  
#include "PRM_Soft.h"  
  
// ----- Public variable definitions -----  
  
// Set this variable to the required PRM value  
tWord PRM_period_new_G;  
  
// ----- Private variable definitions-----  
  
// The PRM counter  
static tWord PRM_position_G;  
static tByte PRM_period_G;  
  
/*-----*  
PRM_Soft_Init()  
  
Prepare for software PRM.  
*-----*/  
void PRM_Soft_Init(void)  
{  
    // Init the main variable  
    PRM_period_G = 2;  
    PRM_period_new_G = 2;  
  
    PRM_position_G = 0;  
}  
  
/*-----*/
```

```
PRM_Soft_Update()
```

Update the software PRM output.

We have three key variables (see text for details):

1. PRM_period_G is the PRM period
(units are milliseconds, if we schedule once / ms)
2. PRM_period_new_G is the new PRM period, set by the user
(The 'new' value is copied to PRM-period only at the
end of a cycle, to avoid noise)
(units are milliseconds, if we schedule once / ms)
3. PRM_position_G is the current position in the PRM cycle
(units are milliseconds, if we schedule once / ms)

```
- *-----*/  
void PRM_Soft_Update(void)  
{  
    // Increment the 'position' variable  
    if (++PRM_position_G >= PRM_period_G)  
    {  
        PRM_position_G = 0;  
  
        PRM_period_G = PRM_period_new_G;  
  
        PRM_pin = 0;  
  
        return;  
    }  
  
    // Generate the PRM output  
    if (PRM_position_G < (PRM_period_G / 2))  
    {  
        PRM_pin = 1;  
    }  
    else  
    {  
        PRM_pin = 0;  
    }  
}  
/*-----*-
```

```
PRM_Soft_Test()
```

To test the PRM library, this function is called once every minute, to change the PRM output setting.

```
-----*/  
void PRM_Soft_Test(void)  
{  
    PRM_period_new_G += 2;  
  
    if (PRM_period_new_G >= 60000)  
    {  
        PRM_period_new_G = 2;  
    }  
}  
  
/* ----- END OF FILE ----- */  
-----*/
```

Listing 31.8 Part of an example demonstrating software-based pulse-rate modulation

Further reading

Using analogue-to-digital converters (ADCs)

Introduction

The recording of analog signals is an important part of many condition monitoring, data acquisition and control applications.

We will consider how to read analog values using an 8051 microcontroller in this chapter. In doing so, we will present the following patterns:

- **ONE-SHOT ADC** [page 757] addresses the problem of measuring an analogue voltage signal at irregular (or infrequent) intervals, using a microcontroller
- **ADC PRE-AMP** [page 777] addresses the problem of amplifying an analogue signal, in order to convert it to a range suitable for subsequent analogue-to-digital conversion.
- **SEQUENTIAL ADC** [page 782] addresses the problem of recording a sequence of analogue samples using a microcontroller.
- **A-A FILTER** [page 794] addresses the problem of filtering an analogue signal to remove high-frequency components.
- **CURRENT SENSOR** [page 802] addresses the problem of monitoring the current flowing through a DC load.

ONE-SHOT ADC

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you measure an analogue voltage or current signal at irregular (or infrequent) intervals?

Background

We consider some basic background material in this section.

Measuring voltages

Consider the analogue input from the potentiometer shown in Figure 32.1.

The resulting analogue voltage (in the range, here, of 0–5V) can be used as part of a user interface, if we have an on- or off-chip analogue-to-digital converter (ADC) available; as we will see, such devices are common.⁴

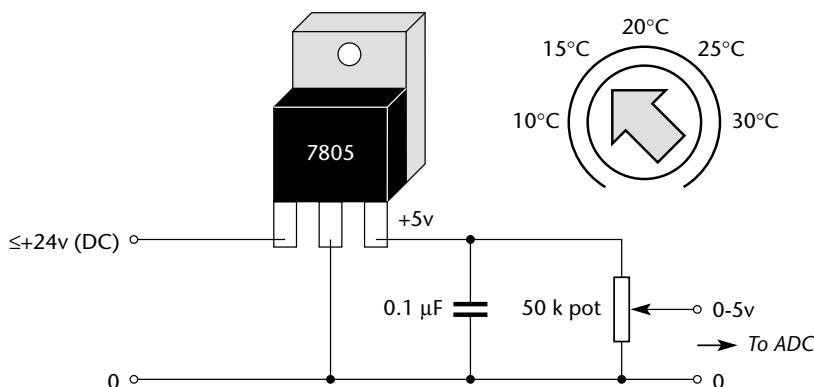


FIGURE 32.1 Using a potentiometer as part of a user interface

4. Please note that, although in widespread use, there may be better ways of creating such a user interface: refer to 'Related patterns and alternative solutions' (page 762).

This general approach is more widely applied. For example, consider Figure 32.2. Here we use three potentiometers (and an appropriate three-channel ADC) to measure angles in a mechanical excavator. By measuring the angles at points A, B and C, we can determine the position (specifically, the depth) of the shovel (X).

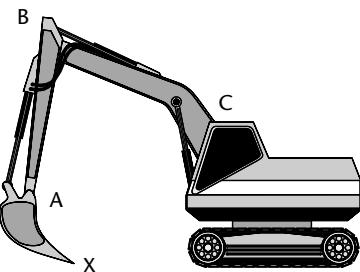


FIGURE 32.2 Using potentiometers to measure angles in a mechanical excavator

[Note: By measuring the angles at points A, B and C, we can determine the position (specifically, the depth) of the shovel (X).]

Measuring currents

In these simple earlier examples, we illustrated how analogue voltage signals might be a useful source of information in embedded applications. However, particularly in industrial applications, it can be helpful to be able to measure analogue *current* signals.

To see why current signals can be useful, consider Figure 32.3.

Figure 32.3 shows a sensor which, we assume, is connected to a long stretch of wire (with resistance R_{wire}) and, thereby, to the analogue (voltage) input of a microcontroller. The sensor, we will assume, measures temperature and generates an output of 5V to represent 100°C and 1 V to represent 0°C. We will further assume that only positive temperatures are possible and that, if the sensor or the wiring is broken, the voltage measured will be 0V, indicating an error.

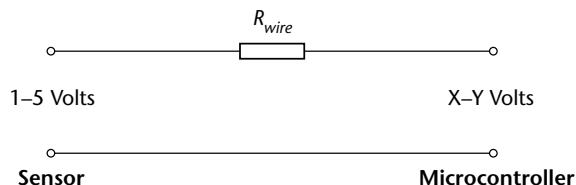


FIGURE 32.3 The problems with voltage-output sensors

There are two problems with this arrangement:

- 1 The voltage at the microcontroller will be less – possibly much less – than the voltage at the sensor. We will therefore probably need to fine-tune the sensor code to take into account the voltage drop in the wiring.
- 2 The wire resistance (and, hence, the voltage drop) will be temperature dependent, making it very difficult to obtain accurate readings even after fine-tuning.

One good solution to this problem is to digitize the analogue readings in the sensor and to transmit a digital representation of the voltage signal to the main microcontroller; the techniques discussed in Part F can be used to implement this solution.

Another solution, which is very common in the process industry, is to use a varying current (e.g. 4–20 mA) rather than a varying voltage to encode the sensor information (Figure 32.4).

The current-source sensor works well, even over long distances, even where the wiring resistance varies with temperature, since the sensor can adjust its output voltage, as required, to keep the current at the specified level. Note also that, at the microcontroller, we may simply convert the current signal back to a voltage signal by using a fixed resistor.

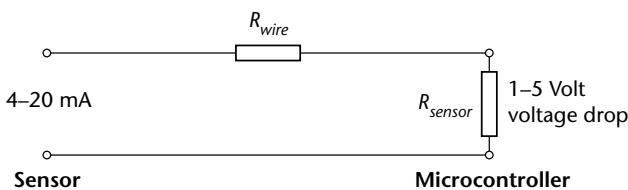


FIGURE 32.4 A schematic representation of a current-mode sensor

Solution

We will briefly consider here some of the hardware options that are available to allow the measurement of analogue voltage or current signals using a microcontroller.

Specifically, we will consider four options:

- Using a microcontroller with on-chip (voltage-mode) ADC
- Using an external serial (voltage-mode) ADC
- Using an external parallel (voltage-mode) ADC
- Using a current-mode ADC

We present a number of complete software libraries in the examples that follow.

Using a microcontroller with on-chip (voltage-mode) ADC

Many members of the 8051 family contain on-board ADCs. In general, use of an internal device will result in increased reliability, since both hardware and software

complexity will often be lower. In addition, the ‘internal’ solution will usually be physically smaller and have a lower system cost.

Here are some examples of the available ADC components provided on two 8051 devices.

Infineon c515c

From the Infineon⁵ c515c data sheet:

The c515c includes a high performance / high speed 10-bit A/D-converter (ADC) with 8 channels. It operates with a successive approximation technique and uses self calibration mechanisms for reduction and compensation of offset and linearity errors.

The A/D converter provides the following features:

- 8 multiplexed input channels (port 6), which can also be used as digital inputs
- 10-bit resolution
- Single or continuous conversion mode
- Internal or external start-of-conversion trigger capability
- Uses successive approximation conversion technique via a capacitor array
- Built-in hidden calibration of offset and linearity errors

Analog Devices AD μ C812

From the Analog Devices⁶ AD μ C812 data sheet:

The ADC conversion block incorporates a fast, multi-channel, 12-bit, single supply A/D converter. This block provides the user with multi-channel mux, track/hold, on-chip reference, calibration features and A/D converter. All components in this block are easily configured via the SFR interface from the core MCU.

The A/D converter section in this block consists of a conventional successive-approximation converter based around a capacitor DAC. The converter accepts an analogue input range of 0 to +VREF. A high precision, low drift 2.5V reference is provided on-chip. The internal reference may be overdriven via the external VREF pin. This external reference can be in the range 2.3V to AVDD.

Single step or continuous conversion modes can be initiated in software or alternatively by applying a convert signal to the an external pin. Timer 2 can also be configured to generate a repetitive trigger for ADC conversions. The ADC may also be configured to operate in a DMA Mode whereby the ADC block continuously converts and captures samples without any interaction from the MCU core.

The ADC core contains self-calibration and system calibration options to ensure accurate operation over time and temperature. A voltage output from an On-Chip bandgap reference proportional to absolute temperature can also be routed through the front-end ADC multiplexor facilitating a temperature sensor implementation.

5. www.infineon.com

6. www.analog.com

Using an external parallel (voltage-mode) ADC

The ‘traditional’ alternative to an on-chip ADC is a parallel ADC. In general, parallel ADCs have the following strengths and weaknesses:

- ☺ They can provide fast data transfers.
- ☺ They tend to be inexpensive.
- ☺ They require a very simple software framework.
- ☹ They tend to require a large number of port pins. In the case of a 16-bit conversion, the external ADC will require 16 pins for the data transfer, plus between one and three pins to control the data transfers.
- ☹ The wiring complexity can be a source of reliability problems in some environments.

Examples of the use of a parallel ADC follow.

Using an external serial (voltage-mode) ADC

Many more recent ADCs have a serial interface. In general, serial ADCs have the following strengths and weaknesses:

- ☺ They require a small number of port pins (between two and four), regardless of the ADC resolution.
- ☺ They require on-chip support for the serial protocol or the use of a suitable software library.
- ☹ The data transfer may be slower than a parallel alternative.
- ☹ They can be comparatively expensive.

Two examples of the use of serial ADCs follow.

Using a current-mode ADC

As we discussed in ‘Background’, use of current-based data transmission can be useful in some circumstances.

A number of current-mode sensor components (e.g. the Burr-Brown⁷ XTR105) and ADCs (e.g. the Burr-Brown RCV420) are now available.

In addition, **CURRENT SENSOR** [page 802] discusses current sensing using voltage-mode ADCs.

Hardware resource implications

Use of the internal ADC will generally mean that at least one input pin is unavailable for other purposes; use of an external ADC will require the use of larger numbers of port pins.

7. www.burr-brown.com

Use of the on-chip ADC may also have an impact on the power consumption of your microcontroller.

Reliability and safety implications

Use of ADCs has no particular safety implications. However, all things being equal, an application using an internal ADC is likely to prove more reliable than the same application created using an external ADC, largely because the hardware complexity of the ‘internal’ solution will be much less than that of the ‘external’ solution.

Portability

All ADCs vary in their properties: for example, code for one serial ADC will often not work without alteration on another serial device. However, the required changes are generally minor.

Overall strengths and weaknesses

- ☺ ADC inputs are essential in many applications.
- ☺ Microcontrollers with ADCs are more expensive than those without such facilities.

Related patterns and alternative solutions

Many microcontrollers now have multiple A/D converter channels available, making it possible to implement low-cost user inputs, for example for setting temperatures, operating points and so on.

The main advantage of this type of analogue input is that it directly provides user feedback: for example, we simply add a scale around the potentiometer control to indicate the required temperature. The cost of such an input sensor is frequently much lower than the corresponding digital (two switch plus display) equivalent (Figure 32.5).

However, in almost all circumstances the digital solution will provide more precise control and will not alter with age (as the performance of the potentiometer is likely to do). In addition, if we wish to alter the temperature of the system under software control we can easily, in the case of the digital solution, update the temperature display, while we cannot generally rotate the potentiometer under software control to match the new temperature.

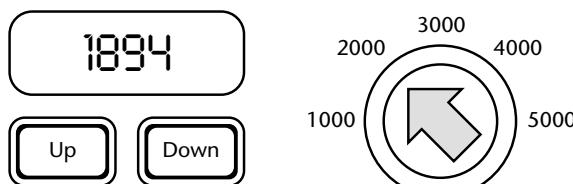


FIGURE 32.5 Two possible user interfaces, one based on an LCD/LED display with switches, the other based on an analog potentiometer

Nonetheless, the analog solution is often adequate, where high precision is not a requirement. For example, for a room thermostat control, a potentiometer may well be appropriate.

Example: Using an external SPI ADC

This example illustrates the use of an external, serial (SPI) ADC (Listings 32.1 to 32.3): the SPI protocol was described in detail in Chapter 24.

The hardware comprises an Atmel AT89S53 microcontroller and a Maxim⁸ Max1110 ADC: these devices are connected as shown in Figure 32.6.

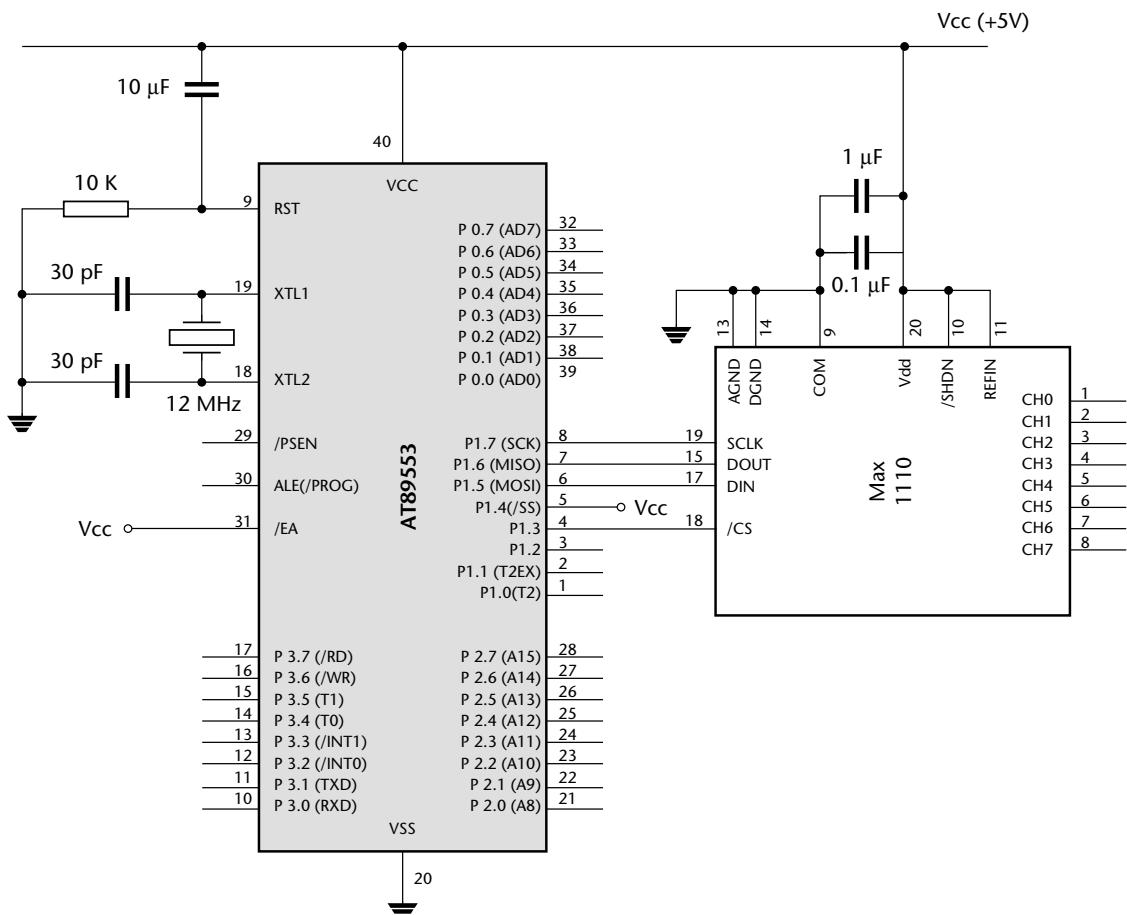


FIGURE 32.6 Connecting an SPI ADC to an 8051 microcontroller

```

/*-----*
Port.H (v1.00)

-----'

'Port Header' (see Chap 10) for project SPI_ADC

-*-----*/
```

```

// ----- SPI_Core.C -----
```

```

// Create sbits for all required chip selects here
sbit SPI_CS = P1^4;

// NOTE: pins P1.4, P1.5, P1.6 and P1.7 also used - see text

/*-----*
----- END OF FILE -----*
```

```

-*-----*/
```

Listing 32.1 Part of an example illustrating the use of a serial (SPI) ADC

```

/*-----*
```

```

Main.C (v1.00)

-----'
```

```

Simple test program for SPI code library.

Reads from MAX1110 / 1111 SPI ADC.

-*-----*/
```

```

#include "Main.h"
#include "SPI_Core.h"
#include "SPI_1110.h"
#include "Delay_T0.h"

// In this test program, we define the error code variable here.
tByte Error_code_G = 0;

void main(void)
{
    tByte Data1 = 0;
    tByte Data2 = 0;

    tWord Data_address = 0;
```

```
// See text for details

// SPI Control Register.
// Bit 0 = SPR0
// Bit 1 = SPR1 (these two control the clock rate)
// Bit 2 = CPHA (transfer format, see p15 of AT89S53 docs)
// Bit 3 = CPOL (clock polarity, 1 = high when idle, 0 = low when idle)
// Bit 4 = MSTR (1 for master, 0 for slave)
// Bit 5 = DORD (data order, 1 for LSB first, 0 for MSB first)
// Bit 6 = SPE  (enable SPI)
// Bit 7 = SPIE (enable SPI interrupt, if ES is also 1)

// To interface with the MAX1110 ADC, we need a clock rate in the
// range 50-500 kHz, so with a 12 MHz oscillator SPR0 and SPR1 are
// set at 1 and 0, so SPI speed is Fosc / 64, which is 187.5 kHz
//
// CPHA and CPOL both need to be zero, see MAX1110 docs
// DORD needs to be zero (MSB first)
// MSTR, SPE, SPIE need to be one
// -> SPCR = 0x52;
SPI_Init_AT89S53(0x52);

while (1)
{
    //
    // Read ADC byte
    Data2 = SPI_MAX1110_Read_Byte();

    // Display data
    P2 = 255 - Data2;

    // Display error codes (if any)
    P3 = 255 - Error_code_G;

    Hardware_Delay_T0(1000);
}
}

/*
-----*
---- END OF FILE -----
-*-----*/
```

Listing 32.2 Part of an example illustrating the use of a serial (SPI) ADC

```
/*-----*
 * SPI_1110.C (v1.00)
 *
 *-----*
 * Simple SPI library for Atmel AT89S53
 * - allows data to be read from MAX1110 / 1111 ADC
 *
 *-----*/
#include "Main.H"
#include "Port.h"

#include "SPI_Core.h"
#include "SPI_1110.h"
#include "TimeoutH.h"

// ----- Public variable declarations -----
// The error code variable
//
// See Port.H for port on which error codes are displayed
// and for details of error codes
extern tByte Error_code_G;

/*-----*/
SPI_MAX1110_Read_Byte()

Read a byte of data from the ADC.

/*-----*/
tByte SPI_MAX1110_Read_Byte(void)
{
    tByte Data, Data0, Data1;

    // 0. Pin /CS is pulled low to select the device
    SPI_CS = 0;

    // 1. Send a MAX1110 control byte

    // Bit 7 = 1 (start of control byte)
    // Bit 6 = SEL2 - {SEL2, SEL1, SEL0 select the input channel}
    // Bit 5 = SEL1 - (see Maxim documentation)
    // Bit 4 = SEL0
    // Bit 3 = 1 for unipolar, 0 for bipolar
    // Bit 2 = 1 for single ended, 0 for differential
    // Bit 1 = 1 for fully operational, 0 for power-down mode
    // Bit 0 = 1 for external clock, 0 for internal clock
```

```

// Control byte 0x8F sets single-ended unipolar mode,
// input channel 0 (pin 1)
// 0 (pin 1)
SPI_Exchange_Bytes(0x8F);

// 2. The data requested is shifted out on S0 by sending two dummy
// bytes
Data0 = SPI_Exchange_Bytes(0x00);
Data1 = SPI_Exchange_Bytes(0x00);

// The data are contained in bits 5-0 of Data0
// and 7-6 of Data1 - shift these bytes to give a combined byte,
Data0 <= 2;
Data1 >= 6;
Data = (Data0 | Data1);

// 3. We pull the /CS pin high to complete the operation
SPI_CS = 1;

// 4. We return the required data
return Data; // Return SPI data byte
}

/* -----
--- END OF FILE ---
----- */

```

Listing 32.3 Part of an example illustrating the use of a serial (SPI) ADC

Example: Using an external I²C ADC

This example illustrates the use of an external, serial (I²C) ADC (Listings 32.4 to 32.6): the I²C protocol was described in detail in Chapter 23.

The ADC hardware comprises a Maxim⁹ Max127 ADC: this device is connected to the microcontroller as shown in Figure 32.7.

```

/* -----
----- */

Port.H (v1.00)

-----
'Port Header' (see Chap 10) for project ADC_M127 (see Chap 32)

```

⁹. www.maxim-ic.com

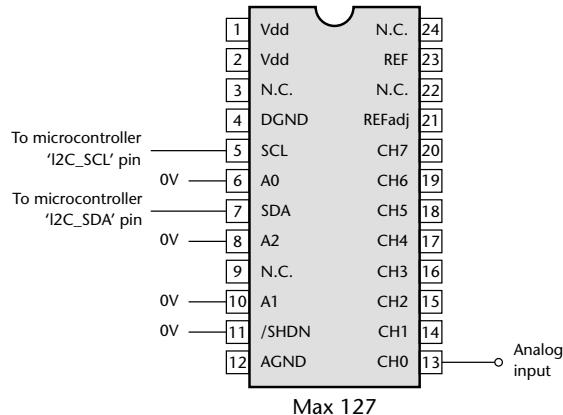


FIGURE 32.7 The Maxim Max127 serial (I^2C) ADC

```

- * ----- * /
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P2

#endif

// ----- I2C_Core.C -----
// The two-wire I2C bus
sbit I2C_SCL = P1^7;
sbit I2C_SDA = P1^6;

/* ----- END OF FILE ----- */
- * ----- * /

```

Listing 32.4 Part of an example illustrating the use of a serial (I^2C) ADC

```
/*-----*  
Main.C (v1.00)  
-----  
  
Simple test program for I2C (MAX127 ADC) library.  
Connect a MAX127 to the SDA and SCL pins described  
in the library file (I2C_Core.C).  
Terminating resistors not generally required on the bus.  
-----*/  
  
#include "Main.h"  
#include "I2C_m127.h"  
#include "Delay_T0.h"  
  
extern tByte ADC_G;  
  
// In this test program, we define the error code variable here  
// (Usually in the scheduler library)  
tByte Error_code_G = 0;  
  
void main( void )  
{  
    while(1)  
    {  
        I2C_ADC_MAX127_Read();  
        P1 = ADC_G;  
        P2 = Error_code_G;  
        Hardware_Delay_T0(1000);  
    }  
}  
/*-----*  
---- END OF FILE -----  
-----*/
```

Listing 32.5 Part of an example illustrating the use of a serial (I²C) ADC

```
/*-----*  
I2C_m127.C (v1.00)  
-----  
Code library for MAX127 (I2C) ADC.  
-*-----*/  
  
#include "Main.H"  
#include "Port.h"  
  
#include "I2C_Core.h"  
#include "I2C_m127.h"  
#include "Delay_t0.h"  
  
// ----- Public variable definitions -----  
  
// The ADC value  
tByte ADC_G;  
  
// ----- Public variable declarations -----  
  
// The error codes - see scheduler  
extern tByte Error_code_G;  
  
// ----- Private constants -----  
  
// Chip address = 0101xxxW  
#define I2C_MAX127_ADDRESS (80)  
  
// Start bit set  
// Normal power mode (not in power-down mode)  
// Range 0 - 5V  
#define I2C_MAX127_MODE (0x80)  
  
// ----- Private variable definitions-----  
  
// The ADC channel (0 - 7)  
// *** Value here is required channel value << 4 ***  
// *** We are using Channel 2 ***  
static tByte I2C_MAX127_Channel_G = 0x20;  
  
/*-----*
```

```
I2C_ADC_MAX127_Read()  
Reads from the I2C 12-bit ADC  
The channel used is given by ADC_Channel_G  
This version reads only 8-bits (most sig) data  
----- */  
void I2C_ADC_MAX127_Read(void)  
{  
    I2C_Send_Start(); // Generate I2C START condition  
  
    // Send DAC device address (with write access request)  
    if (I2C_Write_Byte(I2C_MAX127_ADDRESS | I2C_WRITE))  
    {  
        Error_code_G = ERROR_I2C_ADC_MAX127;  
        return;  
    }  
  
    // Set the ADC mode and channel - see above  
    if (I2C_Write_Byte(I2C_MAX127_MODE | I2C_MAX127_Channel_G))  
    {  
        Error_code_G = ERROR_I2C_ADC_MAX127;  
        return;  
    }  
  
    I2C_Send_Stop(); // Generate STOP condition  
    I2C_Send_Start(); // Generate START condition (again)  
  
    // Send MAX127 device address (with READ access request)  
    if (I2C_Write_Byte(I2C_MAX127_ADDRESS | I2C_READ))  
    {  
        Error_code_G = ERROR_I2C_ADC_MAX127;  
        return;  
    }  
  
    // Receive first (MS) byte from I2C bus  
    ADC_G = I2C_Read_Byte();  
  
    I2C_Send_Master_Ack(); // Perform a MASTER ACK
```

```

// Here we require temperature only accurate to 1 degree C
// - we discard LS byte (perform a dummy read)
I2C_Read_Byte();

I2C_Send_Master_NAck(); // Perform a MASTER NACK

I2C_Send_Stop(); // Generate STOP condition
}

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 32.6 Part of an example illustrating the use of a serial (I^2C) ADC

Example: Using an external parallel ADC

This example illustrates this use of an 8-bit parallel ADC: the Maxim¹⁰ Max150 (Figure 32.8) (Listings 32.7 to 32.9).

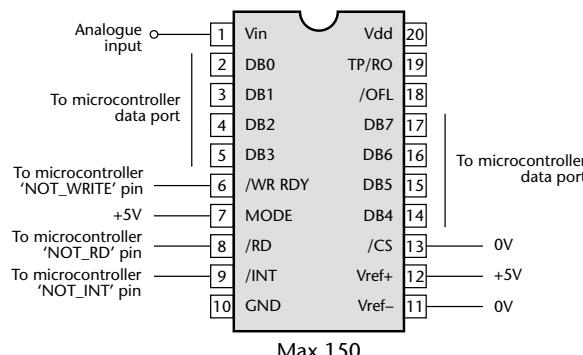
```

/*-----*
Port.H (v1.00)

-----*
'Port Header' (see Chap 10) for the project 'ADC_m150'

-----*/

```



```

// ----- ADC_M150.C -----
// Interface to the MAX150 parallel ADC
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 -----
// Connect LED from +5V (etc) to these pins, via appropriate resistor
// [see Chapter 7 for details]
// The 8 port pins may be distributed over several ports if required
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;

/* -----
   ---- END OF FILE -----
   */
```

Listing 32.7 Part of an example illustrating the use of a parallel (8-bit) ADC

```

/* -----
Main.c (v1.00)

-----
Demo program for ADC -> Bargraph display

Required linker options (see text for details):
OVERLAY
(main ~ (AD_Get_Sample,Bargraph_Update),
sch_dispatch_tasks ! (AD_Get_Sample,Bargraph_Update))

*/
#include "Main.h"
#include "2_01_12g.h"
#include "ADC_m150.h"
#include "BarGraph.h"
```

```

/* ..... */
/* ..... */

void main(void)
{
    SCH_Init_T2();           // Set up the scheduler
    ADC_MAX150_Init();       // Prepare the ADC
    BARGRAPH_Init();         // Prepare a bargraph-type display (P4)

    // Read the ADC regularly
    SCH_Add_Task(ADC_MAX150_Get_Sample, 10, 1000);

    // Simply display the count here (bargraph display)
    SCH_Add_Task(BARGRAPH_Update, 12, 1000);

    // All tasks added: start running the scheduler
    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

/*-----*
----- END OF FILE -----*
-----*/

```

Listing 32.8 Part of an example illustrating the use of a parallel (8-bit) ADC

```

/*-----*
----- ADC_m150.c (v1.00)

-----
Simple, single-channel, 8-bit A-D (input) library for 8051 family
- uses MAX150 8-bit parallel ADC.

See Chapter 32 for details.

-----*/
#include "Main.H"
#include "Port.h"

#include "Bargraph.h"

// ----- Public variable definitions -----

```

```
// Stores the most recent ADC reading
tByte Analog_G;

// ----- Public variable declarations -----
extern tByte Error_code_G;

/* -----
ADC_MAX150_Init()

Set up the MAX150 ADC. Using WR-RD mode (see data sheet)

*/
void ADC_MAX150_Init(void)
{
    // Set 'NOT read' pin high
    ADC_MAX150_NOT_Read_pin = 1;

    // Set 'NOT write' pin high
    ADC_MAX150_NOT_Write_pin = 1;

    // Prepare 'NOT Int' pin for reading
    ADC_MAX150_NOT_Int_pin = 1;
}

/* -----
ADC_MAX150_Get_Sample()

Get a single data sample (8 bits) from the ADC.

*/
void ADC_MAX150_Get_Sample(void)
{
    tWord Time_out_loop = 1;

    // Start conversion by pulling 'NOT Write' low
    ADC_MAX150_NOT_Write_pin = 0;

    // Take sample from A-D (with simple loop time-out)
    while ((ADC_MAX150_NOT_Int_pin == 1) && (Time_out_loop != 0));
    {
        Time_out_loop++; // Disable for use in dScope...
    }

    if (!Time_out_loop)
    {
        // Timed out
        Error_code_G =
```

```
    Analog_G = 0;
}
else
{
    // Set port to 'read' mode
    ADC_MAX150_port = 0xFF;

    // Set 'NOT read' pin low
    ADC_MAX150_NOT_Read_pin = 0;

    // ADC result is now available
    Analog_G = ADC_MAX150_port;

    // Set 'NOT read' pin high
    ADC_MAX150_NOT_Read_pin = 1;
}

// Pull 'NOT Write' high
ADC_MAX150_NOT_Write_pin = 1;
}

/*-----*-
---- END OF FILE -----
-*-----*/
```

Listing 32.9 Part of an example illustrating the use of a parallel (8-bit) ADC

Example: Using the c515c internal ADC

See **SEQUENTIAL ADC** [page 782] for a complete code library using the on-chip ADC in the Infineon c515c.

Further reading

Lynn, P. and Fuerst, W. (1998) *Introductory Digital Signal Processing with Computer Applications*, Wiley, Chichester.

ADC PRE-AMP

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you convert an analogue voltage signal into a range suitable for subsequent analogue-to-digital conversion?

Background

In a 5V system, an ADC will typically encode a range of analogue signals, from 0V to approximately 5V. If we have an analogue signal in the range 0–5 mV, we need to amplify this voltage prior to use of the ADC or the digital signal will be a very poor representation of the analogue original.

This pattern describes some suitable circuits. Note that this pattern is hardware based; no software is required.

Solution

An operational amplifier provides the basis of a widely used solution to the problem of scaling analogue signals and will be used here.

A wide range of operational amplifiers are available, including the ‘classic’ 741 or 411 chips and more recent devices such as the Microchip¹¹ MCP601.

We consider two basic operations: amplification and level shifting

Voltage amplification

All operational amplifiers are used in the same way when implementing a simple voltage amplifier. Figure 32.9 illustrates the approach.

The gain of this circuit, G , is given by:

$$G = \frac{V_{out}}{V_{in}} = \frac{R1 + R2}{R1} = 1 + \frac{R2}{R1}$$

This is very simply applied, as we illustrate in the following example.

Note that for precise gains, you need to use good-quality resistors, with a precision of 1%: a precision of 5% will not generally be adequate.

¹¹. www.microchip.com

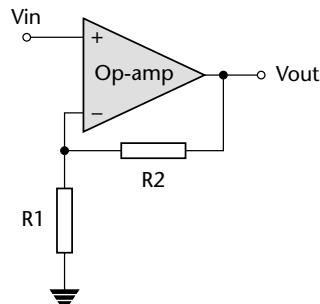


FIGURE 32.9 Amplifying a voltage signal using an op-amp

Level shifting

Suppose we wish to sample the speech signal shown in Figure 32.10.

The signal has a maximum value of around 50 mV, so will require amplification before it is sampled. In addition, however, the signal has a mean value of approximately 0V; it therefore must also be ‘level shifted’ to bring it into the positive voltage range that we are able to analyze.

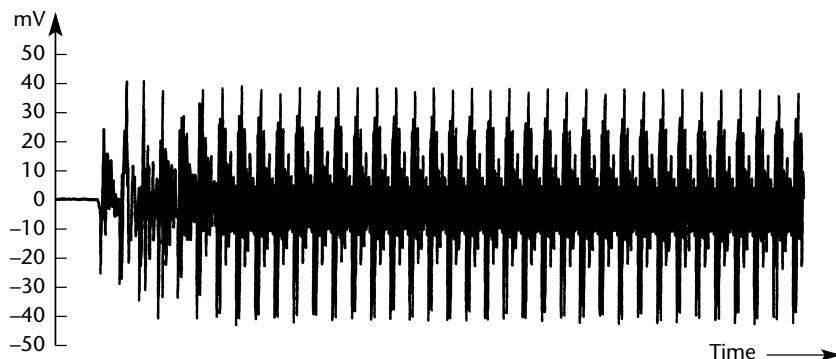


FIGURE 32.10 A speech signal with a mean value of approximately 0V

Figure 32.11 illustrates an op-amp circuit that may be used for level shifting.

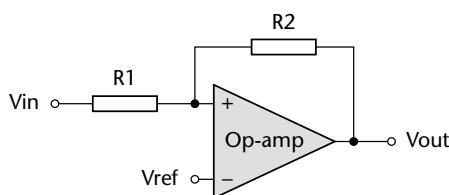


FIGURE 32.11 A simple op-amp level-shifting circuit

The output of this circuit is given by:

$$V_{out} = V_{ref} - \frac{R2}{R1} V_{in}$$

Another alternative level-shifting circuit is shown in Figure 32.12.

The output of this circuit is given by:

$$V_{out} = \frac{R4}{R3} V_1 - \frac{R2}{R1} V_{in}$$

Figure 32.12 is a particularly flexible circuit.

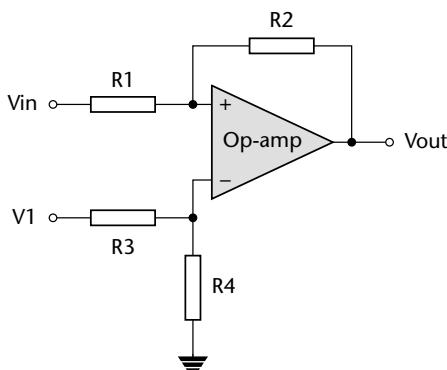


FIGURE 32.12 An alternative op-amp level-shifting circuit

Hardware resource implications

Use of this pattern has no implications for the hardware resources (e.g. CPU time or memory) on the microcontroller itself. Clearly, however, the op-amp and associated resistors will have an associated cost.

In addition, some op-amp circuits may require the presence of both positive and negative supply rails (e.g. +15V, -15V). This can add to the complexity of the power supply design and make it difficult (or more expensive) to use battery supplies. Increasing, 'single-supply' op-amps are available and can often be used in amplifier applications.

Reliability and safety implications

There are no specific reliability or safety implications.

Portability

This hardware-only pattern can be used with any microcontroller.

Overall strengths and weaknesses

- ☺ Simple and effective.
- ☹ May require use of a two-rail power supply.

Related patterns and alternative solutions

Example: An amplifier with a gain of 1,000

Suppose we have a sensor with a maximum 5mV output and we require a 5V input to our ADC, we need a gain of 1,000.

We can achieve this result (approximately) by applying the equation on page 779. Here we choose R_2 and R_1 to have a ratio of 1,000; here, values of, say, $R_1 = 1\text{ k}\Omega$ and $R_2 = 1\text{ M}\Omega$ (1% tolerance) will be fine.

Example: A microphone pre-amplifier

Figure 32.13 (adapted from an Analog Devices data sheet) shows an application of voltage amplifier to create a pre-amp for an electret microphone: the amplifier stage has a gain of approximately 10.

Note that the AD8517 is a single-supply op-amp.

R_1 is used to bias an electret microphone and C_1 blocks DC voltage from the amplifier. The magnitude of the gain of the amplifier is approximately R_3/R_2 when $R_2 = 10 \times R_1$. VREF should be equal to $1/2 \times 1.8\text{ V}$ for maximum voltage swing.

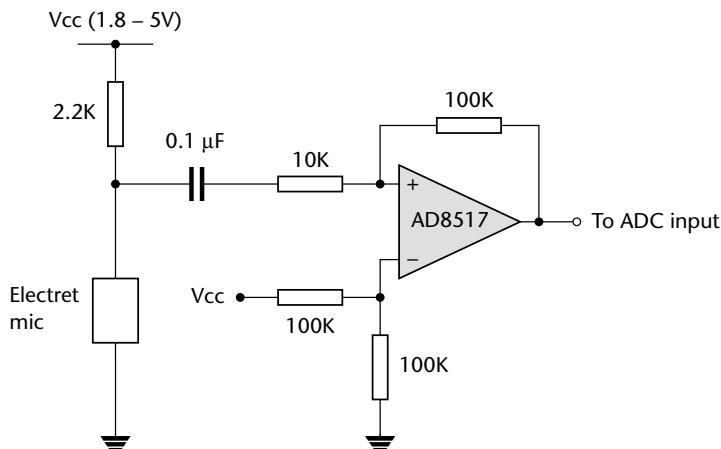


FIGURE 32.13 Creating a microphone pre-amp

Further reading

- 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.

SEQUENTIAL ADC

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you record a sequence of analog samples?

Background

In the pattern **ONE-SHOT ADC** [page 757], we were concerned with the use of analogue signals to address questions such as:

- What central-heating temperature does the user require?
- What is the current angle of the crane?
- What is the humidity level in Greenhouse 3?

In the present pattern, we are concerned with the recording of *sequences* of analogue samples, in order to address questions such as:

- How quickly is the car accelerating?
- How fast is the plane turning?
- What is the frequency of this sound?

For example, suppose that we are required to design a system to be used by an environmental organization to record and automatically classify the songs of whales recorded (underwater) in the Antarctic. Our embedded system will be encased in plastic and attached to a floating buoy. It will run on batteries for a two-year period and will be required to send a radio broadcast to base every time it detects a whale in the vicinity.

In this case we will need an underwater pressure transducer (hydrophone) to convert the fluctuations in water pressure produced by the whale song into a voltage value that can be processed by an ADC.

The output of the ADC will be a sequence of numbers (Figure 32.14). Individually, these numbers (e.g. '0.89') have no meaning; it is the *sequence* of numbers that allows us to determine, for example, the frequency components that make up the whale song.

In this pattern, we are concerned with the recording of sequences of analogue signals, at a fixed sampling frequency: for example, the whale song might be sampled at

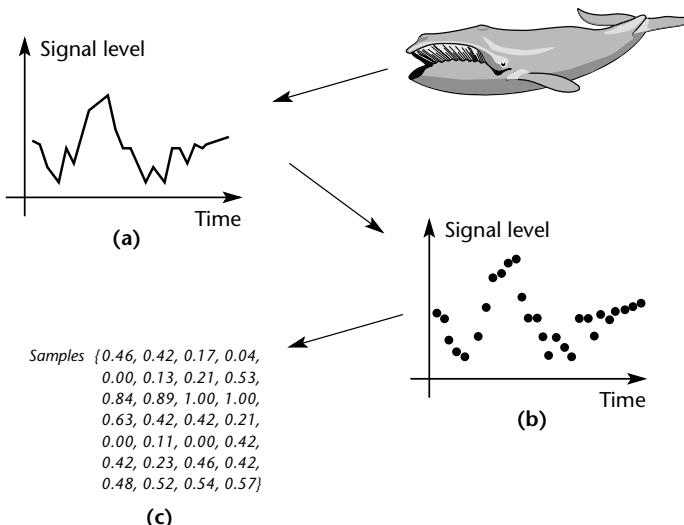


FIGURE 32.14 Performing analog-to-digital conversion on a sample of whale song

[(a) The original (analog) sample, recorded as a continuously varying voltage from an underwater microphone. (b) The sampled (quantized) version of the same signal. (c) The digital representation of the whale song to be stored on the computer.]

a rate of 50 kHz. To record such a sequence, we will build on the techniques discussed in **ONE-SHOT ADC** [page 757]. However, as we will discuss in ‘Solution’, the recording of sequences of signals introduces several new challenges for the developer.

Solution

There are several key design stages to be carried out implementing **SEQUENTIAL ADC**:

- 1 You need to determine the required sample rate
- 2 You may need to remove any high-frequency components from the input signal
- 3 You need to determine the required bit rate
- 4 You need to employ an appropriate software architecture
- 5 You need to select an appropriate ADC

We now deal with each of these points in turn.

Determining the required sample rate

As discussed in ‘Background’, we are concerned in this pattern with the recording of sequences of analogue signals, at a fixed sampling frequency. The first important design decision involves determining the required sample frequency.

Monitoring and signal-processing applications

Suppose that we wish to create a system to perform speech recognition. From the speech waveform alone (for example, the time-varying voltage waveform from a microphone), we aim to recognize the words spoken (Figure 32.15).

To determine the sample rate for this application, we need to know the *bandwidth* of the signal we are trying to sample. We then need to sample at a frequency of at least twice this bandwidth (a frequency known as the Nyquist frequency).

Remember that the bandwidth refers, simply, to the frequency of the highest frequency component in the signal we wish to measure. For example, Figure 32.16 shows an idealized example: a pure tone (sine wave) of frequency F_{sine} . In this case, the signal bandwidth is the same as the tone frequency (F_{sine}) and we need to sample at twice this frequency to represent the signal correctly.

More commonly, signals we wish to work with will have broadband characteristics (Figure 32.17).

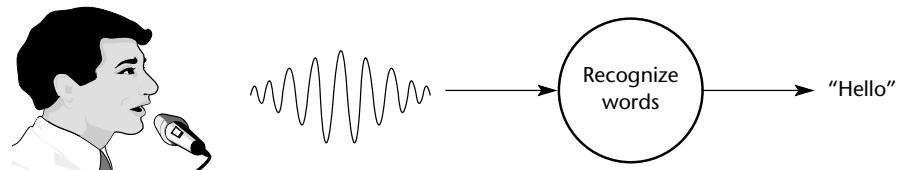


FIGURE 32.15 A simple speech recognition (classification) system

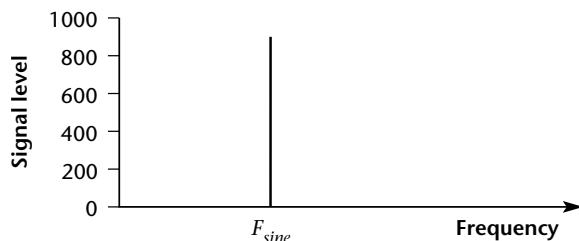


FIGURE 32.16 The representation of a pure tone in the frequency domain

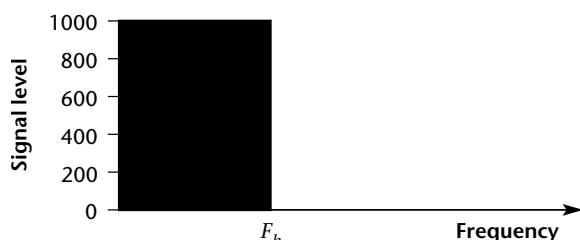


FIGURE 32.17 The representation of a broadband signal in the frequency domain

In this case, the bandwidth refers to the maximum signal frequency; here, F_b . In general, we may need to use specialized hardware, such as a spectrum analyzer, to determine the maximum frequency components in any signal we wish to sample. If we use such hardware then, in the case of the speech signal, we will find that the maximum frequency components have a frequency of around 20 kHz, implying that we need to sample at around 40 kHz (Figure 32.18).

Note that it is not always possible, or cost-effective, to sample at (or above) the highest signal frequency: see 'Removing high-frequency components' (below) for a discussion of techniques that can be used to deal with this issue.

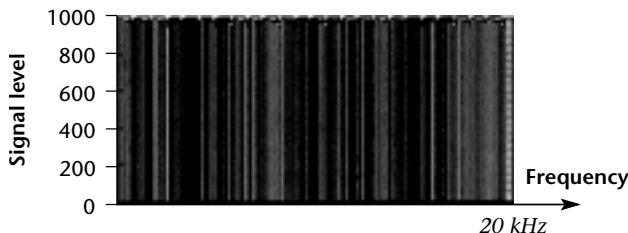


FIGURE 32.18 The representation of a broadband signal (bandwidth 20 kHz) in the frequency domain

Control applications

If we wish to carry out some of the following operations:

- Speech recognition
- Recording ECGs
- Recording auditory-evoked responses
- Vibration monitoring

then we can determine the maximum frequency components in the signal using an appropriate spectrum analyzer, and select the sample rate accordingly.

However, suppose we wish to develop a digital control application of the form illustrated in Figure 32.19.

This type of application also involves regular sampling, in this case of the motor speed. For this type of control application, different techniques are required to determine the required sampling rate; we delay consideration of these techniques until Chapter 35.

Removing high-frequency components

If you are sampling a signal at regular frequency (F Hz), you will generally need to include a filter in your system to remove all frequencies above $F/2$ Hz, to avoid a phenomenon known as aliasing.

Refer to **A-A FILTER** [page 794] for further details.

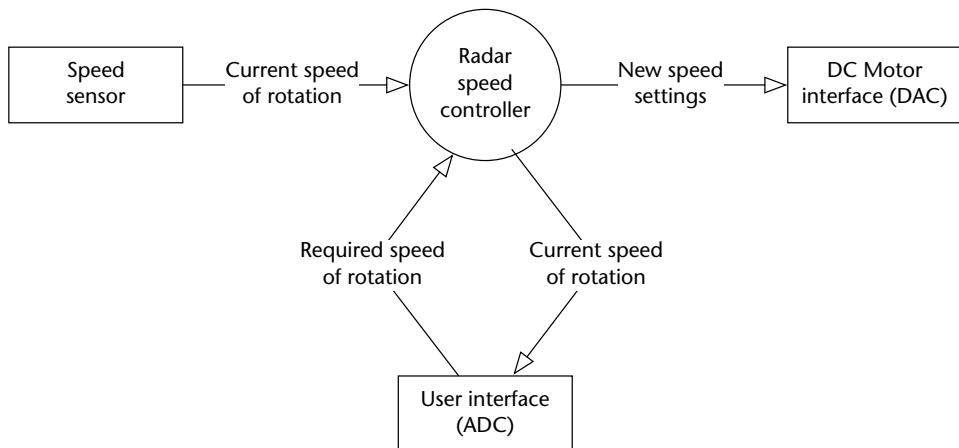


FIGURE 32.19 High-level design for a control application

Determining the required bit rate

The process of analogue-to-digital conversion is never perfect, since we are representing a continuous analogue signal with a digital representation that has only a limited number of possible values (Figure 32.20).

For example, if we were to use a 3-bit ADC, then we would have only eight possible signal levels (2^3) possible signal levels to represent our analogue signal. The error introduced by the digitization process is half the quantization level; thus, for our 3-bit ADC, this error would be equal to $\pm 1/16$ of the available analogue range. The resulting errors, over a sequence of samples, can be viewed as a form of quantization noise.

In most practical cases, use of a 12-bit ADC will provide adequate performance and even the most sophisticated speech-processing systems rarely use more than 16 bits.

Formal techniques for determining the required bit rate for a general sampled-data application are complex and beyond the scope of the present text: refer to Lynn and Fuerst (1998) for an introduction to this topic and to Oppenheim *et al.* (1999) for more detailed coverage.

Software architecture

The main impact that the use of **SEQUENTIAL ADC** has on the software architecture is the need to allow regular and frequent samples to be made.

Where sample rates of up to 1 kHz are required, this is rarely a problem. Obtaining a sample from the ADC typically requires 100 ns, and the scheduled architectures we have presented throughout this book can support the creation of suitable data-acquisition tasks with unduly loading the system.

Where sample rates in excess of 1 kHz are required, use of a fast 8051 device will generally be required. For example, the Dallas high-speed and ultra-high-speed family

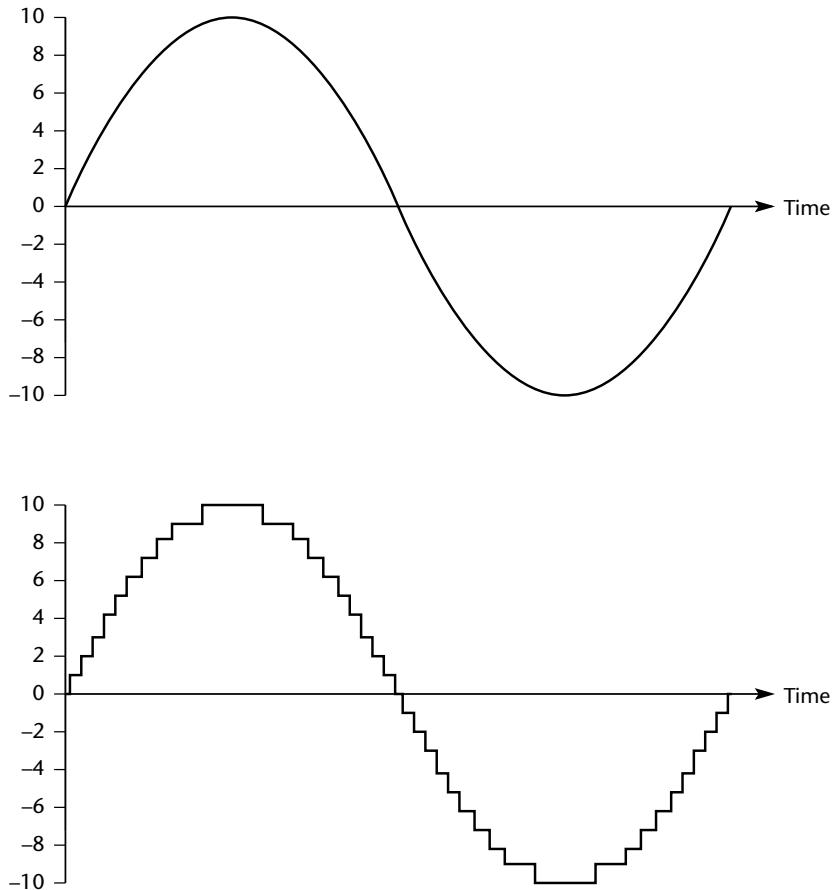


FIGURE 32.20 A pure tone in both its original analog form (top) and after quantization (bottom)

of devices will – as we demonstrated in Chapter 14 – allow the use of short tick intervals without unduly loading the CPU. However, even where sample rates of 10 kHz and above can be supported, this has important implications for other aspects of the design and, specifically, on the task durations. Use of **HYBRID SCHEDULER** [page 289] can be particularly valuable in these circumstances, since this allows the data sampling to be configured as a pre-emptive task.

Selecting an appropriate ADC

We discussed the range of possible ADCs in the pattern **ONE-SHOT ADC** [page 757]. Refer to this pattern for further information on this topic.

Note that, to a greater extent than one-shot applications, sequential ADC readings rely on rapid analogue-to-digital conversion and it is important to ensure that any ADC you select has a rapid conversion time. This can make ‘flash’ ADCs more appropriate than successive-approximation ADCs, for example, in these circumstances.

Hardware resource implications

Use of the internal ADC will generally mean that at least one input pin is unavailable for other purposes.

Use of the ADC may also have an impact on the power consumption of your microcontroller.

Reliability and safety implications

All things being equal, an application using an internal ADC is likely to prove more reliable than the same application created using an external ADC, largely because the hardware complexity of the 'internal' solution will be much less than that of the 'external' solution.

More specifically, use of an ADC with a long conversion time may introduce delays that will impact on the general performance of signal-processing applications and which may impact on the stability of control applications. Make sure the speed of the ADC is an appropriate match for your intended application.

Portability

All ADCs vary in their properties: for example, code for one serial ADC will not generally work without alteration on another serial device. However, the required changes are generally minor.

Overall strengths and weaknesses

- 😊 Use of sequential ADC inputs is essential in many applications.
- 😢 The need to sample data frequently will have an significant impact on the system architecture.

Related patterns and alternative solutions

See the rest of this chapter for related patterns and alternative solutions.

Example: Using the c515c internal ADC

This small library illustrates how we can make analogue readings, as required, using the on-chip ADC in an Infineon c515c microcontroller (Listings 32.10 to 32.12 (See also Figure 32.21)).

The ADC is initialized. Each time a reading is required, we start the ADC conversion and wait (with timeout, of course) for the conversion to complete.

The duration of the individual ADC tasks depends on the speed of the internal ADC.

```

/*-----*
Port.H (v1.00)

-----*
'Port Header' (see Chap 10) for the project ADC_BAR

-----*/
// ----- ADC_515c.C -----
// Reads from ADC channel 0 (Pin 6.0)
// ----- Bargraph.C -----
// Connect LED from +5V (etc) to these pins, via appropriate resistor
// [see Chapter 7 for details]
// The 8 port pins may be distributed over several ports if required
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;

/*-----*
--- END OF FILE ---
-----*/

```

Listing 32.10 Part of an example illustrating the use of the internal ADC in an Infineon c515c microcontroller

```

/*-----*
Main.c (v1.00)

-----*
Demo program for ADC -> Bargraph display

Required linker options (see Chapter 14 for details):
OVERLAY
(main ~ (AD_Get_Sample,Bargraph_Update),
sch_dispatch_tasks ! (AD_Get_Sample,Bargraph_Update))

```

```

/*-----*/
#include "Main.h"
#include "2_01_10i.h"
#include "ADC_515c.h"
#include "BarGraph.h"

/* . . . . . */
/* . . . . . */

void main(void)
{
    SCH_Init_T2();      // Set up the scheduler
    AD_Init();          // Prepare the ADC
    BARGRAPH_Init();   // Prepare a bargraph-type display (P4)

    // Read the ADC regularly
    SCH_Add_Task(AD_Get_Sample, 10, 1000);

    // Simply display the count here (bargraph display)
    SCH_Add_Task(BARGRAPH_Update, 12, 1000);

    // All tasks added: start running the scheduler
    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

/*-----*
----- END OF FILE -----
*-----*/

```

Listing 32.11 Part of an example using the internal ADC in an Infineon c515c microcontroller

```

/*-----*/
ADC_515c.c (v1.00)

-----
Simple, single-channel, 8-bit A-D (input) library for C515c

/*-----*/
#include "Main.H"
#include "Bargraph.h"

```

```
// ----- Public variable definitions -----
// Stores the most recent ADC reading
tByte Analog_G;

/*-----*
AD_Init()

Set up the A-D converter.

-*-----*/
void AD_Init(void)
{
    // Select internally-triggered single conversion
    // Reading from P6.0 (single channel)
    ADEX = 0;    // Internal A/D trigger
    ADM = 0;     // Single conversion
    MX2 = MX1 = MX0 = 0; // Read from Channel 0 (Pin 6.0)

    // Leave ADCON1 at reset value: prescalar is /4
}

/*-----*
AD_Get_Sample()

Get a single data sample (8 bits) from the (10-bit) ADC.

-*-----*/
void AD_Get_Sample(void)
{
    tWord Time_out_loop = 1;

    // Take sample from A-D
    // Write (value not important) to ADDATL to start conversion
    ADDATL = 0x01;

    // Take sample from A-D (with simple loop time-out)
    while ((BSY == 1) && (Time_out_loop != 0));
    {
        // Time_out_loop++; // Disable for use in dScope...
    }

    if (!Time_out_loop)
    {
        Analog_G = 0;
    }
    else
```

```

    {
        // 10-bit A-D result is now available
        Analog_G = ADDATH; // Read only 8 most significant 8-bits of A-D
    }
}

/* -----
--- END OF FILE
----- */

```

Listing 32.12 Part of an example using the internal ADC in an Infineon c515c microcontroller



FIGURE 32.21 Output (in the Keil hardware simulator) from the application described in Listings 32.10 to 32.12

[Note: This is the output from the hardware simulator in Keil C51 v5.5; at the time of writing, the Infineon range is not fully supported in Keil C51 v6.1 (included with this book). Refer to the Keil WWW site¹² for possible product updates.]

12. www.keil.com

Further reading

- 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 FILTER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you remove high-frequency components from a signal prior to digitization?

Background

As we discussed in **SEQUENTIAL ADC** [page 782], the sample rate in a sampled-data system must satisfy the Nyquist criterion; that is, sample rate (in Hz) must be at least twice the highest frequency (in Hz) that you wish to record or analyze. Therefore, if, for example, we wish to record speech, for a speech-recognition application to be used in a factory, we may decide a 5kHz bandwidth will be adequate and that we will therefore use a 10 kHz sample rate (Figure 32.22).

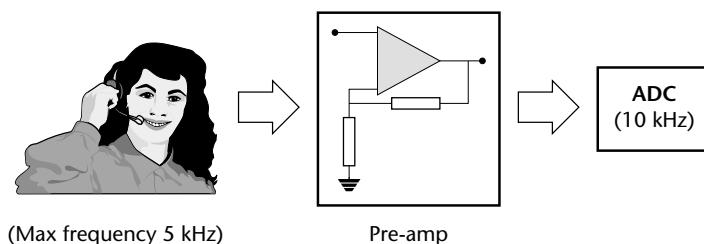


FIGURE 32.22 Performing speech recognition

However, there remains a problem. In the factory environment, there will be frequencies above our 5 kHz limit. Speech signals alone extend up to 20 kHz; other environmental sounds may extend even higher. All frequencies within the range of our microphone will be sampled at 5 kHz, giving rise to an effect known as aliasing.

To see what aliasing means in this context, first consider Figure 32.23. This figure illustrates a signal (a pure tone) sampled at a rate that satisfies the Nyquist criterion.

Figure 32.24 illustrates the impact of aliasing. In this figure, the solid line is a high-frequency signal (well above the Nyquist limit) which, we will assume, has been sampled. The dotted line shows how this high-frequency signal will be represented after sampling. What the figure illustrates is that, if we attempt to sample a high-frequency signal (that is, one with a frequency greater than half the sample rate), the resulting digital representation will simply not be correct.

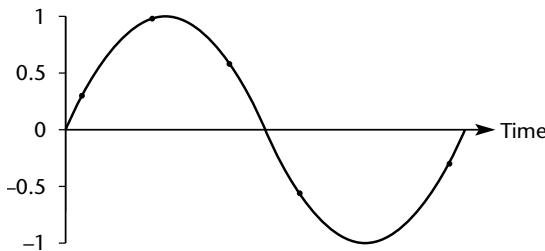


FIGURE 32.23 A sampled pure-tone signal

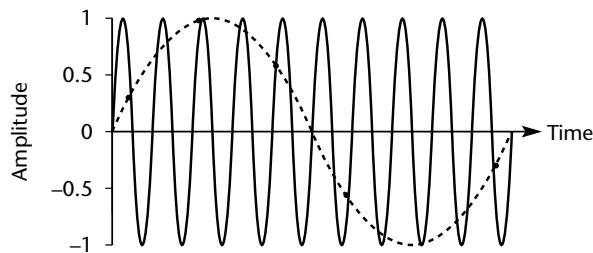


FIGURE 32.24 The impact of aliasing

It is vital to appreciate that, as far as sampled-data signal is concerned, both the low-frequency signal (sampled in Figure 32.23) and the high-frequency signal (sampled in Figure 32.24) have *exactly* the same representation; **it is impossible to distinguish these two signals after sampling, and no form of post-processing can reverse this effect.**

The only way to solve the aliasing problem is by using an anti-aliasing filter before sampling the data. The anti-aliasing filter will take the form of a *low-pass* filter (Figure 32.25).

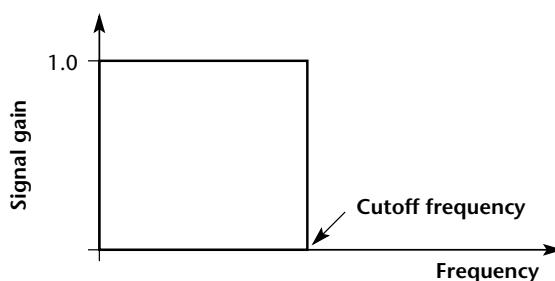


FIGURE 32.25 An idealized low-pass filter

Very broadly, filters can be thought of as devices that measure the signal frequencies present in the input and alter the amounts of those frequencies present at the output. The ranges of input frequencies which the filter amplifies (or at least does not attenuate) are known as the pass bands while the ranges that the filter suppresses are called the stop bands. In a low-pass filter (such as that shown in Figure 32.25), the pass band covers a lower frequency range than the stop band.

If we include an anti-aliasing filter, our complete speech-acquisition system will then take the form shown in Figure 32.26.

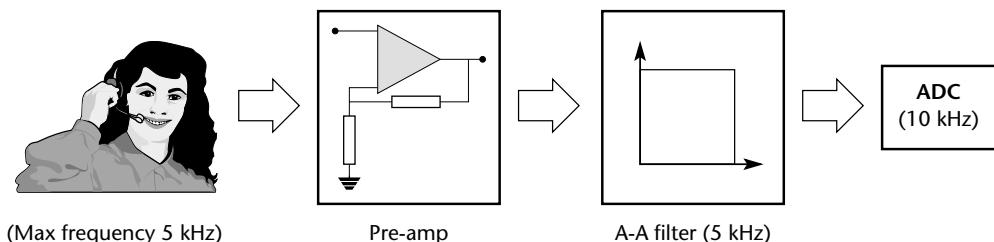


FIGURE 32.26 Adding an anti-aliasing filter to the speech-recognition system

Solution

We discussed the need for anti-aliasing (A-A) filters in ‘Background’. In theory, design and use of such a filter is very straightforward, for we simply require a good-quality low-pass filter with a cutoff frequency that corresponds to the maximum frequency we wish to sample; that is, it will be, at most, half of the sample rate.

In practice, the idealized filter characteristics illustrated in Figure 32.25 cannot be achieved and the filter in Figure 32.27 is more representative of real filters.

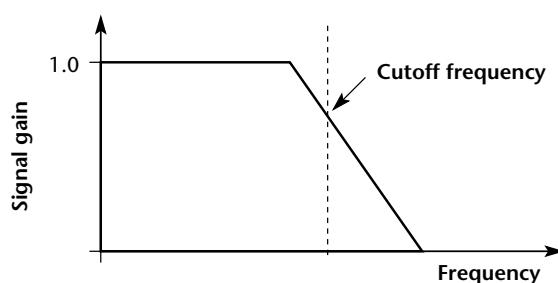


FIGURE 32.27 A more realistic low-pass filter characteristic

[Note: while this figure provides a more realistic representation of the filter characteristics at the boundary between the ‘pass’ and ‘stop’ bands, it is still idealized.]

As a general rule, the closer that a particular filter design comes to matching the ideal filter shown in Figure 32.25, the more complex and more expensive the filter becomes.

We consider a range of possible A-A filters in this section.

Simple op-amp filters

In **ADC PRE-AMP** [page 777] we considered the pre-processing of analogue signals for the purposes of signal amplification and level shifting. We can build on this approach, to provide a filtering characteristic. Figure 32.28 illustrates the basic approach.

The performance of this filter is shown, schematically, in Figure 32.29.

From Figure 32.29, F_1 (Hertz) is calculated from C (in Farads) and R (Ohms) as follows:

$$F_1 = \frac{1}{2\pi CR_2}$$

The gain in the low-frequency pass band is given by:

$$\frac{R_2}{R_1}$$

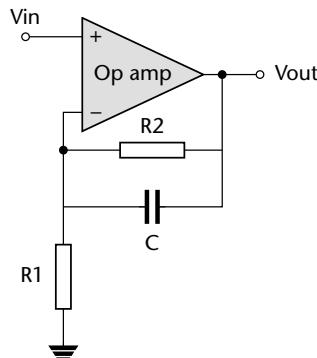


FIGURE 32.28 Low-pass op-amp filter with gain

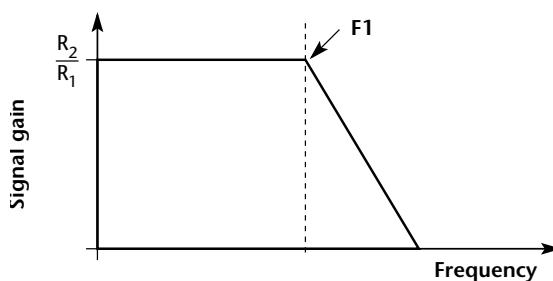


FIGURE 32.29 The idealized characteristics of the filter shown in Figure 32.25

More advanced op-amp based designs

To obtain improved performance from op-amp based filters, more complex designs with multiple stages are required. The design of such filters is beyond the scope of this book.

Note, however, that various filter-design packages are available which can allow you to create suitable filters. For example, Microchip distribute a free filter-design package – FilterLab – available from their WWW site¹³ – which may be used to create appropriate designs (Figure 32.30).

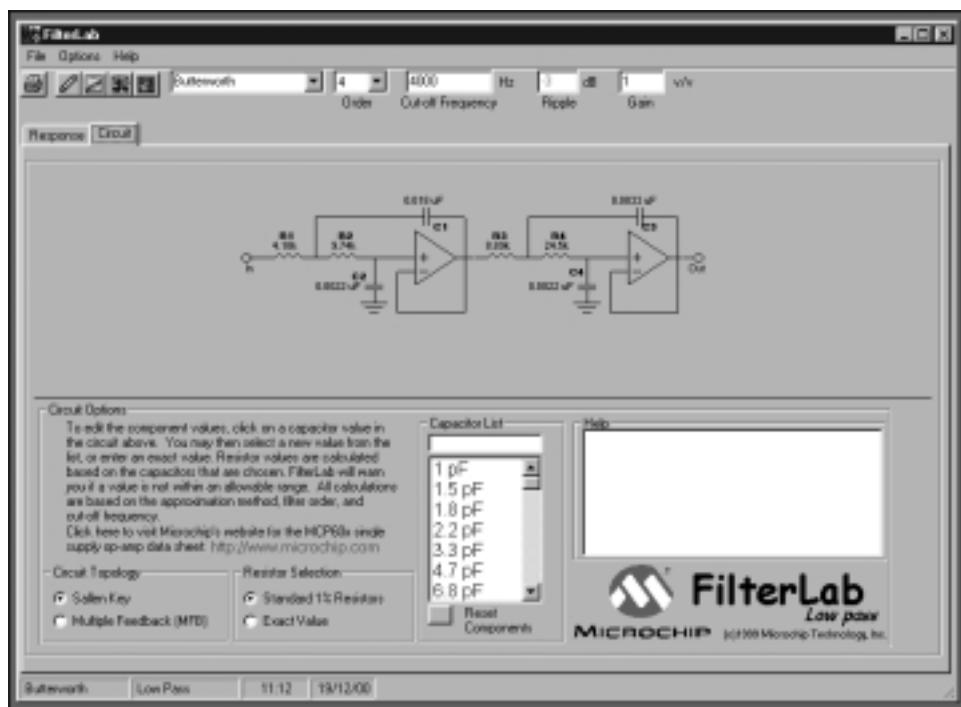


FIGURE 32.30 Designing a good-quality A-A filter using the FilterLab package from Microchip.
(Reproduced courtesy of Arizona Microchip Technology Ltd)

Switched-capacitor filter ICs

A number of switched-capacitor filters are available; these are widely used in A-A applications. The performance of such filters is much higher than that which can be obtained from the single op-amp design given earlier; advanced op-amp designs can compete with these filters, but such designs (typically requiring more than 10 op-amps) tend to be bulky and must be designed and constructed with care.

13. www.microchip.com

Switched-capacitor filters are driven by an external clock at, typically, 100 times the cutoff frequency. Thus, for a 10 kHz cutoff frequency, we need to drive the clock with a 1 MHz clock. This is easy to achieve with an 8051-based design: see **HARDWARE PRM** [page 742] for a discussion of techniques which may be used to generate clock frequencies in this range with little or no software overhead.

Note that using the microcontroller to control the clock rate, and hence cutoff frequency, of the filter means that we can change the cutoff frequency by altering the clock rate. This is useful if you are creating an application that needs to vary the sample rate.

The chief drawback with switched-capacitor designs is that the clock introduces noise in the filtered signal. For high-quality applications, the switched-capacitor filter will typically be used in conjunction with an op-amp filter (precisely as discussed in Section ‘Simple op-amp filters’) to remove the switching noise.

Various switched-capacitor products are available. For example, Linear Technology¹⁴ make the widely used ‘1064’ range of switched-capacitor filters, aimed at A-A applications. The Maxim¹⁵ Max7408 (and similar devices) are a useful alternative. Note that the Maxim chips tend to be designed for single-rail supplies, while the LT devices generally require both positive and negative supply rails. Please consult the manufacturer’s data sheets for these devices for further details.

Continuous-time filter ICs

A limited number of continuous-time filter ICs also exist; these are not clock driven and therefore do not require additional filters to remove the clock noise. Two examples are the Maxim Max270 and Maxim Max275.

Hardware resource implications

Use of this pattern has no implications for the hardware resources (e.g. CPU time or memory) on the microcontroller itself. Clearly, however, the op-amp and associated resistors will have an associated cost.

In addition, some op-amp circuits may require the presence of both positive and negative supply rails (e.g. +15V, -15V). This can add to the complexity of the power supply design and make it difficult (or more expensive) to use battery supplies. Increasing, ‘single-supply’ op-amps are available and can often be used in amplifier applications.

Reliability and safety implications

There are no specific reliability or safety implications.

14. www.linear-tech.com

15. www.maxim-ic.com

Portability

This hardware-only pattern can be used with any microcontroller.

Overall strengths and weaknesses

- 😊 Use of A-A FILTER can greatly improve the performance of many systems.
- 😢 Adding such a filter increases the product cost.

Related patterns and alternative solutions

Good-quality A-A filters can add significantly to the cost of an embedded application. Sometimes it is possible to reduce the need for A-A filters altogether or at least to manage with simple op-amp filters, without reducing the signal quality. This is possible if we *over-sample* the signal.

Consider our speech-recognition system (introduced in ‘Background’) again. This system required data at 10 kHz sample rate, in order to analyze speech sounds at up to 5 kHz. However, we assumed that other sounds in the vicinity would extend up to a frequency of at least 20 kHz and that the frequencies above 5 kHz would need to be removed by a good-quality A-A filter.

Suppose, however, that we carry out the following:

- Filter the signal using a low-quality, 5 kHz, analogue A-A filter
- Sample at **40 kHz** (thereby correctly sampling all frequencies up to 20 kHz)
- Digitally low-pass filter the 40 kHz signal, in software, to remove frequencies above 5 kHz
- Discard three out of every four samples (a process referred to as decimation), to provide the 10 kHz data that we require

This process results in a high-quality signal, without the need to invest in an expensive analogue A-A filter: it is for these reasons that almost all manufacturers of CD players use over-sampling (typically 4x) to reduce the cost of their products without sacrificing quality.

Performing the required digital filtering operating is straightforward (e.g. see Lynn and Fuerst, 1998). The main drawback with this approach is that we require high sample rates; this may, in turn, necessitate the use of a HYBRID SCHEDULER [page 333].

Example: Applying an A-A filter in a speech-recognition system

Consider the speech-recognition example discussed earlier in this pattern and summarized in Figure 32.31.

Figure 32.32 shows a possible design for a suitable A-A filter, created using the Microchip FilterLab software.

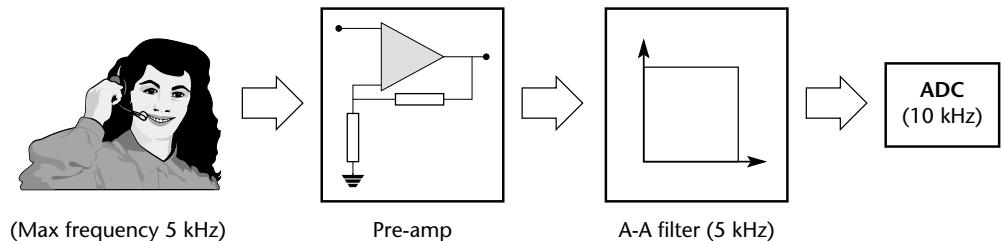


FIGURE 32.31 The speech-recognition system revisited

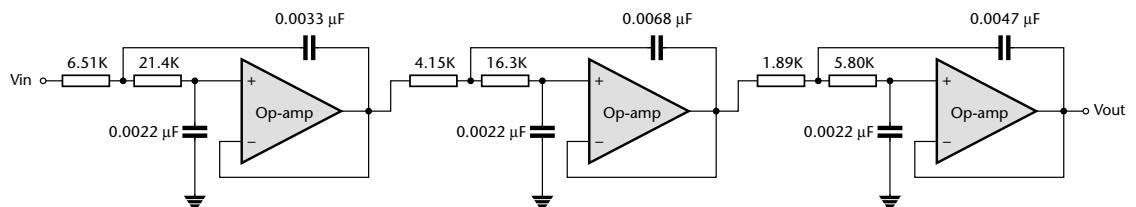


FIGURE 32.32 An anti-aliasing filter suitable for use with the speech-recognition example

Further reading

- 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.

CURRENT SENSOR

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you monitor the current flowing through a DC load?

Background

Despite the name, **CURRENT SENSOR** involves the measurement of analogue voltage: see **ONE-SHOT ADC** [page 757] and **SEQUENTIAL ADC** [page 782] for relevant background information.

Solution

We consider solutions to the problem of current sensing in this section.

Using current-sense resistors

The theoretical basis of traditional current-sensing techniques is very straightforward. Suppose, for example, we wish to monitor the current flowing through the load shown in Figure 32.33.

To measure this current, we can place a resistor in series with the load and measure the voltage drop across the resistor (Figure 32.34).

The current through the load can then simply be determined, from Ohm's law, as follows:

$$I_{load} = \frac{V_{load}}{R_{load}}$$

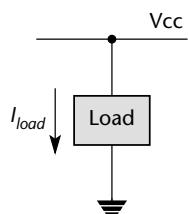


FIGURE 32.33 The problem: we wish to measure the current flowing through this load

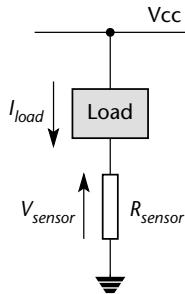


FIGURE 32.34 Using a resistor as a ‘current to voltage converter’ in a current-sense application

Any resistor may be used, but in most cases a component with a very small resistance is best: such devices are specially produced for current sensing. Typical values used will be less than 1 Ohm.

To understand why small resistors are required, suppose that we have a 12V load (connected to a 12V supply), with a current requirement of 1A: this is a typical requirement for many small bulbs or DC motors. If we add a 10 Ohm resistor in series with this load, the voltage drop across the resistor (again determined from Ohm’s law, $V = IR$), will be 10V. If we use a resistor of 0.2 Ohms, the voltage drop will be reduced to 0.2V: this is acceptable in most situations.

Note that we need to have a sufficiently large voltage across the sensor resistor to enable us to monitor the voltage using an ADC connected to the microcontroller. To allow for reliable monitoring, you will generally need to have a voltage drop across the resistor of around 0.1V. Any less than this and you run the risk of supply fluctuations or EMI interfering with your measurements.

Note also that you must ensure that the sensor resistor is of an appropriate power rating. The required power rating, in Watts, can be determined as follows:

$$P_{resistor} = R_{sensor}(I_{load})^2$$

Thus, with a 0.5 Ohm resistor and a 3A load, the required power rating will be 4.5W.

Current-sensing resistors are available in ratings of 50W and above. However, the stated power ratings often assume the use of a heat sink connected to the resistor. If you do not use such a heat sink, then choose a resistor with at least double the calculated power rating.

A resistor-free alternative

In microcontroller-based systems using MOSFETs or BJTs for switching loads, we can often carry out current sensing without the use of a separate current-sense resistor.

For example, consider Figure 32.35.

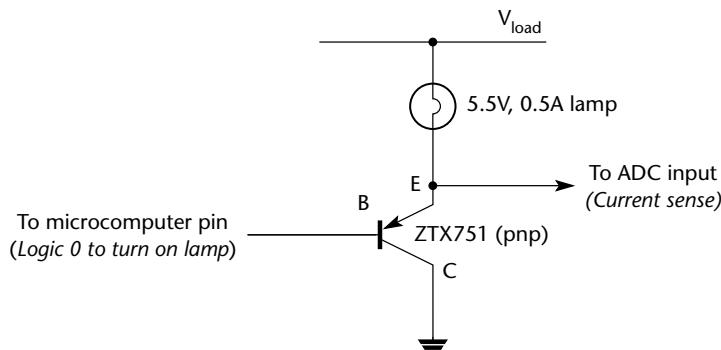


FIGURE 32.35 Current sensing without a resistor

With the BJT in saturation, the voltage drop across the emitter-collector terminals will be about 1V. We can therefore determine whether the bulb has blown by measuring the voltage at the emitter (relative to ground): if the voltage is ~1V, the bulb is lit. Note that we cannot reliably determine the level of the load current using this approach: we can only tell whether the load is being driven (or not).

MOSFETs provide a similar solution: see, for example, Figure 32.36.

In this circuit, if the MOSFET has an 'on' resistance of R_{on} , then the voltage at the MOSFET drain pin, when the load is driven, is given (again using Ohm's law) by:

$$V_{Drain} = I_{load}R_{on}$$

For the IRF540N (for example), R_{on} is 0.055Ω , so that – at a typical load current of around 2A – we have a voltage of ~0.1V at the drain pin, when the load is being driven. This may be readily measured even with an 8-bit ADC.

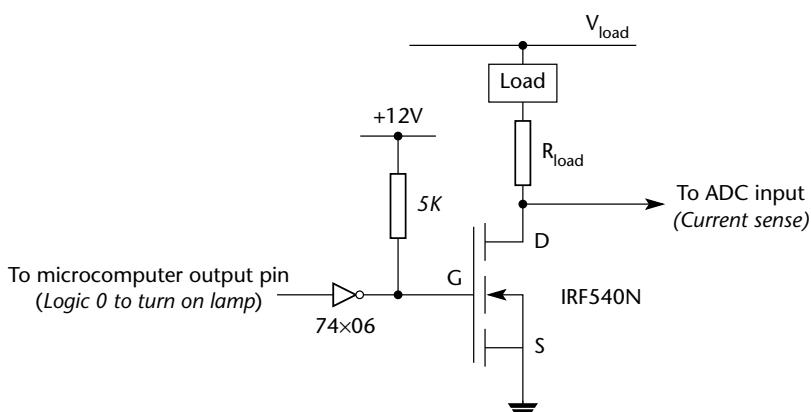


FIGURE 32.36 Current sensing without a resistor

Hardware resource implications

Use of this pattern generally requires the use of an on-chip or external ADC or comparator circuit.

Reliability and safety implications

Careful use of this technique can improve reliability and safety by allowing changes in the condition of a load (e.g., blown bulb, stalled DC motor) to be detected.

Portability

This hardware-only pattern is highly portable.

Overall strengths and weaknesses

- 😊 Can improve reliability and safety.
- 😢 Requires use of an ADC or similar hardware.

Related patterns and alternative solutions

See **ONE-SHOT ADC** [page 757] for alternative current-sense solutions.

Example: Detecting a blown bulb

As part of a security system, a 12V, 20W bulb (in fact, a car headlight bulb) is to be controlled by a microcontroller-based system. The basic arrangement is illustrated in Figure 32.37.

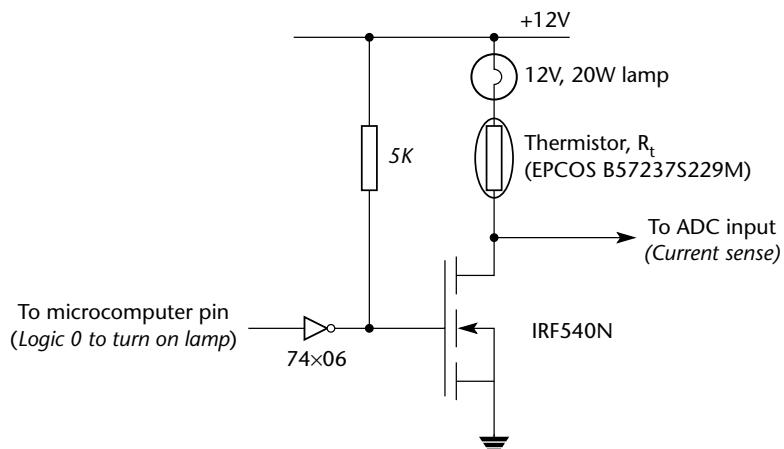


FIGURE 32.37 Detecting a blown bulb

With the bulb blown, the voltage is 0; with the bulb lit, a voltage of $\geq 0.1V$ can be measured.

Further reading

chapter 33

Pulse-width modulation

Introduction

Wait until it gets dark. Go to a room in your office or home where there is an ordinary (filament) light bulb. Switch the light on and off as rapidly as you can. The room will be dimly lit, with a flickering light. Pulse-width modulation (PWM) does exactly the same thing, at a much higher frequency. More specifically, PWM allows us to control (for example) the brightness of the light – without visible flickering – by switching the light on and off at a particular duty cycle.

PWM is an efficient basis for the control of high-power loads and is particularly widely used in applications such as DC (and AC) motor speed control.

There are two practical approaches to PWM signal generation in a time-triggered application:

- Several 8051-family microcontrollers provide hardware support for PWM on chip. This is generally easy to use. Where on-chip support is not available, specialist external hardware can provide cost-effective, high-frequency PWM switching.
- PWM outputs can be generated easily, at low frequencies, using software-only techniques.

These approaches are considered in the patterns **HARDWARE PWM** [page 808] and **SOFTWARE PWM** [page 831]. We also consider the post-processing that may be required to filter PWM-based signals, through the pattern **PWM SMOOTHER** [page 818] and creation of a high-frequency PWM output without using specialized hardware in the pattern **3-LEVEL PWM** [page 822].

HARDWARE PWM

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you create a high-frequency PWM output signal?

Background

A pulse-width modulated signal has many characteristics in common with the pulse-rate modulated signal discussed in **HARDWARE PRM** [page 742].

Like PRM, PWM is carried out by setting a port pin to Logic 1 for a period (X) and then to Logic 0 for another period (Y). We then repeat this process (Figure 33.1).

The (average) voltage at the port pin is determined by the duty cycle of the waveform. The duty cycle and some other common PWM features are defined as follows:

$$\text{Duty cycle (\%)} = \frac{x}{x + y} \times 100$$

Period = $x + y$, where x and y are in seconds.

$$\text{Frequency} = \frac{1}{x + y}, \text{ where } x \text{ and } y \text{ are in seconds.}$$

The key point to note is that the average voltage seen by the load is given by the duty cycle multiplied by the load voltage.

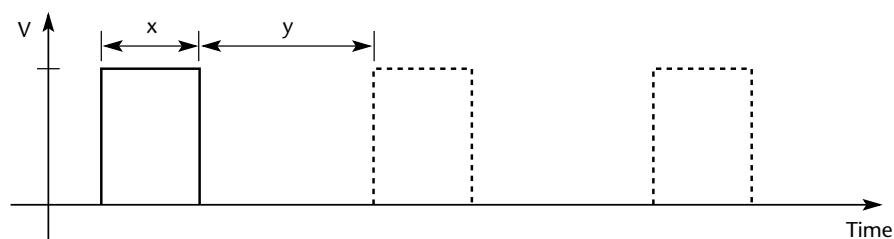


FIGURE 33.1 The underlying principles of PWM

Solution

We consider the following issues in this section:

- Creating PWM signals using on-chip hardware
- Creating PWM signals using external hardware
- Determining the required switching frequency
- Buffer and driver limitations
- Smoothing the PWM outputs

Creating PWM signals using on-chip hardware

We consider two specific examples of the generation of PWM signals using on-chip hardware in this section.

The Infineon c515c

In the Infineon c515c the PWM timer (which is based on Timer 2) is incremented with the machine clock, at one-sixth of the oscillator frequency. To generate a 16-bit PWM signal at the maximum (10 MHz) operational frequency, the timer is incremented every 300 ns (0.3 μ s). The total period is $2^{16} \times 0.3 \mu\text{s}$, which is 19,660.8 μs or 19.7 ms. The frequency is therefore approximately 50 Hz.

The Dallas 87C550

The Dallas 87C550 incorporates a high-speed, 4-channel, PWM hardware interface. The maximum PWM frequency (8-bit PWM, 12 MHz oscillator) is approximately 46 kHz.

Creating PWM signals using external hardware

If your microcontroller does not have hardware PWM support, external PWM chips are available: see, for example, the low-cost Dallas DS1050 family,¹⁶ which has a (5-bit) PWM output operating at up to 100 kHz.

Determining the required switching frequency: general guidelines

Determining, on paper, the PWM frequency you need to use for your application is generally not easy: it depends on a number of factors associated with the load and driver and it is generally necessary to perform some practical tests to determine the required frequencies: we discuss such tests later.

16. www.dalsemi.com

However, some basic guidelines can be given:

- The human eye can detect 'flicker' at rates of up to around 50 Hz. If your PWM hardware is controlling something like a bulb, the user may be able to see the flicker if the switching frequency is below this.
- The human auditory system has a range, at birth, of around 20 Hz to 20 kHz. If you switch high-power loads within this range, it is likely that the results will be audible to users, typically as a very annoying whine.

Determining the required switching frequency: practical studies

The only way of determining the required switching frequency in a practical application is to try a range of different frequencies.

One flexible way of doing this is to use a signal generator to drive the hardware and observe the results.

An effective alternative, that can often be carried out using your prototype system, is to apply the pattern **HARDWARE PRM** [page 742]. This allows the generation of square waves with a fixed (50%) duty cycle but variable period, at frequencies from less than 100 Hz to 3 MHz or more.

Buffer and driver limitations

The speed of the PWM hardware is, of course, not the only consideration in developing a PWM interface. For example, Figure 33.2 shows a circuit which attempts to perform PWM-based speed control of an AC motor.

This approach is doomed to failure, since the switch time of the EM relay will, typically, be of the order of 10 milliseconds; this restricts the PWM switching frequency to around 100 Hz. In many applications, this will not be adequate.

It is essential that you check the switching times of both switch hardware **and any associated buffer circuitry** when developing a PWM application.

Smoothing the PWM outputs

It is sometimes necessary to smooth PWM outputs, in order to remove high-frequency harmonics: see **PWM SMOOTHER** [page 818] for details.

Hardware resource implications

The main hardware resource implication is that, if you use the on-chip PWM, this is often based on Timer 2. As we have discussed, this timer is often used to drive the system scheduler (in a single-processor system).

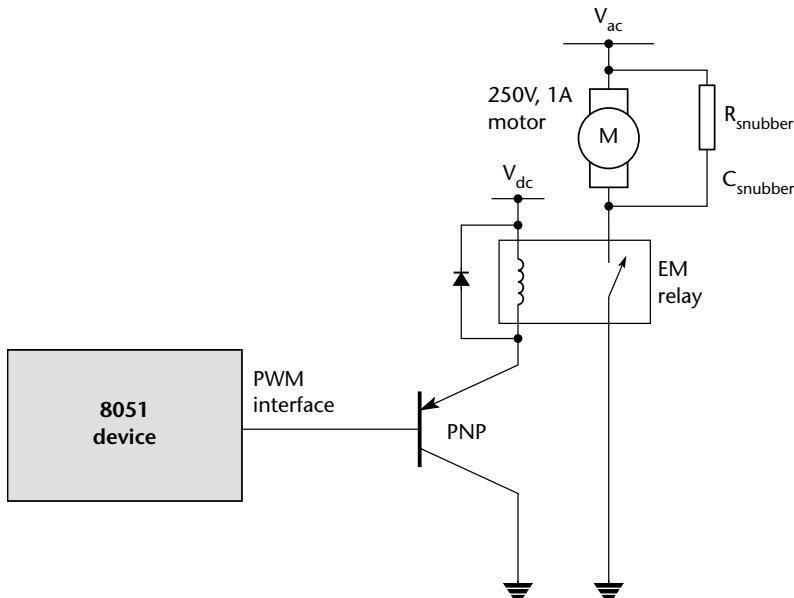


FIGURE 33.2 An attempt to control the speed of an AC motor using PWM control, using a driver based on an EM relay

[Note: This approach will fail, accept at very low switching frequencies.]

Reliability and safety implications

PWM routines are frequently used with high-power loads and, as usual with such loads, you need to ensure that your system starts in a safe state: for example, you need to ensure that, when the microcontroller is reset, the associated machinery begins at a slow speed, rather than a high speed. Refer to Chapters 7 and 8 for further discussion of this issue and some possible solutions.

More specifically, high-frequency PWM signals can be a source of EMI; this can be a particular problem for microcontroller-based embedded applications. Please see Ong *et al.* (2001) for further discussion of this issue.

Portability

A drawback with on-chip PWM hardware is that the features and implementation vary greatly over the 8051 family. In general, code written for external PWM hardware will be more easily ported across the 8051 family.

Overall strengths and weaknesses

- ☺ PWM has been used for many years as efficient means of providing a continuously varying output voltage to ‘slow’ external components, such as AC and

DC motors and large heating elements. It remains a very effective way of controlling such applications, without the need for complex external hardware, like digital-to-analogue converters (see **DAC OUTPUT** [page 841]).

- (?) PWM can be a source of EMI.
- (?) Not all 8051 devices have on-chip support for PWM.
- (?) Where on-chip PWM support is available, it is in no sense ‘standard’ and code written for one chip will not be particularly portable.

Related patterns and alternative solutions

See **SOFTWARE PWM** [page 831].

See **PWM SMOOTHER** [page 818].

See **DAC OUTPUT** [page 841].

Example: Using the c515c on-chip ADC and PWM hardware

In this example, we consider the control the brightness of a bulb under PWM control using a c515c (Listings 33.1 to 33.4).

Any suitable hardware interface may be used, with AC or DC bulbs: see Chapters 7 and 8 for suggestions.

Note that, by increasing the light intensity slowly when power is applied, the stress on the lamp filament can be greatly reduced, extending the bulb life. This can be easily achieved with PWM circuits and is particularly effective in setting where it is physically difficult or expensive to replace damaged bulbs.

```
/*-----*  
Port.H (v1.00)  
-----*  
  
'Port Header' (see Chap 10) for the project ADC_PWM (see Chap 33)  
-----*/  
  
// ----- ADC_515c.C -----  
  
// ADC reading from Pin 6.0  
  
// ----- PWM_515c.C -----  
  
// PWM output on Pin 1.1  
  
/*-----*  
--- END OF FILE ---  
-----*/
```

Listing 33.1 Part of an example illustrating the use of on-chip PWM hardware in the Infineon 515 family

```

/* -----
Main.c (v1.00)

-----
Simple 'ADC to PWM' example program (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);
    }
}

/* -----
--- END OF FILE ---
----- */

```

Listing 33.2 Part of an example illustrating the use of on-chip PWM hardware in the Infineon 515 family

```

/* -----
PWM_515c.c (v1.00)

-----
Rudimentary PWM library for 80c515c.

----- */

```

```
#include "Main.h"
#include "PWM_515c.h"

/*-----*
PWM_Init_T2()

Prepare on-chip PWM unit on the c515c.

*-----*/
void PWM_Init_T2(void)
{
    // ----- T2 Mode -----
    // Mode 1 = Timerfunction

    // Prescaler: Fcpu/6

    // ----- T2 reload mode selection -----
    // Mode 0 = auto-reload upon timer overflow
    // Preset the timer register with autoreload value ! 0x0000;
    TL2 = 0x00;
    TH2 = 0xFF;

    // ----- T2 general compare mode -----
    // Mode 0 for all channels
    T2CON |= 0x11;

    // ----- T2 general interrupts -----
    // timer 2 overflow interrupt is disabled
    ET2=0;
    // timer 2 external reload interrupt is disabled
    EXEN2=0;

    // ----- Compare/capture Channel 0 -----
    // disabled
    // Set Compare Register CRC on: 0xFF00;
    CRCL = 0x00;
    CRCH = 0xFF;

    // CC0/ext3 interrupt is disabled
    EX3=0;

    // ----- Compare/capture Channel 1 -----
    // Compare enabled
    // Set Compare Register CC1 on: 0xFF80;
    CCL1 = 0x80;
    CCH1 = 0xFF;
```

```

// CC1/ext4 interrupt is disabled
EX4=0;

// ----- Compare/capture Channel 2 -----
// disabled
// Set Compare Register CC2 on: 0x0000;
CCL2 = 0x00;
CCH2 = 0x00;
// CC2/ext5 interrupt is disabled
EX5=0;

// ----- Compare/capture Channel 3 -----
// disabled
// Set Compare Register CC3 on: 0x0000;
CCL3 = 0x000;
CCH3 = 0x000;

// CC3/ext6 interrupt is disabled
EX6=0;

// Set all above mentioned modes for channel 0-3
CCEN = 0x08;
}

/*-----*
 * PWM_Update_T2()
 *
 Update the PWM output value (capture/compare Channel 1)
 *
 Output is on Pin 1.1.
 *
 NOTE: Hardware will continue to produce this value (indefinitely),
 without software intervention, until the next update.
 *-----*/
void PWM_Update_T2(const tByte New_PWM_value)
{
    CCL1 = New_PWM_value;
}

/*-----*
 * --- END OF FILE ---
 *-----*/

```

Listing 33.3 Part of an example illustrating the use of on-chip PWM hardware in the Infineon 515 family

```
/*-----*
ADC_515c.c (v1.00)

-----
Simple, single-channel, 8-bit A-D (input) library for C515c

-*-----*/
#include "Main.H"
//#include "Bargraph.h"

// ----- Public variable definitions -----
// Stores the most recent ADC reading
tByte Analog_G;

/*-----*
AD_Init()

Set up the A-D converter.

-*-----*/
void AD_Init(void)
{
    // Select internally-triggered single conversion
    // Reading from P6.0 (single channel)
    ADEX = 0;    // Internal A/D trigger
    ADM = 0;    // Single conversion
    MX2 = MX1 = MX0 = 0; // Read from Channel 0 (Pin 6.0)

    // Leave ADCON1 at reset value: prescalar is /4
}

/*-----*
AD_Get_Sample()

Get a single data sample (8 bits) from the (10-bit) ADC.

-*-----*/
void AD_Get_Sample(void)
{
    tWord Time_out_loop = 1;

    // Take sample from A-D
    // Write (value not important) to ADDATL to start conversion
    ADDATL = 0x01;

    // Take sample from A-D (with simple loop time-out)
```

```
while ((BSY == 1) && (Time_out_loop != 0));
{
//    Time_out_loop++; // Disable for use in dScope...
}

if (!Time_out_loop)
{
    Analog_G = 0;
}
else
{
    // 10-bit A-D result is now available
    Analog_G = ADDATH; // Read only 8 most significant 8-bits of A-D
}
}

/*-----*-
----- END OF FILE -----
-*-----*/
```

Listing 33.4 Part of an example illustrating the use of on-chip PWM hardware in the Infineon 515 family

Further reading

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.

PWM SMOOTHING

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you filter the output of a PWM generator?

Background

Refer to **HARDWARE PWM** [page 808] and **A-A FILTER** [page 794] for relevant background information.

Solution

We consider, first, why it may be necessary to smooth PWM signals. We then consider how this can be achieved. Finally, we consider situations where the expense and complexity of PWM filtering may prove unnecessary.

Why do we need to smooth PWM signals?

In all the PWM signals we have considered in this book, the frequency of the waveform is held constant while the duty cycle varies (from 0% to 100%) according to the amplitude of the original signal. The resulting time-domain representation of such a signal is shown in Figure 33.3.

If we consider the frequency-domain representation of this signal shown in Figure 33.3, it is apparent that there is a strong peak at frequency $1/T$: this is the

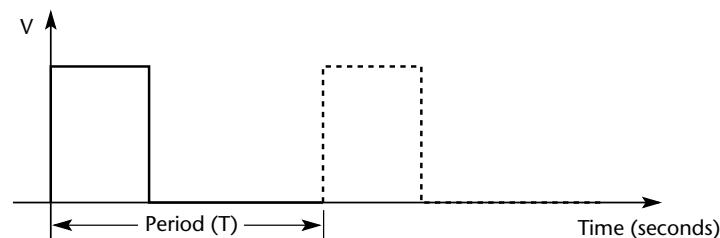


FIGURE 33.3 The representation of a PWM signal in the time domain

fundamental component of the PWM signal. In addition, there are harmonic peaks at frequency N/T (where N is an integer) (Figure 33.4).

The fundamental component of the PWM signal cannot be removed and we must therefore ensure that an appropriate frequency (for example, outside the audible range) is chosen. However, the harmonic components are, as far as we are concerned, an unwanted noise source; these must generally be removed, using an appropriate filter.

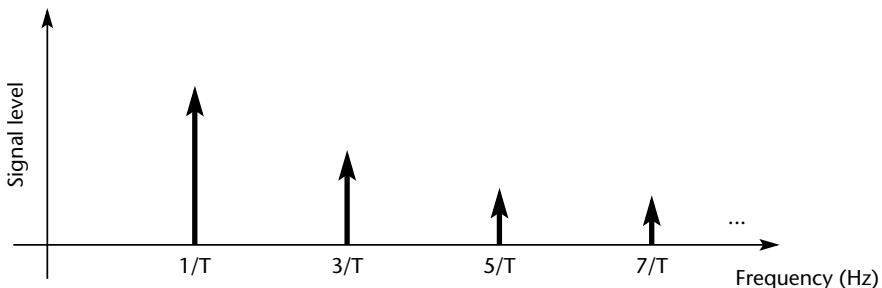


FIGURE 33.4 The representation of a PWM signal in the frequency domain

[Note: This type of representation might be produced, for example, using a spectrum analyzer.]

How do we remove PWM ‘noise’?

To remove PWM ‘noise’, the filter requirements are very similar to those discussed in **A-A FILTER** [page 794]: that is, the ideal PWM smoothing filter has a ‘brick wall’ specification and a cut off frequency of $1/T$ (Figure 33.5).

As we discussed in **A-A FILTER**, filters with such characteristics are not (economically) viable in most applications and instead the design in Figure 33.6 will prove adequate in most situations.

Refer to **A-A FILTER** for implementation details.

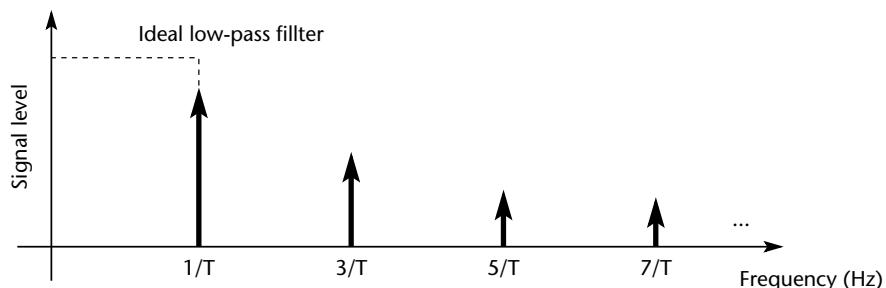


FIGURE 33.5 Removing ‘noise’ from PWM outputs using an ideal low-pass filter

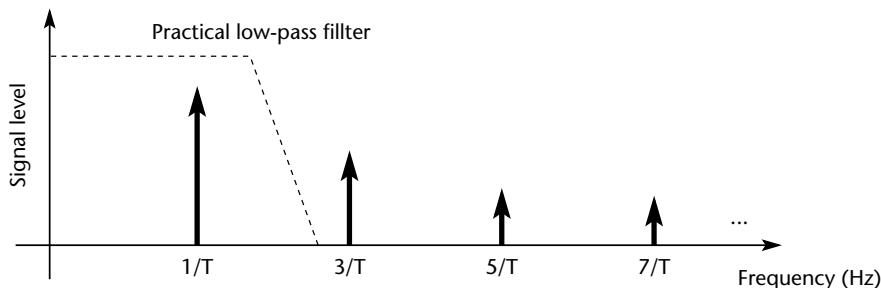


FIGURE 33.6 Removing 'noise' from PWM outputs using a more practical low-pass filter

Do you need a PWM filter?

Not all applications require the use of a PWM filter. For example, in motor control (a popular application area), PWM filtering is not generally necessary. Indeed, the main area in which filtering is required is in audio applications, such as speech generation. Note, however, that even in audio applications the need for filtering can be reduced if a high PWM frequency is employed; for example, at a 100 kHz fundamental frequency, harmonic components will be beyond the (human) auditory range (but not that of some other species).

Hardware resource implications

This hardware-only pattern has no particular implications for the use of (microcontroller) hardware resources.

Reliability and safety implications

Use of appropriate smoothing hardware can significantly reduce the levels of high-frequency EMI generated by PWM hardware; this can have important reliability implications.

Portability

This hardware-only pattern is highly portable.

Overall strengths and weaknesses

- ☺ Reduces audio and EMI interference in the PWM output.
- ☹ Adds to the system cost.

Related patterns and alternative solutions

See A-A FILTER [page 794].

Example: Creating a 5 kHz PWM smoothing filter

Refer to the 'speech-recognition' example in **A-A FILTER** [page 794] for implementation details of a suitable filter.

Further reading

- 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.
- Palacheria, A. (1997) 'Using PWM to generate analog output', Microchip Application Note AN538.

3-LEVEL PWM

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you create a high-frequency PWM output without using specialized hardware?

Background

See **HARDWARE PRM** [page 742] for discussions of pulse-rate modulation.

Solution

Suppose we wish to control the speed of a DC motor. If we switch the motor on and off at a slow rate via a single output pin on a microcontroller, we can view this as ‘1-bit PWM’, or ‘2-level PWM’. That is, we are controlling the motor with a pulse width of zero, or an infinite pulse width.

We can take this one significant stage further. As we saw in **HARDWARE PRM** [page 742], the majority of modern 8051 devices contain Timer 2; this timer can be programmed to generate a 50% duty cycle output at very high frequencies (up to 3 MHz, even with a 12 MHz / 12 oscillator cycle device).

Using this option, plus the ‘on’ or ‘off’ capabilities just mentioned, we can do the following with a minimal software load:

- Run the motor at full speed.
- Run the motor at half-speed, with a high-frequency PWM output.
- Stop the motor.

This behaviour is adequate for many applications and is very low cost. Of course, it may be applied equally well to other external devices, such as AC or DC lighting.

Hardware resource implications

This pattern requires the use of Timer 2. In single-processor applications, Timer 2 is often the most suitable means of driving the scheduler itself; as a result, use of this technique may have an impact on other parts of the application.

Note that, if a two-processor solution is possible (using a UART- or interrupt-based scheduler), then Timer 2 can be used to drive the scheduler in the Master node, leaving Timer 2 in the Slave node free for use as a PWM generator: see Part F for details of various multiprocessor solutions that may be appropriate here.

Reliability and safety implications

This technique is very reliable.

Portability

This technique may only be used with 8052-based devices: that is, those with an implementation of Timer 2 available. While most modern ‘8051s’ have such a timer, some popular devices – such as the Atmel Small 8051s – do not.

Overall strengths and weaknesses

- ☺ A simple, effective technique for creating PRM signals over a very wide frequency range.
- ☺ Imposes no measurable memory or CPU load.
- ☹ Requires exclusive use of Timer 2.

Related patterns and alternative solutions

The most directly comparable patterns are **SOFTWARE PWM** [page 831] and **HARDWARE PWM** [page 808].

As we noted in ‘Hardware resource implications’, it may be appropriate to use this technique in a Slave node in a multiprocessor application. Please refer to the various patterns in Part F for further information on this topic.

Example: Menu-driven 3-level PWM example

We wish to control the brightness of the small bulb, illustrated in the hardware schematic in Figure 33.7.

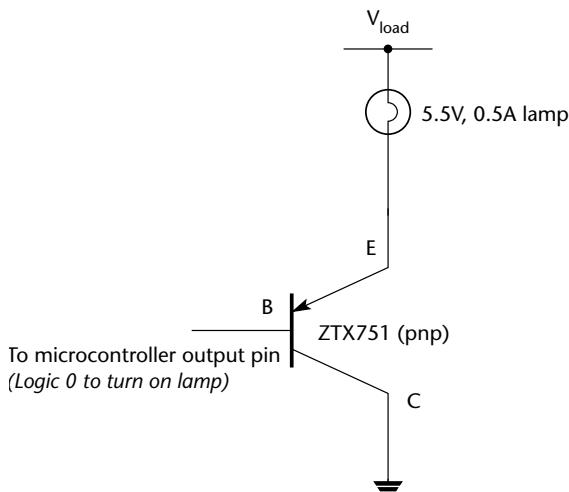


FIGURE 33.7 Controlling a small bulb using a BJT driver

A suitable small library for **3-LEVEL SOFTWARE PWM** is presented in Listings 33.5 to 33.7.

Figure 33.8 shows the example running in the Keil hardware simulator.

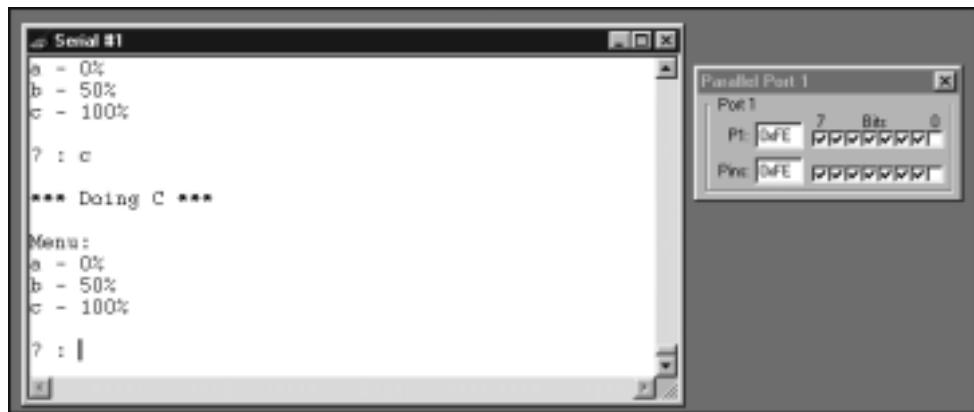


FIGURE 33.8 Illustrating the use of the example 3-level PWM library in the Keil hardware simulator

```
/*-----*
Port.H (v1.00)

-----
'Port Header' (see Chap 10) for the project PWM_3lev

-*-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P2

#endif
// ----- PC_IO.C -----
// Pins 3.0 and 3.1 used for RS-232 interface
// ----- 3_PWM.C -----
sbit PWM_pin = P1^0;
```

```
/*-----*  
---- END OF FILE -----  
-*-----*/
```

Listing 33.5 Part of an example illustrating the generation of 3-level PWM signals without specialized hardware

```
/*-----*  
Main.c (v1.00)  
-----  
  
Test program for menu-driven 3-level PWM library.  
  
Linker options:  
  
OVERLAY (main ~ (PWM_3_Command_Processor),  
SCH_Dispatch_Tasks ! (PWM_3_Command_Processor))  
-*-----*/  
  
#include "Main.h"  
  
#include "0_05_11g.h"  
#include "PC_I0_T1.h"  
#include "PWM_3.h"  
  
/* ..... */  
/* ..... */  
  
void main(void)  
{  
    // Set up the scheduler  
    SCH_Init_T0();  
  
    // Set baud rate to 9600: generic 8051 version  
    PC_LINK_Init_T1(9600);  
  
    // We have to schedule this task (10x - 100x a second)  
    //  
    // TIMING IS IN TICKS NOT MILLISECONDS (5 ms tick interval)  
    SCH_Add_Task(PWM_3_Command_Processor, 10, 2);  
  
    SCH_Start();  
  
    while(1)  
    {  
        // Displays error codes on P4 (see Sch51.C)
```

```

        SCH_Dispatch_Tasks();
    }
}

/*-----*
----- END OF FILE -----*
-----*/

```

Listing 33.6 Part of an example illustrating the generation of 3-level PWM signals without specialized hardware

```

/*-----*
----- PWM_3.C (v1.00)

-----*
----- Simple 3-level PWM example (see Chap 33).
----- Use 'Hyperterminal' (under Windows 95, 98, 2000) or similar
----- terminal emulator program on other operating systems.

----- Terminal options:
----- - Data bits      = 8
----- - Parity         = None
----- - Stop bits     = 1
----- - Flow control  = Xon / Xoff

----- */
----- */

#include "Main.h"
#include "Port.h"

#include "0_05_11g.h"
#include "PWM_3.h"
#include "PC_IO_T1.h"

// ----- Private constants -----
#define PWM_OFF 1
#define PWM_ON 0

/*-----*
----- PWM_3_Command_Processor()

----- This function is the main menu 'command processor' function.

----- Schedule this once every 10 ms (approx.).
----- */

```

```
-----*/
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();
    }

    // Check for user inputs
    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()

Display menu options on PC screen (via serial link)
- edit as required to meet the needs of your application.

-----*/
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()

Perform the required user task
- edit as required to match the needs of your application.
-----*/
```

```
-----*/  
void PWM_3_Perform_Task(char c)  
{  
    // Echo the menu option  
    PC_LINK_Write_Char_To_Buffer(c);  
    PC_LINK_Write_Char_To_Buffer('\n');  
  
    // Perform the task  
    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()  
  
Set PWM output to 0% duty cycle.  
-----*/  
void PWM_3_Set_000(void)  
{  
    PC_LINK_Write_String_To_Buffer("\n*** 0% ***\n\n");  
  
    TR2 = 0; // Stop Timer 2  
  
    PWM_pin = PWM_OFF;  
}
```

```

/* -----
   PWM_3_Set_050()

   Set PWM output to 50% duty cycle using Timer 2.

- */
void PWM_3_Set_050(void)
{
    {
        PC_LINK_Write_String_To_Buffer("\n*** 50% ***\n\n");

        T2CON &= 0xFD;    // Clear *only* C /T2 bit
        T2MOD |= 0x02;    // Set T20E bit (omit in basic 8052 clone)

        // Set at lowest frequency (~45Hz with 12MHz xtal)
        // - adjust as required (see PRM HARDWARE)
        TL2      = 0x00;    // Timer 2 low byte
        TH2      = 0x00;    // Timer 2 high byte
        RCAP2L   = 0x00;    // Timer 2 reload capture register, low byte
        RCAP2H   = 0x00;    // Timer 2 reload capture register, high byte

        ET2      = 0; // No interrupt.

        TR2      = 1; // Start Timer 2
    }

/* -----
   PWM_3_Set_100()

   Set PWM output to 100% duty cycle.

- */
void PWM_3_Set_100(void)
{
    {
        PC_LINK_Write_String_To_Buffer("\n*** Doing C ***\n\n");

        TR2      = 0; // Stop Timer 2

        PWM_pin = PWM_ON;
    }

/* -----
   ---- END OF FILE -----
- */

```

Listing 33.7 Part of an example illustrating the generation of 3-level PWM signals without specialized hardware

Further reading

Huang, H-W (2000) *Using the MCS-51 Microcontroller*, Oxford University Press, New York.

SOFTWARE PWM

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you create a low-frequency PWM output signal without using specialized PWM hardware?

Background

See **HARDWARE PWM** [page 808] for general background material on PWM.

See **SOFTWARE PRM** [page 748] for a similar solution.

Solution

The techniques discussed in **SOFTWARE PRM** [page 748] may be readily adapted to allow us to generate low-frequency pulse-width modulated signals. Listing 33.8 illustrates this.

```
void PWM_Soft_Update(void)
{
    // Have we reached the end of the current PWM cycle?
    if (++PWM_position_G >= PWM_PERIOD)
    {
        // Reset the PWM position counter
        PWM_position_G = 0;

        // Update the PWM control value
        PWM_G = PWM_new_G;

        // Set the PWM output to OFF
        PWM_pin = PWM_OFF;

        return;
    }

    // We are in a PWM cycle
    if (PWM_position_G < PWM_G)
```

```

    {
        PWM_pin = PWM_ON;
    }
else
{
    PWM_pin = PWM_OFF;
}
}
}

```

Listing 33.8 Implementing software PWM

The key to Listing 33.8 is the use of the variables `PWM_position_G`, `PWM_period_G` and `PWM_period_new_G`:

- `PWM_period_G` is the current PRM period. Note that if the update function is scheduled every millisecond, this period is in milliseconds. `PWM_period_G` is fixed during the program execution.
- `PWM_G` represents the current PWM duty cycle (see Figure 33.9).
- `PWM_new_G` is the next PWM duty cycle. This period may be varied by the user, as required. Note that the ‘new’ value is only copied to `PWM_G` at the end of a PWM cycle, to avoid noise.
- `PWM_position_G` is the current position in the PWM cycle. This is incremented by the update function. Again, the units are milliseconds if these conditions apply.

The link between these various variables and constants is illustrated in Figure 33.9.

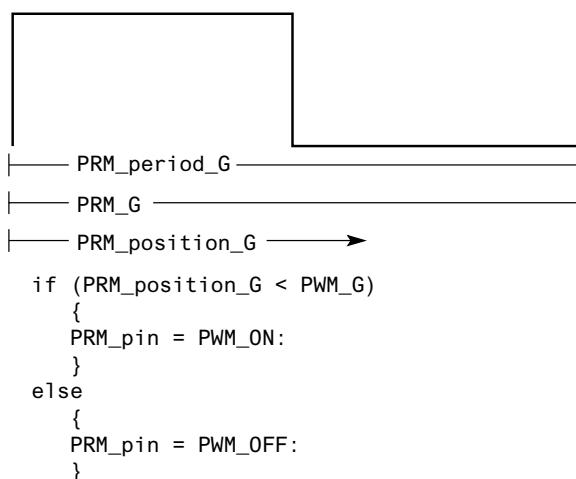


FIGURE 33.9 Implementing pulse-rate modulation in software

Using a 1 ms scheduler, the PWM frequency (Hz) and PWM resolution (%) we obtain are given by:

$$\text{Frequency}_{\text{PWM}} = \frac{1000}{2^N}$$

and:

$$\text{Resolution}_{\text{PWM}} = \frac{1}{2^N} \times 100\%$$

where N is the number of PWM bits you use.

For example, 5-bit PWM allows you to control the output to a resolution of approximately 3%, at a frequency of approximately 31 Hz.

Hardware resource implications

SOFTWARE PWM has no significant hardware resource implications.

Reliability and safety implications

When working with PWM, you may be switching high-power loads and / or mains voltages. You need to take care with start-up states, to ensure that any equipment you are controlling is not going to cause damage or injury when the application starts.

Portability

SOFTWARE PWM uses only core 8051 features: it may be easily used throughout the 8051 family.

Overall strengths and weaknesses

- ☺ PWM has been used for many years as efficient means of providing a continuously varying output voltage to ‘slow’ external components, such as AC and DC motors and large heating elements. It remains a very effective way of controlling such applications, without the need for digital-to-analogue converters.
- ☺ **SOFTWARE PWM** does not impose a heavy CPU load.
- ☺ **SOFTWARE PWM** can only operate at comparatively low frequencies. For example, at a 1 ms tick rate and 8-bit PWM, the PWM frequency is 1000 / 256: that is, less than 4 Hz. Typical hardware PWM units operate at up to 100 kHz. For many applications, high-speed PWM is an important consideration: please see **HARDWARE PWM** [page 808] for further information on this issue.

Related patterns and alternative solutions

See **HARDWARE PWM** [page 808] for one alternative.

An alternative implementation of the present pattern is also worth highlighting. Consider the code shown in Listing 33.9.

```
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;
        }

        // Appropriate delay here
        PWM_Delay();
    }
}
```

Listing 33.9 Attempting to increase the frequency of software-only PWM signals

In this solution, we assume that we are generating an entire PWM cycle every time we call the task: this seems to offer the possibility of generating faster PWM signals. For example, at a 0.25 ms tick intervals, we can generate a 4 kHz PWM frequency, which will be adequate for many applications. Moreover, we can generate any number of 'PWM_INCREMENTS': that is, any required bit rate.

However, this solution has two problems. The first is that the CPU is tied up, completely, in the PWM signal generation. This is not insurmountable but might mean, for example, that the solution could only be implemented on a Slave node in a multi-processor system.

The second problem is more serious and relates to the delays required within the loop. For example, at a 1 ms tick interval, the generation of 5-bit PWM outputs requires the generation of delays of precisely 3.125 µs. This is very difficult to achieve, as we discussed in Chapter 11. At higher bit rates and tick frequencies, the required delays are very much shorter and cannot be obtained in most implementations.

Example: Controlling the brightness of a bulb

We again wish to control the brightness of the small bulb, illustrated in the hardware schematic in Figure 33.10.

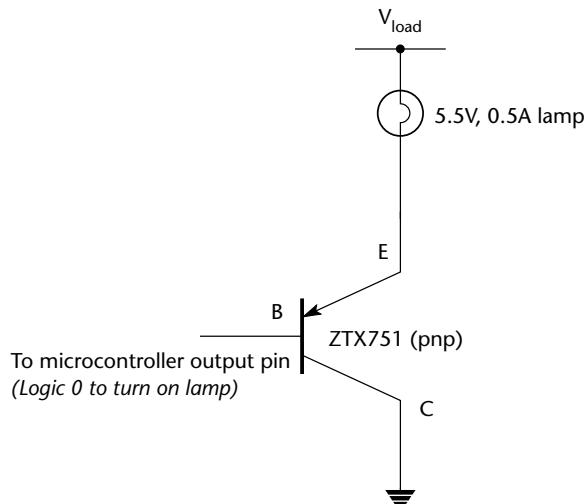


FIGURE 33.10 Controlling the brightness of a bulb

A suitable small library for software PWM is presented in Listings 33.10 to 33.12.

```
/*-----*
Port.H (v1.00)
-----
'Port Header' (see Chap 10) for the project PWM_Soft (see Chap 33)
*-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1
#endif
// ----- PWM_Soft.C -----
sbit PWM_pin = P1^0;
```

```
/*-----*  
--- END OF FILE ---  
-----*/
```

Listing 33.10 Part of an example illustrating the generation of PWM signals without specialized hardware

```
/*-----*  
-----*  
Main.C (v1.00)  
-----*  
  
Test program for 'Software PWM' library.  
See Chapter 33 for details.  
-----*-----*/  
  
#include "Main.h"  
#include "2_01_12g.h"  
#include "PWM_Soft.h"  
  
/* ..... */  
/* ..... */  
  
void main()  
{  
    SCH_Init_T2();  
  
    PWM_Soft_Init();  
  
    // Call every millisecond to update PWM output  
    SCH_Add_Task(PWM_Soft_Update, 0, 1);  
  
    // Call every five seconds to change PWM control value  
    SCH_Add_Task(PWM_Soft_Test, 10, 3000);  
  
    SCH_Start();  
  
    while(1)  
    {  
        SCH_Dispatch_Tasks();  
    }  
}
```

```
/*-----*  
----- END OF FILE -----  
-----*/
```

Listing 33.11 Part of an example illustrating the generation of PWM signals without specialized hardware

```
/*-----*  
-----  
----- PWM_SOFT.C (v1.00)  
-----  
-----  
----- Simple 'Software PWM' library.  
----- See Chapter 32 for details.  
-----*/  
  
#include "Main.h"  
#include "Port.h"  
  
#include "2_01_12g.h"  
#include "PWM_Soft.h"  
  
// ----- Public variable definitions -----  
  
// Set this variable to the required PWM value  
tByte PWM_new_G;  
  
// ----- Private constants -----  
  
#define PWM_ON 0  
#define PWM_OFF 1  
  
// ----- Private variables -----  
  
// The PWM counter  
static tWord PWM_position_G;  
static tByte PWM_G;  
  
/*-----*  
-----  
----- PWM_Soft_Test()  
----- To test the PWM library, this function is called once every  
----- five seconds, to update the PWM settings.  
-----  
----- NOTE:  
----- In a real application, this function will be replaced by a user-  
----- defined function for setting brightness, speed, etc.
```

```
- *-----*/  
void PWM_Soft_Test(void)  
{  
    if (++PWM_new_G >= PWM_PERIOD)  
    {  
        PWM_new_G = 0;  
    }  
}  
/*-----*/  
  
PWM_Soft_Init()  
  
    Prepares some of the key PWM variables.  
- *-----*/  
void PWM_Soft_Init(void)  
{  
    // Init the main variable  
    PWM_new_G = 0;  
    PWM_position_G = 0;  
    PWM_pin = PWM_OFF;  
}  
/*-----*/  
  
PWM_Soft_Update()  
  
    The key PWM function. Schedule as rapidly as possible.  
- *-----*/  
void PWM_Soft_Update(void)  
{  
    // Have we reached the end of the current PWM cycle?  
    if (++PWM_position_G >= PWM_PERIOD)  
    {  
        // Reset the PWM position counter  
        PWM_position_G = 0;  
  
        // Update the PWM control value  
        PWM_G = PWM_new_G;  
  
        // Set the PWM output to OFF  
        PWM_pin = PWM_OFF;  
  
        return;  
    }  
}
```

```
// We are in a PWM cycle
if (PWM_position_G < PWM_G)
{
    {
        PWM_pin = PWM_ON;
    }
}
else
{
    {
        PWM_pin = PWM_OFF;
    }
}

/* -----
   ----- END OF FILE -----
   ----- */
```

Listing 33.12 Part of an example illustrating the generation of PWM signals without specialized hardware

Further reading

Using digital-to-analog converters (DACs)

Introduction

In the 1980s, if you wanted to create an analog signal from a microcontroller, you would probably have used a ‘digital-to-analog converter’ (DAC). More recently, as the operating frequencies for digital hardware have grown, pulse-width modulation (PWM) has become a more cost-effective means of creating an analog signal in many circumstances. As a result, use of DACs in areas such as motor control and robotics has become much less common.

However, there are two main sets of circumstances in which use of a DAC is still cost-effective: in high-frequency / high bit-rate applications (particularly audio applications) and in process control.

We consider how to use a DAC to generate analog outputs in this chapter. In doing so we present the following patterns:

- **DAC OUTPUT** [page 841]
- **DAC SMOOTH** [page 853]
- **DAC DRIVER** [page 857]

DAC OUTPUT

Context

You are developing a ‘hard’ or ‘soft’ real-time application, for either a desktop or embedded environment (see Chapter 1 for definitions of these various terms).

Problem

How do you use a digital-to-analog converter (DAC) to generate analog signals?

Background

Suppose you are developing an intruder alarm system for a home environment. In the event of a break-in, we generally expect simply to sound an alarm (ring a bell) to notify the neighbours. This is adequate: the intruder alarm is only intended to detect one kind of problem.

However, suppose you are developing a safety monitoring system for an industrial plant. Various problems may arise: different forms of gas leak, nuclear materials leak, fire, intruder and so forth. In some situations, those in the plant need to take different action (for example, donning different kinds of protective clothing), depending on the type of threat they face.

How do we tell those in the plant what kind of problem has occurred?

One appropriate solution might be to use digitized messages that describe the problem and tell users what to do. For example: ‘Warning, a leak of Contaminant 356 has been detected at a level of 345 parts per million. Please use your respirator now and leave the building via your designated escape route.’

To play back this type of message, we will generally require a digital-to-analog converter (DAC).

DACs have an important role, not only in the playback of digitized warnings, but also in other control applications. We consider how to use DACs in this pattern.

Solution

There are several key design decisions that must be made when generating an analog output:

- 1 You need to determine the required sample rate.
- 2 You need to determine the required bit rate (DAC resolution).
- 3 You need to select an 8051 family member with an appropriate DAC or, if necessary, add an external DAC.

- 4 You may need to shape the frequency response of the output signal.
- 5 You need to use an appropriate software architecture.

We now deal with each of these points in turn.

Determining the required sample rate

Refer to **SEQUENTIAL ADC** [page 782] for discussions on sample rates.

Determining the required bit rate

Refer to **ONE-SHOT ADC** [page 757] for discussions on bit rates.

8051 microcontrollers with on-chip DACs

While, as we saw in **ONE-SHOT ADC** [page 757], many 8051 microcontrollers contain on-chip ADCs, the number of devices with on-chip DACs is very limited.

Some examples of the available DAC components provided on two recent 8051 devices follow.

Analog Devices AD μ C812

Adapted from the Analog Devices¹⁷ data sheet:

The AD μ C812 incorporates two 12-bit voltage-mode DACs On-Chip.

DAC operation is controlled via three special function registers. In normal mode of operation, each DAC is updated when the low DAC byte (DACxL) SFR is written. Both DACs can be updated simultaneously using the SYNC bit in the DACCON SFR. The DAC's can operate in 12 or 8-bit modes and have programmable output ranges of 0 -> 2.5V or 0 ->VDD.

Cygnal C8051F000

Adapted from the Cygnal¹⁸ C8051F000 data sheet:

The C8051F000 MCU family has two 12-bit voltage-mode DACs (DAC0, DAC1).

Each DAC has an output swing of 0V to VREF-1LSB for a corresponding input code range of 0x000 to 0xFFFF. Using DAC0 as an example, the 12-bit data word is written to the low byte (DAC0L) and high byte (DAC0H) data registers. Data is latched into DAC0 after a write to the corresponding DAC0H register, so the write sequence should be DAC0L followed by DAC0H if the full 12-bit resolution is required. The DAC can be used in 8-bit mode by initializing DAC0L to the desired value (typically 0x00), and writing data to only DAC0H. DAC0 Control Register (DAC0CN) provides a means to enable/disable DAC0 and to modify its input data formatting.

The DAC0 enable/disable function is controlled by the DAC0EN bit (DAC0CN.7). Writing a 1 to DAC0EN enables DAC0 while writing a 0 to DAC0EN disables DAC0. While disabled, the

17. www.analog.com

18. www.cygnal.com

output of DAC0 is maintained in a high-impedance state, and the DAC0 supply current falls to 1uA or less.

In some instances, input data should be shifted prior to a DAC0 write operation to properly justify data within the DAC input registers. This action would typically require one or more load and shift operations, adding software overhead and slowing DAC throughput. To alleviate this problem, the data-formatting feature provides a means for the user to program the orientation of the DAC0 data word within data registers DAC0H and DAC0L. The three DAC0DF bits (DAC0CN.[2:0]) allow the user to specify one of five data word orientations.

DAC1 is functionally the same as DAC0 described above.

Using an external voltage-mode DAC

In general, while the use of an internal DAC will result in increased reliability, smaller system size and lower system cost, this is not always possible.

As with ADCs, both parallel and serial (voltage-mode) DACs are available. For example, in terms of parallel components, Analog Devices¹⁹ produce the parallel AD7245a (12-bit), AD7248a (12-bit) and AD7801 (8-bit), while Maxim²⁰ produce the Max5480 (8-bit). In serial devices, Maxim (again) produce, for example, the Max517 with an I²C interface and the Max541 with an SPI interface: refer to Part E for discussion of the use of these interfaces in a time-triggered environment.

Because we will generally be using DACs in circumstances where a high bit-rate and sample frequency is required, the parallel solution will often be the most practical solution.

Using an external current-mode DAC or a transconductance amplifier

As we discussed in **ONE-SHOT ADC** [page 757], there are circumstances, particularly in monitoring or process control, when the use of current-mode components (sensors or actuators) may be more appropriate than the use of voltage-mode devices.

To generate analog current signals from a microcontroller, we have two main options:

- Use a current-mode DAC
- Use a voltage-mode DAC *and* a voltage-to-current converter (often referred to as a *transconductance amplifier*)

The second option may be particularly attractive if your microcontroller has an on-chip DAC.

An example of a suitable DAC component is the Analog Devices²¹ AD421, a DAC designed specifically for 4 to 20mA current loops and which is powered by the loop itself.

An example of a transconductance amplifier is the XTR 110 from Burr Brown.²² This can be used, for example, to convert a 1–5V analog voltage (from, say, a microcontroller

19. www.analog.com

20. www.maxim-ic.com

21. www.analog.com

22. www.burr-brown.com

voltage-mode DAC output) into the 4–20mA range required in process-control applications. As such, it can be a useful component in some ‘intelligent’ process sensors.

Shaping the frequency response

Use of a DAC will introduce ‘noise’ (through aliasing effects) and frequency distortions. In most cases, the aliasing effects, at least, must be eliminated: see **DAC SMOOTHER** [page 853] for further details.

General implications for the software architecture

The use of a DAC at high frequencies (10 kHz or 16 kHz) will have a major impact on the overall architecture of your application. For example, even at 10 kHz, you may require a 0.1 ms tick interval. This imposes a substantial load on a basic 8051 device.

In general, only 8051 devices which operate fewer than 12 clock cycles per instruction can provide these levels of performance. Use of recent devices such as the Dallas 89C420 (a ‘Standard 8051’ with 1 clock cycle per instruction, operating at up to 50 MHz: see Chapter 3) can make it practical to operate at 16 kHz (0.0625 ms tick interval).

Hardware resource implications

Use of the internal ADC will generally mean that at least one input pin is unavailable for other purposes; use of an external ADC will require the use of larger numbers of port pins.

Use of the ADC may also have an impact on the power consumption of your microcontroller.

Reliability and safety implications

In general, use of on-chip DACs is likely to improve the system reliability (compared with an off-chip solution), since the hardware (and, possibly, software) complexity, number of soldered joints, etc., are all greatly reduced.

More specifically, use of a DAC with a long conversion time may introduce delays that will impact on the general performance of signal-processing applications and which may impact on the stability of control applications. Make sure the speed of the DAC is an appropriate match for your intended application.

Portability

All DAC components vary. The principles and basic techniques are portable but the details will always be heavily hardware dependent.

Overall strengths and weaknesses

☺ DAC outputs are essential in many applications and are generally easy to use.

- (?) Microcontrollers with on-board DACs are comparatively rare and more expensive than those without such facilities.

Related patterns and alternative solutions

In many cases, PWM outputs (see Chapter 33) provide a low-cost and effective alternative to the use of DACs: see **HARDWARE PWM** [page 808].

See **DAC SMOOTH** [page 853] for techniques suitable for removing ‘noise’ (introduced through aliasing effects) and frequency distortions introduced by the use of DACs.

See **DAC DRIVER** [page 857] for information about the hardware needed to drive high-power loads, such as loudspeakers or DC motors.

Example: Speech playback using a 12-bit parallel DAC

Here we consider how we can use a 12-bit parallel DAC to play back a speech sample at a 10 kHz sample rate.

Figure 34.1 shows the speech waveform.

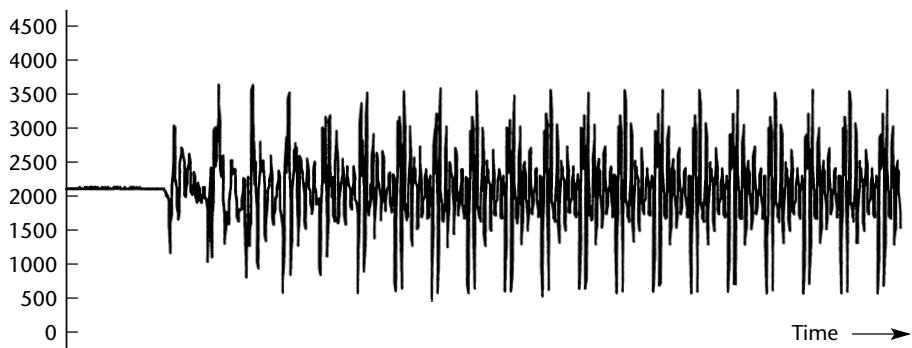


FIGURE 34.1 An example of a fragment of speech

Figure 34.2 shows the basic hardware used. Note that the necessary smoothing and amplification components will be discussed in **DAC SMOOTH** [page 853] and **DAC DRIVER** [page 857].

The key code listings are given in Listings 34.1 to 34.4.

```
/* ----- * -  
Port.H (v1.00)  
-----  
'Port Header' (see Chap 10) for the project Play_DAC  
* ----- */
```

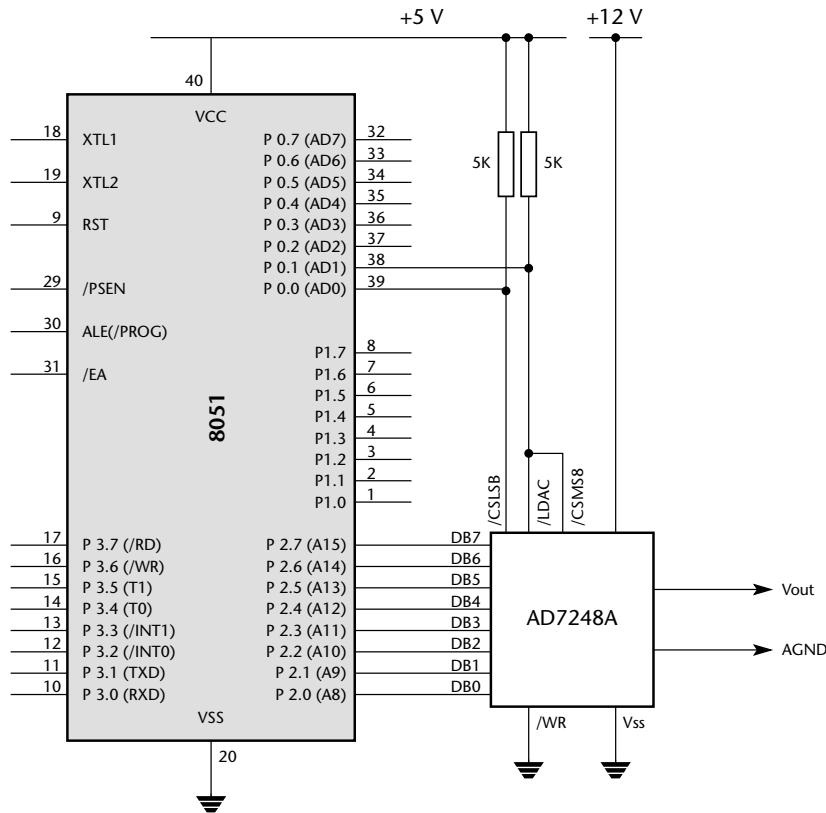


FIGURE 34.2 Using an AD7148A for speech playback

```

// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS

#ifdef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#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 -----

```

```
// Connect single push-button switch on this pin (to gnd)
// - debounced in software
sbit Sw_pin = P3^3; // The switch pin

/* -----
----- END OF FILE -----
----- */
```

Listing 34.1 Part of an example illustrating the control of an external 12-bit (parallel) DAC

```
/* -----
----- Main.c (v1.00)

-----
Speech playback example, uses hybrid scheduler.

Required linker options (see Chapter 13 for details):
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)
{
    //
    // Set up the scheduler
    hSCH_Init_T2();

    //
    // Set up the switch pin
    SWITCH_Init();

    //
    // Add the 'switch' task (check every 200 ms)
    // THIS IS A PRE-EMPTIVE TASK
    hSCH_Add_Task(SWITCH_Update, 0, 200, 0);
```

```

// NOTE:
// 'Playback' task is added by the SWITCH_Update task
// (as requested by user)
// 'Playback' is CO-OPERATIVE
// *** NOTE REQUIRED LINKER OPTIONS (see above) ***

// Start the scheduler
hSCH_Start();

while(1)
{
    hSCH_Dispatch_Tasks();
}

/*-----*
----- END OF FILE -----
-*-----*/

```

Listing 34.2 Part of an example illustrating the control of an external 12-bit (parallel) DAC

```

/*-----*
Playback.C (v1.00)

-----*/

Library of functions to allow playback of stored speech sample
from on-chip ROM.

Assumes presence of AD7248A DAC: see text for hardware connections.

Play back continuously while switch is depressed.

Data are replayed at 10 kHz, 12-bit resolution.

-*-----*/
#include "Main.h"
#include "Port.h"

#include "Playback.h"

// ----- Public constants -----
// The speech data we are going to play
extern const tWord code BA_12_BIT_10KHZ_G[3500];

// ----- Public variable declarations -----

```

```
extern bit Sw_pressed_G;

// ----- Public variable definitions -----
bit SPEECH_PLAYBACK_Playing_G = 0;

// ----- Private variables -----
static bit LED_state_G;

// ----- Private constants -----
#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()

The main update function for the playback software.

This will usually be scheduled, as required, as a one-shot
(co-operative) task.

Task duration is approximately 350 milliseconds.

User can abort at any time by releasing the switch.

-*-----*/
void SPEECH_PLAYBACK_Update(void)
{
    int Sample;

    SPEECH_PLAYBACK_Playing_G = 0;

    // Configure Timer 0 as a 16-bit timer
    TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
    TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)
    ET0 = 0; // No interrupts

    // Playback at ~ 10 kHz
    for (Sample = 0; Sample < 3500; Sample++)
    {
        // Avoid multiple calls to this function
        SPEECH_PLAYBACK_Playing_G = 1;

        // Play a sample
        SPEECH_PLAYBACK_Play_Sample(BA_12_BIT_10KHZ_G[Sample]);

        // Delay ~0.1 ms (to give 10 kHz sample rate)
        TR0 = 0;
        TH0 = T_100micros_H;
```

```

    TL0 = T_100micros_L;
    TF0 = 0; // Clear flag
    TR0 = 1; // Start timer
    while (!TF0);
    TR0 = 0;

    if (!Sw_pressed_G)
    {
        break; // Abort loop if user releases switch
    }
}

// Set flag to 0 as we leave this function
SPEECH_PLAYBACK_Playing_G = 0;
}

/* -----
SPEECH_PLAYBACK_Play_Sample()

Send 12-bit data sample to AD7248A ADC.

----- */
void SPEECH_PLAYBACK_Play_Sample(const tWord SAMPLE)
{
    // Samples are 12-bits, in 16-bit (tWord) value

    // Send lower 8 bits first
    tByte Data_8bit = (tByte) (SAMPLE & 0x00FF);

    SPEECH_Port = Data_8bit;
    SPEECH_CSLSB_pin = 0;
    SPEECH_CSMSB_pin = 1;

    // Now the upper 4 bits
    Data_8bit = (tByte) ((SAMPLE >> 8) & (0x0F));

    SPEECH_Port = Data_8bit;
    SPEECH_CSLSB_pin = 1;
    SPEECH_CSMSB_pin = 0;
}

/* -----
---- END OF FILE -----
----- */

```

Listing 34.3 Part of an example illustrating the control of an external 12-bit (parallel) DAC

```
/*-----*  
SWIT_PLY.C (v1.00)  
-----  
Simple switch interface code, with software debounce.  
Controls DAC speech playback.  
-----*/  
  
#include "Main.h"  
#include "Port.h"  
  
#include "Swit_Ply.h"  
#include "Playback.h"  
#include "2_01_12h.h"  
  
// ----- Public variable definitions -----  
  
bit Sw_pressed_G = 0; // The current switch status  
  
// ----- Public variable declarations -----  
  
extern bit SPEECH_PLAYBACK_Playing_G; // Current playback status  
  
// ----- Private constants -----  
  
// Allows NO or NC switch to be used (or other wiring variations)  
#define SW_PRESSED (0)  
  
// SW_THRES must be > 1 for correct debounce behaviour  
#define SW_THRES (3)  
  
/*-----*  
SWITCH_Init()  
Initialization function for the switch library.  
-----*/  
void SWITCH_Init(void)  
{  
    Sw_pin = 1; // Use this pin for input  
}  
  
/*-----*  
SWITCH_Update()  
This is the main switch function.  
It should be scheduled every 50 - 500 ms.  
-----*
```

```

-----*/
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; // Switch is pressed...

            // We add the 'playback' task to the scheduler here
            // (after checking if it is already running)
            // Add the 'playback' task (duration 10 seconds)
            // THIS IS A CO-OPERATIVE (one-shot) TASK
            if (SPEECH_PLAYBACK_Playing_G == 0)
            {
                hSCH_Add_Task(SPEECH_PLAYBACK_Update, 0, 0, 1);
            }
        }

        return;
    }

    // Switch pressed, but not yet for long enough
    Sw_pressed_G = 0;
    return;
}

// Switch not pressed - reset the count
Duration = 0;
Sw_pressed_G = 0; // Switch not pressed...
}

/*
----- END OF FILE -----
*/

```

Listing 34.4 Part of an example illustrating the control of an external 12-bit (parallel) DAC

Further reading

DAC SMOOTHING

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you reduce 'conversion noise' from the output of a digital-to-analog converter?

Background

See **DAC OUTPUT** [page 841] for background information.

Solution

Suppose we wish to create a high-quality digital communication system, to be used in a hydrofoil. Specifically, we will assume that the hydrofoil contains a computer network intended for non-critical operations, such as monitoring the passenger cabin temperature; this network has spare bandwidth, which we intend to utilize to provide the means of conveying messages from the crew to the passengers (Figure 34.3).

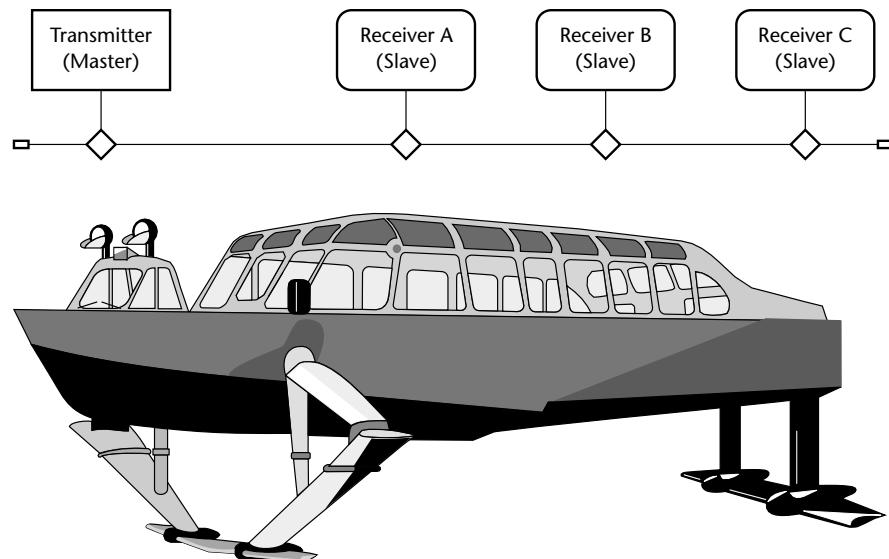


FIGURE 34.3 A digital communication system designed for use in a hydrofoil

To play back the speech signals (transmitted digitally over the network), we will use the hardware shown in Figure 34.4. We assume that this hardware is repeated on each of the Slave nodes.

Unfortunately, the quality of the speech produced from this system will be very poor, for two reasons.

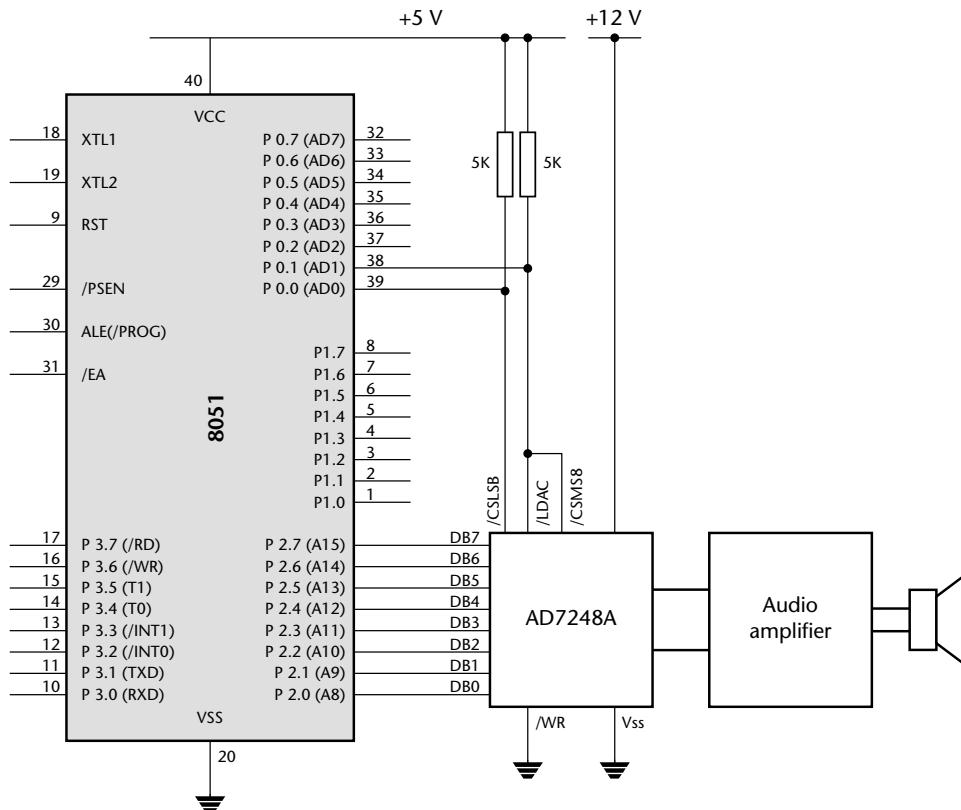


FIGURE 34.4 Hardware to be used for speech playback

The first (and main) cause of poor quality will be aliasing effects: see **ONE-SHOT ADC** [page 757] for discussion of this issue. The impact of aliasing effects can be removed using a low-pass filter with a cutoff frequency at half the sample rate. The implementation of suitable filters is discussed in detail in **A-A FILTER** [page 794].

The second cause of poor playback quality is that the DAC output is quantized and that, rather than reproducing the waveform shown in Figure 34.5 (top), a waveform similar to that shown in Figure 34.5 (bottom) will be generated.

The ‘staircase’ nature of the DAC output introduces distortions which vary with frequency; to remove these effects, we need to employ a technique known as ‘sinc compensation’; this involves applying a comparatively complex filtering operation (a sinc filter) to the DAC signal. The technical details of this compensation are beyond the scope of this text (see, for example, Smith, 1999, for details).

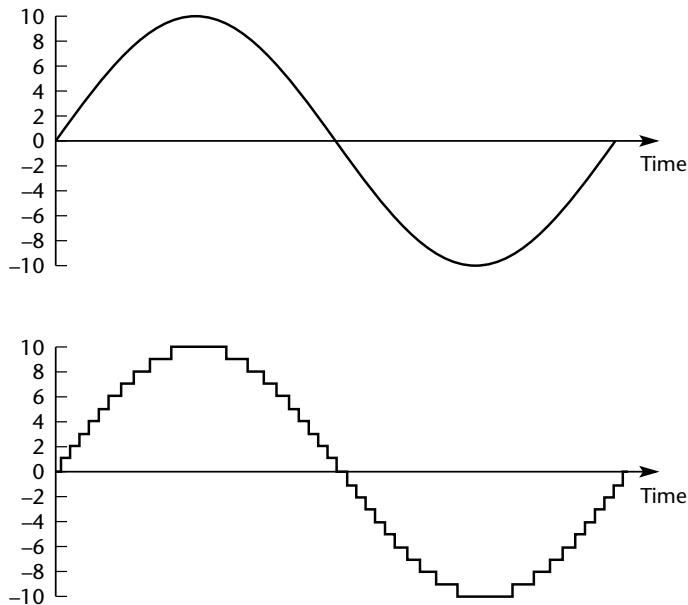


FIGURE 34.5 A quantized sine wave

Overall, for the highest quality reproduction we require the arrangement shown in Figure 34.6. However, in most applications, use of the low-pass filter alone (with a cut-off at half the sampling frequency), will provide adequate performance.

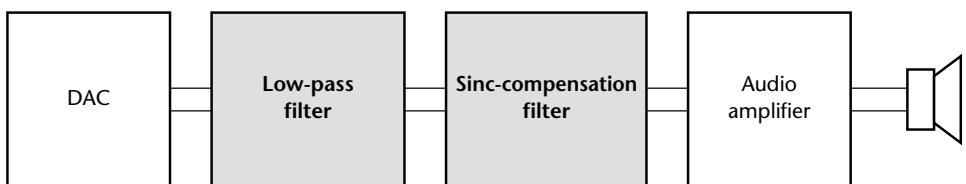


FIGURE 34.6 Producing high-quality audio signals from a DAC

Hardware resource implications

This hardware-only pattern has no particular implications for the use of (microcontroller) hardware resources.

Reliability and safety implications

Use of appropriate smoothing hardware may help reduce the levels of high-frequency EMI generated by DAC hardware in some circumstances; this can help improve system reliability.

Portability

This hardware-only pattern is highly portable.

Overall strengths and weaknesses

- 😊 Reduces conversion noise in the DAC output.
- 😢 Adds to the system cost.

Related patterns and alternative solutions

See A-A FILTER [page 794].

Example: Speech playback using a 12-bit parallel DAC

Consider the speech playback example presented on page 845 in **DAC OUTPUT**.

To complete the hardware for this example, we require at least two further components: appropriate filters and some form of amplifier.

We consider here the required filters.

As noted in this pattern, removal of frequency components at frequencies about $F_s/2$ (where F_s is the sampling frequency) is essential to avoid aliasing effects. Here, we will ignore the use of sinc compensation.

A suitable 5 kHz filter is given in Figure 34.7. Please refer to **A-A FILTER** [page 794] for further details of the design process required to create this (op-amp) filter.

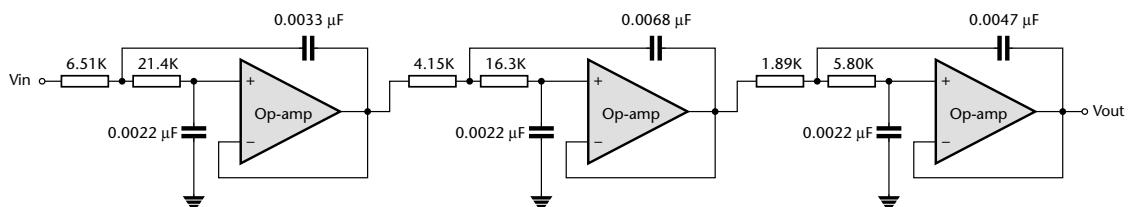


FIGURE 34.7 An 5 kHz low-pass filter suitable for use with the speech playback example

Further reading

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]

DAC DRIVER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you convert the output from a voltage-mode ADC into a form suitable for driving high-power loads?

Background

See **DAC OUTPUT** [page 841] for background details on DACs.

Solution

Voltage-mode DACs are low-power devices; for example, the Maxim Max541 is a 16-bit DAC with an ability to handle a maximum (analog) current of the 50 mA.²³ Clearly if we wish to use such a device to drive, say, a DC motor, we require some form of buffer circuit.

As we saw in Chapter 7, when we considered the switching of DC loads, there are two main alternatives when driving high-power loads: use a solution based on discrete components (which generally means BJTs) or an IC-based solution (which generally means a power op-amp).

We illustrate these solutions in the examples that follow.

Hardware resource implications

Use of this pattern has no implications for the hardware resources (e.g. CPU time or memory) on the microcontroller itself.

Reliability and safety implications

There are no specific reliability or safety implications.

Portability

This hardware-only pattern can be used with any microcontroller.

23. This figure is higher than many common DAC chips

Overall strengths and weaknesses

- ☺ Simple and effective.
- ☹ May require use of a two-rail power supply if op-amps are employed.

Related patterns and alternative solutions

See **DAC OUTPUT** [page 841].

See **DAC SMOOTH** [page 853].

Example: Driving a speaker using discrete components

Figure 34.8 illustrates a BJT-based amplifier circuit. Specifically, the two transistors are arranged as a ‘Darlington pair’, to increase the circuit gain. This is typically necessary because the gain of power transistors (such as the 2N3055) is particularly limited.

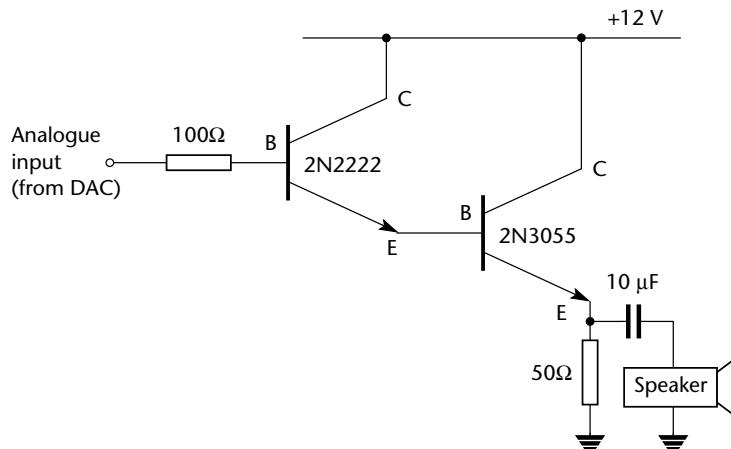


FIGURE 34.8 Amplifying the output from a DAC in order to drive a small loudspeaker

Example: Driving a speaker using a power op-amp

The National Semiconductor²⁴ LM12CL, an IC-based power amplifier suitable for use in high-quality audio equipment, is shown in Figure 34.9. In this amplifier, harmonic distortion is claimed to be ~0.01%.

24. www.national.com

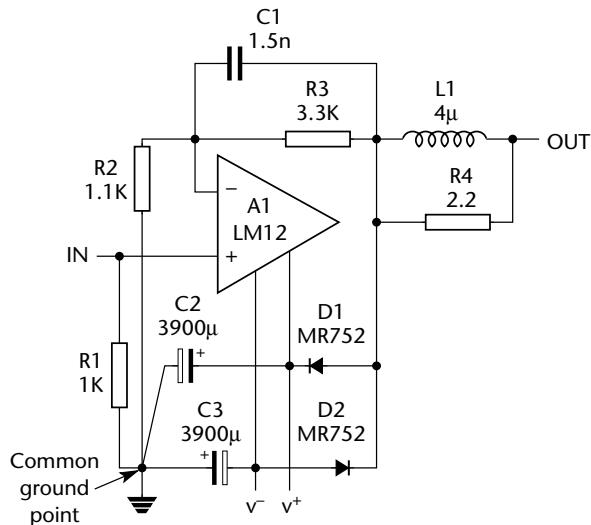


FIGURE 34.9 A power amplifier suitable for use in high-quality audio equipment (reproduced courtesy of National Semiconductor.)

Further reading

chapter **35**

Taking control

Introduction

The focus of the chapter is on proportional-integral-differential (PID) control. PID is both simple and effective: as a consequence it is the most widely used control algorithm. The focus here will be on techniques for designing and implementing PID controllers for use in embedded applications.

PID CONTROLLER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you design and implement a PID control algorithm?

Background

We consider in this section why we need ‘closed-loop’ control algorithms.

Open-loop control

Suppose we wish to control the speed of a DC motor, used as part of an air-traffic control application (Figure 35.1). To control this speed, we will assume that we have decided to change the applied motor voltage using a DAC.²⁵

As described, this is an example of a much more general ‘open-loop’ control approach (Figure 35.2).



FIGURE 35.1 A radar used as part of an air-traffic control application

25. We could use PWM control here: this would, however, simply complicate the example and would not alter the general conclusions

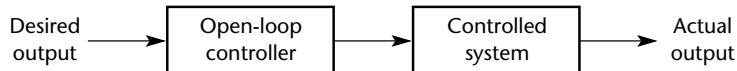


FIGURE 35.2 A schematic representation of an open-loop control system

Using open-loop control, we need to know what the system parameters are that will be required to create the desired system outputs. Thus, in the case of our air-traffic system, having knowledge of the motor, radar hardware and the motor drive circuit, we can set the speed of rotation that we require.

In an ideal world, this type of open-loop control system would be easy to design: we would simply have a lookup table linking the required motor speed to the required output parameters. For example, consider the DC motor to be used at the heart of this navigation system. We might start by assuming that this motor has a speed of operation directly proportional to the applied voltage (Figure 35.3).

As a result, to set the required speed of rotation, we should be able to use a lookup table (Table 35.1).

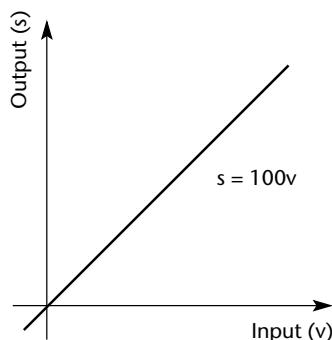


FIGURE 35.3 A simple model predicting the likely speed of rotation of a DC motor as the voltage is varied

TABLE 35.1 Translating the simple model in Figure 35.3 into a lookup table

Radar rotation speed (RPM)	DAC setting (8-bit)
0	0
2	51
4	102
6	153
8	204
10	255

This is an example of a general, linear, single-input single-output (SISO) system: this type of system can be represented graphically as shown in Figure 35.4.

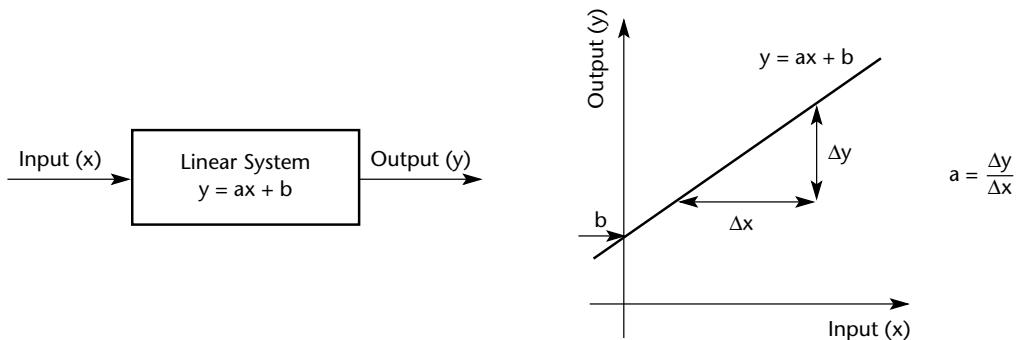


FIGURE 35.4 Modelling a linear system

Unfortunately, such linearity is very rare in practical systems. For example, our practical motor will have a maximum input voltage and a corresponding maximum speed of rotation (Figure 35.5). In addition, our practical motor will not begin rotating until a certain minimum voltage has been reached and will not abruptly stop at a maximum speed of rotation, but will have an I-O curve something like that shown in Figure 35.6.

Overall, our real motor is a non-linear system: it cannot be accurately represented by a simple linear ('straight line') model. Simply by inspecting Figure 35.6, we can see that to write down a description of this curve (that is, create a model of this motor) is going to be more complex than describing the simple linear model in Figure 35.3. Nonetheless, our (open-loop) lookup table solution can be adapted to deal with this non-linearity (Table 35.2).

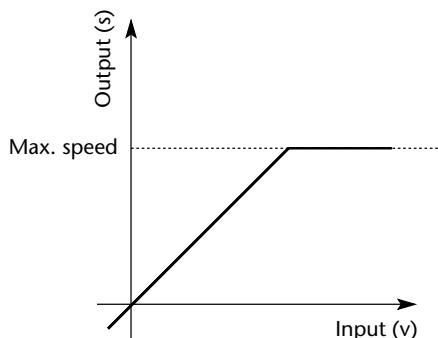


FIGURE 35.5 A slightly more realistic model predicting the likely speed of rotation of a DC motor as the voltage is varied

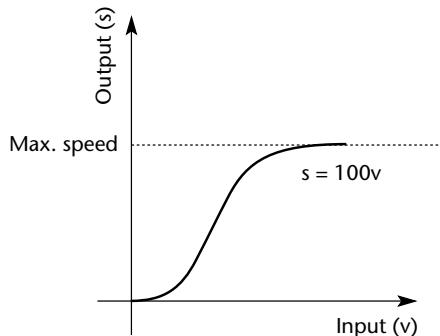


FIGURE 35.6 A further improvement on the DC motor model

TABLE 35.2 Translating the model in Figure 35.6 into a lookup table

Radar rotation speed (RPM)	DAC setting (8-bit)
0	0
2	61
4	102
6	150
8	215
10	255

However, this is not the only problem we have to deal with. Table 35.2 has, we assume, been created as a result of a series of tests on the real system. It therefore takes into account the non-linear nature of the motor and other system components. However, in addition to their non-linearity, most real systems also demonstrate characteristics which vary with time. In the case of the radar system, Table 35.2 does not take into account the wind speed or wind direction. As a result, this table of values is only valid (say) in still (wind-free) conditions.

If we are determined to use an open-loop approach, we will need to measure the wind speed and wind direction (Figure 35.7). Then we will need to create another table for 5 mph SE winds and another for 10 mph NW winds and so on.

Overall, this approach to control system design quickly becomes impractical.

Closed-loop control

The fundamental problem with the open-loop version of the radar control system is that the controller is 'blind': it receives no feedback about the system output (in this case the speed of rotation) which it is trying to control.

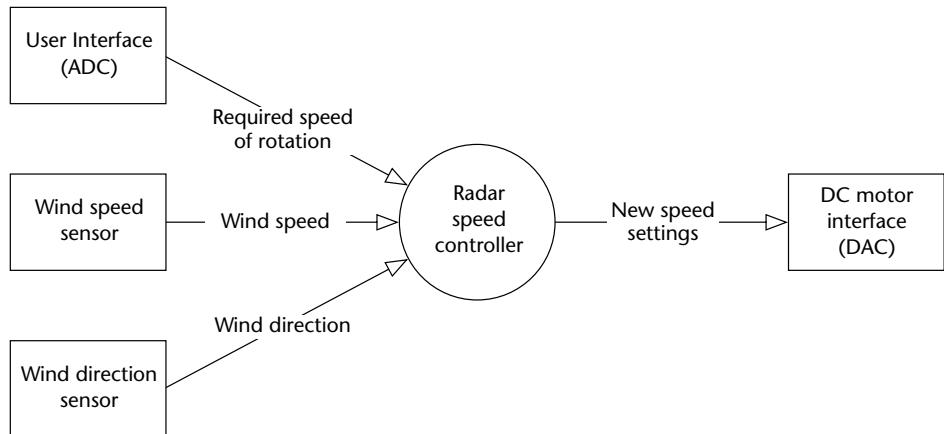


FIGURE 35.7 Taking the open-loop radar control system to its logical extreme

Consider the equivalent closed-loop version of the radar application (Figure 35.8). The key feature of this form of controller is the feedback loop (see Figure 35.9). This allows the system to respond effectively – for example – to external disturbances such as changes in wind speed or direction.

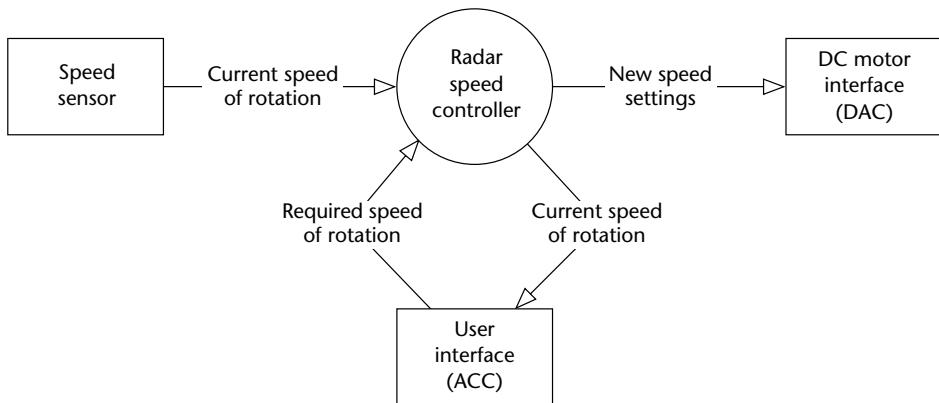


FIGURE 35.8 Controlling the speed of rotation of a radar in an air-traffic application (closed-loop version)

What closed-loop algorithm should you use?

There are numerous possible control algorithms that can be employed in the box marked 'closed-loop controller' in Figure 35.9 and the development and evaluation of new algorithms is an active area of research in many universities. A detailed discussion of some of the possible algorithms available is given by Nise (1995), Dutton *et al.* (1997) and Dorf and Bishop (1998).

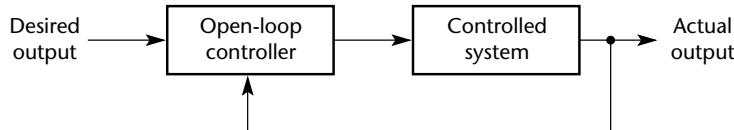


FIGURE 35.9 A schematic representation of a closed-loop control system

Despite the range of algorithms available, proportional-integral-differential (PID) control is found to be very effective in many cases and, as such, it is generally considered the ‘standard’ against which alternative algorithms are judged. Without doubt, it is the most widely used control algorithm in the world at the present time.

Solution

In this section we consider how to implement a PID-based control algorithm using a microcontroller.

What is PID control?

If you open a textbook on control theory, you will encounter a description of PID control containing an equation similar to that shown in Figure 35.10.

This may appear rather complex, but can – in fact – be implemented very simply, if you understand what the algorithm is trying to achieve.

For example, here is a complete PID control algorithm:

```

// Proportional term
Change_in_controller_output = PID_KP * Error;

// Integral term
Sum += Error;
Change_in_controller_output += PID_KI * Sum;

// Differential term
Change_in_controller_output += (PID_KD * SAMPLE_RATE * (Error - Old_error));
  
```

$$u(k) = u(k-1) + K \left[\left(1 + \frac{T}{T_I} + \frac{T_D}{T} \right) e(k) - \left(1 + 2 \frac{T_D}{T} \right) e(k-1) + \frac{T_D}{T} e(k-2) \right]$$

where:

$u(k)$ is the signal sent to the plant and $e(k)$ is the error signal, both at sample k

T is the sample period (in seconds) and $1/T$ is the sample rate (in Hz)

K is the proportional gain

$1/T_I$ is the integral gain

T_D is the derivative gain

FIGURE 35.10 One representation of the PID control algorithm

The algorithm has three components: the proportional component, the integral component and the differential component.

We will consider initially only the proportional term. As we will see, the proportional term makes up the main part of the control algorithm; indeed, this term alone is sufficient to produce many efficient, closed-loop control systems: these are sometimes referred to as **P-only controllers**. As we will see, the integral and differential terms may be used, if required, to ‘fine-tune’ the basic P-only response.

To understand the operation of this algorithm, suppose you are driving a car on a motorway and controlling the speed using only the accelerator (‘gas’) pedal. If you were currently travelling at 30 mph and wish to increase the speed to 35 mph, you would press the pedal gently; if you wished to increase the speed to 70 mph, you would press the pedal more vigorously.

This is the basis of ‘proportional’ control. Specifically, we measure the ‘error’ between the desired system output (the current speed of the vehicle in this example) and the current system output (the desired speed). The difference (desired speed – current speed) is the error and P-only algorithm seeks to reduce this error to 0. The algorithm does this by changing the controller output (the pedal setting, in our example) by an amount proportional to the error term.

Thus, the proportional term in our PID controller may be implemented as follows:

```
Change_in_controller_output = PID_KP * Error;
```

where PID_KP is the ‘proportional gain’ which is adjusted, by the user, to adapt the general algorithm to match the needs of a particular application.

A complete implementation of a PID control algorithm is given in Listing 35.1. We will explore the impact of the I and D terms later.

```
/*
-----*
PID_f1.C (v1.00)
-----
Simple PID control implementation.
*-----*/
#include "PID_f1.h"

// ----- Private constants -----
#define PID_KP (0.70f)      // Proportional gain
#define PID_KI (0.03f)      // Integral gain
#define PID_KD (0.04f)      // Differential gain

#define PID_MAX (0.5f)      // Maximum PID controller output
#define PID_MIN (0.0f)       // Minimum PID controller output

// ----- Private variable definitions-----
```

```

static float Sum_G;           // Integrator component
static float Old_error_G;    // Previous error value

/*-----*/
PID_Control()
Simple floating-point version.

/*-----*/
float PID_Control(float Error, float Control_old)
{
    // Proportional term
    float Control_new = Control_old + (PID_KP * Error);

    // Integral term
    Sum_G += Error;
    Control_new += PID_KI * Sum_G;

    // Differential term
    Control_new += (PID_KD * SAMPLE_RATE * (Error - Old_error_G));

    // Control_new cannot exceed PID_MAX or fall below PID_MIN
    if (Control_new > PID_MAX)
    {
        Control_new = PID_MAX;
    }
    else
    {
        if (Control_new < PID_MIN)
        {
            Control_new = PID_MIN;
        }
    }

    // Store error value
    Old_error_G = Error;

    return Control_new;
}

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 35.1 A complete implementation of the PID control algorithm

Dealing with ‘windup’

While the algorithm shown in Listing 35.1 will work, it can be improved with one minor change to the integral element.

Note how this element is implemented:

```
Sum_G += Error;
Control_new += PID_KI * Sum_G;
```

One problem that frequently arises in a real system is that, at times, the actuator will be operating at its maximum (or minimum) limit; for example, the PWM unit may be operating at 100%. Under these circumstances, the actuator cannot be increased (or decreased), and changing the value of Sum_G will serve no useful purpose; on the contrary, it will make the system response sluggish when the source of the error is removed.

We can improve the system performance by halting the update of Sum_G when the system reaches an actuator limit. This facility is called ‘anti-windup’.

The PID implementation in Listing 35.2 includes this form of windup protection.

```
/* -----
   PID_wf1.C (v1.00)

-----
   Simple PID control implementation, with anti-windup protection.

*/
#include "PID_wf1.h"

// ----- Private constants -----
#define PID_KP (0.2f)           // Proportional gain
#define PID_KI (0.01f)          // Integral gain
#define PID_KD (0.03f)          // Differential gain
#define PID_WINDUP_PROTECTION (1) // Set to TRUE (1) or FALSE (0)
#define PID_MAX (1.0f)           // Maximum PID controller output
#define PID_MIN (0.0f)           // Minimum PID controller output

// ----- Private variable definitions-----
static float Sum_G;           // Integrator component
static float Old_error_G; // Previous error value

/*
PID_Control()

Simple floating-point version.

See text for details.

*/

```

```

float PID_Control(float Error, float Control_old)
{
    // Proportional term
    float Control_new = Control_old + (PID_KP * Error);

    // Integral term
    Sum_G += Error;
    Control_new += PID_KI * Sum_G;

    // Differential term
    Control_new += (PID_KD * SAMPLE_RATE * (Error - Old_error_G));

    // Optional windup protection - see text
    if (PID_WINDUP_PROTECTION)
    {
        if ((Control_new > PID_MAX) || (Control_new < PID_MIN))
        {
            Sum_G -= Error; // Don't increase Sum...
        }
    }

    // Control_new cannot exceed PID_MAX or fall below PID_MIN
    if (Control_new > PID_MAX)
    {
        Control_new = PID_MAX;
    }
    else
    {
        if (Control_new < PID_MIN)
        {
            Control_new = PID_MIN;
        }
    }

    // Store error value
    Old_error_G = Error;

    return Control_new;
}

/* -----
   ---- END OF FILE -----
   ----- */

```

Listing 35.2 A complete implementation of the PID control algorithm, with windup protection

Choosing the controller parameters

Two aspects of PID control algorithms deter new users. The first is that the algorithm is seen to be ‘complex’: as we have demonstrated already, this is a fallacy, since PID controllers can be very simply implemented.

The second concern lies with the tuning of the controller parameters. Fortunately, such concerns are, again, often exaggerated.

We suggest the use of the following methodology to tune the PID parameters:

- 1** Set the integral (KI) and differential (KD) terms to 0.
- 2** Increase the proportional term (KP) slowly, until you get continuous oscillations.
- 3** Reduce KP to half the value determined.
- 4** If necessary, experiment with small values of KD to damp-out ‘ringing’ in the response.
- 5** If necessary, experiment with small values of KI to reduce the steady-state error in the system.
- 6** Always use windup protection if using a non-zero KI value.

Note that steps 1–3 of this technique are a simplified version of the Ziegler–Nichols guide to PID tuning; these date from the 1940s (see Ziegler and Nichols, 1942; Ziegler and Nichols, 1943).

We illustrate this approach in a following example.

What sample rate?

In Chapter 32, we discussed the selection of appropriate sample rates for signal-processing systems and demonstrated the effect of aliasing that result when too low a sample rate is used. Specifically, we argued that the sample rate had to be at least twice the system bandwidth and that this bandwidth was dictated by the maximum frequency component in the system we were analyzing. Thus, to process a speech signal containing components up to 4 kHz, we would need to sample at a minimum of 8 kHz and remove any higher frequencies with an anti-aliasing filter. Determining the required bandwidth of such an application will typically be carried out using a spectrum analyzer.

Determining the required sample rate for a control system must be carried out in a slightly different way. One effective technique involves the measurement of the system rise time (Figure 35.11). This is an open-loop test that involves, for example, having an engine running at minimum speed then – having fully opened the throttle – measuring how long the system takes to reach maximum speed.

Having determined the rise time (measured in seconds), we can – making some simplifying assumptions – calculate the required sample frequency as follows:

$$\text{Sample frequency} = \frac{6}{\text{Rise time}}$$

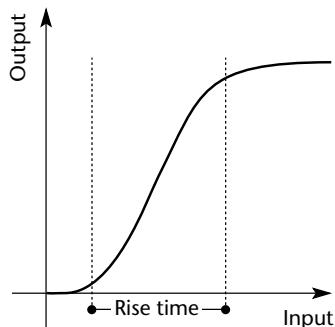


FIGURE 35.11 Measuring the rise time of a system

Here the sample rate is in Hertz and the rise time in seconds. Thus, if the rise time measured in Figure 35.11 were 0.10 seconds, the required sample frequency would be 6 Hz.

This assumes that the bandwidth is approximately 20 times the systems natural frequency: this is a reasonable assumption. (Please see Franklin *et al.*, 1994, for further details.)

Hardware resource implications

Implementation of a PID control algorithm requires some floating-point or integer mathematical operations. The precise load will vary with the implementation used, but a typical implementation requires four multiplications, three additions and two subtractions. With floating-point operations, this amounts to a total of approximately 2,000 instructions (using the Keil compiler, on an 8051 without hardware maths support). This operation can be carried out every millisecond on a standard (12 osc / instruction) 8051 running at 24 MHz, if there is no other CPU-intensive processing to be done.

A one-millisecond loop time is more than adequate for most control applications, which typically require sample intervals of several hundred milliseconds or longer. Of course, if you require higher performance, many more modern implementations of the 8051 microcontroller can provide this. For example, a Dallas 520 microcontroller (at 32 MHz) will execute the PID implementation just outlined within 0.25 ms. Similarly, devices such as the Infineon 517 and 509, which have hardware maths support, will also execute this code rapidly, should this be required.

Safety and reliability implications

Any device that controls a dangerous actuator (such as a high-powered motor or robotic mechanism) has safety implications. Any algorithm used to control such a load must be treated with caution.

Portability

The PID algorithms discussed here can be used with any microcontroller or micro-processor family.

Overall strengths and weaknesses

- 😊 Suitable for many single-input single-output (SISO) systems.
- 😊 Generally effective.
- 😊 Easy to implement.
- 😢 Not suitable for use in multi-input or multi-output applications.
- 😢 Parameter tuning can be time consuming.

Related patterns and alternative solutions

In this section we consider some possible alternatives to PID control.

Why open-loop controllers are still (sometimes) useful

In ‘Background’ we suggested that closed-loop control is always a better alternative than open-loop control. This is an over-simplification.

Open-loop control still has a role to play. For example, if we wish to control the speed of an electric fan in an automotive air-conditioning system, we may not need precise speed control and an open-loop approach might be appropriate.

In addition, it is not always possible to measure the quantity we are trying to control directly, making closed-loop control impractical. For example, in an insulin delivery system used for patients with diabetes, we are seeking to control levels of glucose in the bloodstream. However, glucose sensors are not available, so an open-loop controller must be used (see Dorf and Bishop (1998, p. 22) for further details).

Similar problems apply throughout much of the process industry, where sensors are not available to determine product quality.

Limitations of PID control

PID control is only suitable for ‘single-input single-output’ (SISO) systems or for systems that can be broken down into SISO components. PID control is not suitable for systems with multiple inputs and / or multiple outputs. In addition, even for SISO systems, PID can only control a single system parameter, and is not suitable for multi-parameter (sometimes called multi-variable) systems.

Please refer to Franklin *et al.* (1994), Nise (1995), Dutton *et al.* (1997), Dorf and Bishop (1998) and Franklin *et al.* (1998) for further discussions on multi-input, multi-output and multi-parameter control algorithms.

Fuzzy control

The texts mentioned highlight traditional (mathematically based) approaches to the design of control systems. A less formal approach to control system design has emerged recently: this is known as ‘fuzzy control’ and is suitable for SISO, MISO and MIMO systems, with one or more parameters. (Refer to Passino and Yurkovich, 1998, for further information on fuzzy control.)

Example: Tuning the parameters of a cruise-control system

In this example, we take a simple computer simulation of a vehicle, and develop an appropriate cruise-control system to match.

The model is given in Listing 35.3.

```
/*-----*
Cruise.CPP (v1.00)
-----
Desktop C++ program to demonstrate PID control.
Cruise-control application using simple vehicle model.

-*-----*/
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "PID_f1.h"

// ----- Private constants -----
#define MS_to MPH (2.2369)      // Convert metres/sec to miles per hour
#define FRIC (50)                // Friction coeff- Newton Second / Metre
#define MASS (1000)              // Mass of vehicle (kgs)
#define N_SAMPLES (1000)         // Number of samples
#define ENGINE_POWER (5000)       // N
#define DESIRED_SPEED (31.3f)     // Metres/sec [* 2.2369 to convert to mph]

/* ..... */ /* ..... */
int main()
{
    float Throttle = 0.313f; // Throttle setting (fraction)
    float Old_speed = DESIRED_SPEED, Old_throttle = 0.313f;
    float Speed, Accel;
```

```
float Dist;
float Sum = 0.0f;

float Error;

// Open file to store results
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++)
{
    // Error drives the controller
    Error = (DESIRED_SPEED - Old_speed);

    // Calculate throttle setting
    Throttle = PID_Control(Error, Throttle);
    // Throttle = 0.313f; // Use for open-loop demo

    // Simple car model
    Accel = (float)(Throttle * ENGINE_POWER - (FRIC * Old_speed)) / MASS;
    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; // Sudden gust of wind into rear of car
    }

    if (t == 550)
    {
        Speed = 26.8f; // Sudden gust of wind into front of car
    }

    // Display speed in miles per hour
    cout << Speed * MS_to MPH << endl;
    out_FP << Speed * MS_to MPH << endl;

    // Ready for next loop
    Old_speed = Speed;
    Old_throttle = Throttle;
}
```

```

    return 0;
}

/*-----*
 *----- END OF FILE -----*
 *-----*/

```

Listing 35.3 A simple model of a vehicle to be used to illustrate the design of a cruise-control system

We can illustrate the basic operation of this model by demonstrating it running in open-loop mode (Figure 35.12).

In Figure 35.12, the car is controlled by maintaining a fixed throttle position at all times. Because we assume the vehicle is driving on a straight, flat, road with no wind, the speed is constant (70 mph) for most of the 1,000-second trip. There are, however, two exceptions. At time $t = 50$ seconds, we simulate a sudden gust of wind at the rear of the car; this speeds the vehicle up and it slowly returns to the set speed value. Similarly, at time $t = 550$ seconds, we simulate a sharp gust of wind at the front of the car; this slows the vehicle down.

We will now go through the process of tuning a PID control algorithm that will allow the vehicle to recover its required speed more rapidly in the event of such disturbances. The PID algorithm we will use is given in Listing 35.2.

As outlined in ‘Solution’, we will tune this algorithm by applying the following methodology:

- 1 Set integral (KI) and differential (KD) terms to 0.
- 2 Increase the proportional term (KP) slowly, until we get continuous oscillations.
- 3 Reduce KP to half the value determined.
- 4 If necessary, experiment with small values of KD to damp-out ‘ringing’ in the response.

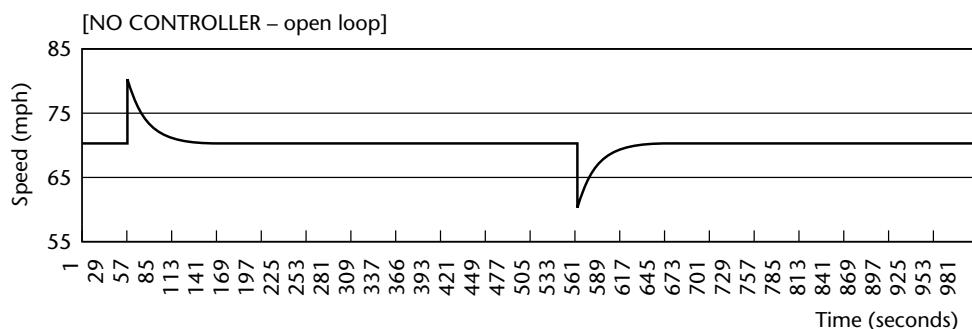


FIGURE 35.12 Running the cruise-control application in open-loop mode

- 5 If necessary, experiment with small values of KI to reduce the steady-state error in the system.
- 6 Always use windup protection if using a non-zero KI value.

Figure 35.13 shows the first test.

```
#define PID_KP (0.20f)
#define PID_KI (0.00f)
#define PID_KD (0.00f)
```

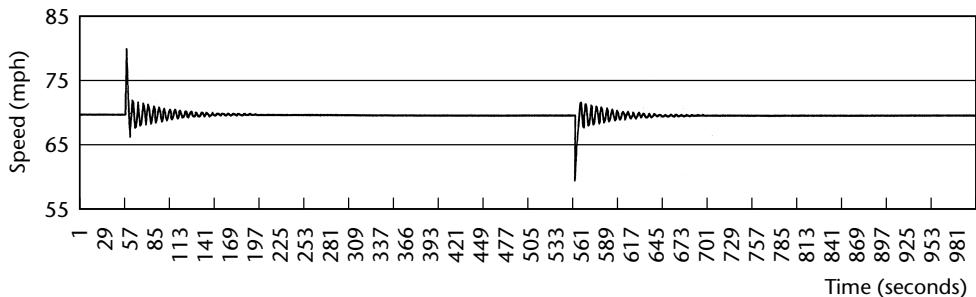


FIGURE 35.13 Tuning the controller

Figure 35.14 shows the result of steadily increasing the value of KP, until we get small, constant, oscillations.

```
#define PID_KP (1.00f)
#define PID_KI (0.00f)
#define PID_KD (0.00f)
```

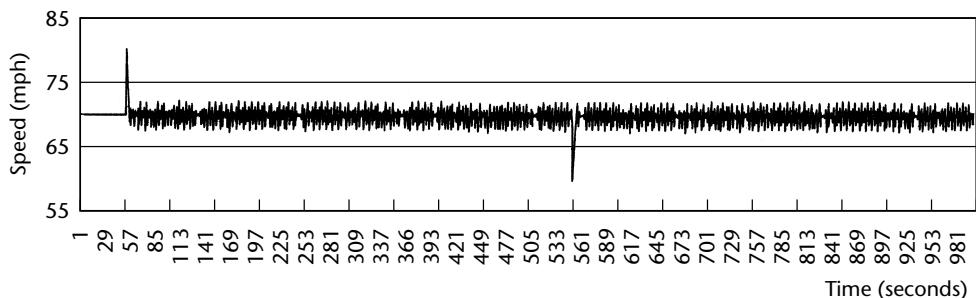


FIGURE 35.14 Tuning the controller

The results of this experiment suggest that a value of $KP = 0.5$ will be appropriate (that is, half the value used to generate the constant oscillations). This value is illustrated in Figure 35.15.

```
#define PID_KP (0.50f)
#define PID_KI (0.00f)
#define PID_KD (0.00f)
```

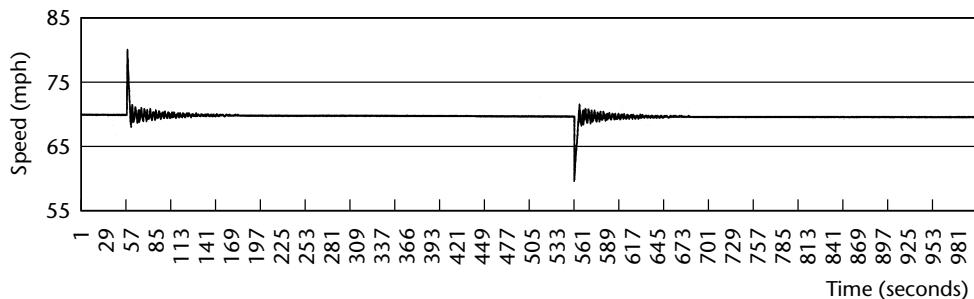


FIGURE 35.15 Tuning the controller

This response might be considered adequate for a basic cruise-control application; however, the guidelines already given suggest that we should be able to reduce the transient ringing effect (after each disturbance) by use of an appropriate differential term. Some brief experiments suggest that the parameters used in Figure 35.16 are appropriate.

```
#define PID_KP (0.50f)
#define PID_KI (0.00f)
#define PID_KD (0.10f)
```

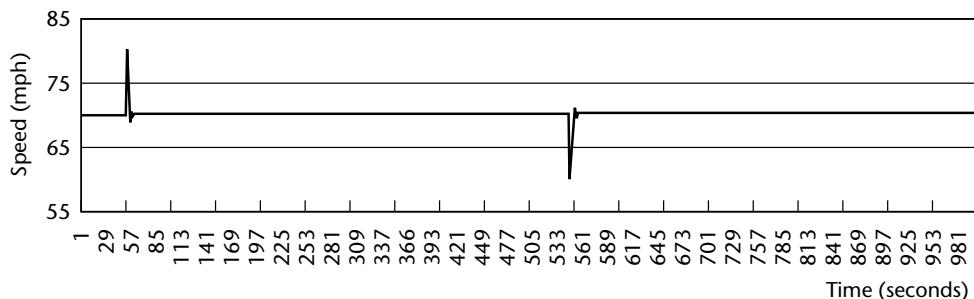


FIGURE 35.16 Tuning the controller

Note that, with these parameters, the system reaches the required speed within a few seconds of each disturbance; you may like to compare this, again, with the original open-loop version shown in Figure 35.12.

Note also that we can reduce the system complexity here by omitting the integral term and using this PD controller.

Example: DC motor speed control

The previous example is based on a computer simulation. In this example, we demonstrate closed-loop control of a real DC motor.

The motor is controlled via a PWM interface (Figure 35.17); to allow closed-loop control an encoder is mounted on the motor shaft: this generates one pulse every time the shaft rotates.

The key source files required in the application can be found in Listings 35.3 to 35.6: all the files for this project are included on the CD.

Note that this example uses a different, integer-based PID implementation. As we discussed in ‘Hardware resource implications’, integer-based solutions impose a lower CPU load than floating-point equivalents.

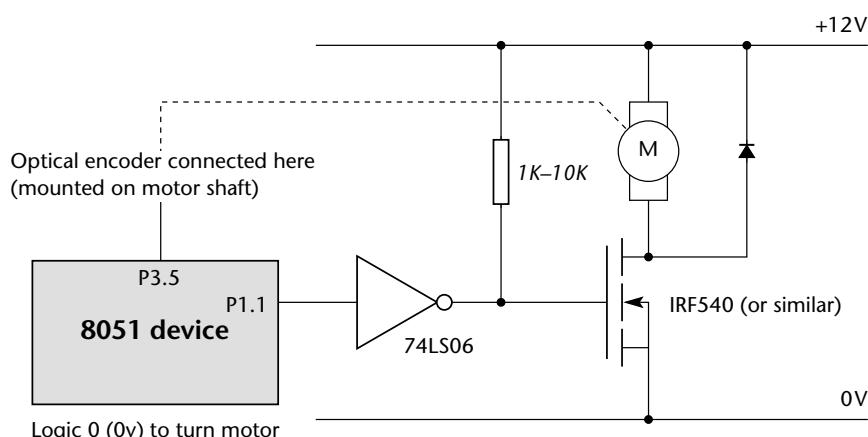


FIGURE 35.17 Hardware for controlling the speed of a DC motor

```
/*
 *-----*
 * Port.H (v1.00)
 *-----*
 'Port Header' (see Chap 10) for the project PIDmotor
 *-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS
#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed

```

```

// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P5

#endif

// ----- PIDmotor.C -----
// ADC reading from Pin 6.0
// PWM output on Pin 1.1

// Opto-encoder (or similar) connected to this pin
sbit Pulse_count_pin = P3^5;

// ----- Lnk_0.C -----
// Pin 3.1 used for RS-232 interface

/*
----- END OF FILE -----
*/

```

Listing 35.4 Part of an example illustrating PID control of a DC motor

```

/* -----
Main.c (v1.00)

-----
Motor (speed) control demonstrator.

PI algorithm.

*/
#include "Main.h"
#include "1_01_12g.h"

#include "PIDMotor.h"
#include "Lnk_0_B.h"

/* ..... */
/* ..... */

void main(void)
{
    SCH_Init_T1(); // Set up the scheduler
    PID_MOTOR_Init();
}

```

```
// Set baud rate to 9600, using internal baud rate generator
// Generic 8051 version
PC_LINK_Init_Internal(9600);

// Add a 'pulse count poll' task
// TIMING IS IN TICKS (1ms interval)
// Every 5 milliseconds (200 times per second)
SCH_Add_Task(PID_MOTOR_Poll_Speed_Pulse, 1, 1);

SCH_Add_Task(PID_MOTOR_Control_Motor, 300, 1000);

// Sending data to serial port
SCH_Add_Task(PC_LINK_Update, 3, 1);

// All tasks added: start running the scheduler
SCH_Start();

while(1)
{
    SCH_Dispatch_Tasks();
}

/*
----- END OF FILE -----
*/

```

Listing 35.5 Part of an example illustrating PID control of a DC motor

```
/*
----- *-
PID_Motor.c (v1.00)

-----
Small library for PID control of a DC motor.
For C515c microcontroller.

The set point (required speed) is read via a potentiometer
and on-chip ADC.

The current speed is read via an optical encoder. The pulses
from the encoder are counted using T0.

The new speed is set by PWM using the on-chip capture-compare
unit (Timer 2).

----- */

```

```
#include "Main.h"
#include "Port.h"

#include "PIDMotor.h"
#include "Lnk_0_B.h"

// ----- Public constants -----
extern const char code CHAR_MAP_G[10];

// ----- Private function prototypes -----
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);

// ----- Private constants -----
#define PULSE_HIGH (0)
#define PULSE_LOW (1)

#define PID_PROPORTIONAL (5)
#define PID_INTEGRAL      (50)
#define PID_DIFFERENTIAL (50)

// ----- Private variables -----
// Used for demo purposes only
tWord Ticks = 0;

// Stores the latest count value
static tByte Pulse_count_G;

// Data to be copied to the serial port
static char PID_MOTOR_data_G[50] = {" "};

// Measured speed, required speed and controller output variables
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()
 *
 * Prepare for UD motor control.
 *-----*/
```

```
void PID_MOTOR_Init(void)
{
// -----
// Set up the initial data to be sent to the PC via RS-232
// -----
char* pScreen_Data = "Cur      Des      PWM      \n";
tByte c;

for (c = 0; c < 30; c++)
{
    PID_MOTOR_data_G[c] = pScreen_Data[c];
}

// -----
// Set up the A-D converter
// (used to measure the 'set point' (the desired motor speed)
// -----

// Select internally-triggered single conversion
// Reading from P6.0 (single channel)
ADCON0 = 0xC0; // Mask bits 0 - 5 to 0

// Select appropriate prescalar ratio: see manual for details
ADCON1 = 0x80; // Make bit 7 = 1 : Prescaler ratio=8

// -----
// Set up the PWM output (Cap Com) unit - T2
// (used to set the desired motor speed)
// -----

// ----- T2 Mode -----
// Mode 1 = Timerfunction

// Prescaler: Fcpu/6

// ----- T2 reload mode selection -----
// Mode 0 = auto-reload upon timer overflow
// Preset the timer register with autoreload value ! 0xFF00;
TL2 = 0x00;
TH2 = 0xFF;

// ----- T2 general compare mode -----
// Mode 0 for all channels
T2CON |= 0x11;

// ----- T2 general interrupts -----
// Timer 2 overflow interrupt is disabled
```

```
ET2=0;
// Timer 2 external reload interrupt is disabled
EXEN2=0;

// ----- Compare/capture Channel 0 -----
// Disabled??
// Set Compare Register CRC on: 0xFF00;
CRCL = 0x00;
CRCH = 0xFF;

// CC0/ext3 interrupt is disabled
EX3=0;

// ----- Compare/capture Channel 1 -----
// Compare enabled
// Set Compare Register CC1 on: 0xFF80;
CCL1 = 0x80;
CCH1 = 0xFF;

// CC1/ext4 interrupt is disabled
EX4=0;

// ----- Compare/capture Channel 2 -----
// Disabled
// Set Compare Register CC2 on: 0x0000;
CCL2 = 0x00;
CCH2 = 0x00;
// CC2/ext5 interrupt is disabled
EX5=0;

// ----- Compare/capture Channel 3 -----
// Disabled
// Set Compare Register CC3 on: 0x0000;
CCL3 = 0x00;
CCH3 = 0x00;

// CC3/ext6 interrupt is disabled
EX6=0;

// Set all above mentioned modes for channel 0-3
CCEN = 0x08;

// -----
// Count pulses on Pin 3.5 [software only]
// (used to measure the current motor speed)
// -----
Pulse_count_pin = 1;
```

```
Pulse_count_G = 0;
}

/*-----*/
PID_MOTOR_Control_Motor()
The main motor control function.

/*-----*/
void PID_MOTOR_Control_Motor(void)
{
    int Error;
    int Control_new;

    // Get the current speed value (0-255)
    Speed_measured_G = PID_MOTOR_Read_Current_Speed();

    // Get the desired speed value (0-255)
    Speed_required_G =
        PID_MOTOR_Get_Required_Speed();

    if (++Ticks == 100)
    {
        Speed_required_G = 200;
    }

    // Difference between required and actual speed (0-255)
    Error = Speed_required_G - Speed_measured_G;

    // Proportional term
    Control_new = Controller_output_G + (Error / PID_PROPORIONAL);

    // Integral term [SET TO 0 IF NOT REQUIRED]
    if (PID_INTEGRAL)
    {
        Sum_G += Error;
        Control_new += (Sum_G / (1 + PID_INTEGRAL));
    }

    // Differential term [SET TO 0 IF NOT REQUIRED]
    if (PID_DIFFERENTIAL)
    {
        Control_new += (Error - Old_error_G) / (1 + PID_DIFFERENTIAL);

        // Store error value
        Old_error_G = Error;
    }
}
```

```
// Adjust to 8-bit range
if (Control_new > 255)
{
    Control_new = 255;
    Sum_G -= Error; // Windup protection
}

if (Control_new < 0)
{
    Control_new = 0;
    Sum_G -= Error; // Windup protection
}

// Convert to required 8-bit format
Controller_output_G = (tByte) Control_new;

// Update the PWM setting
PID_MOTOR_Set_New_PWM_Output(Controller_output_G);

// Update display
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_Write_String_To_Buffer(PID_MOTOR_data_G);
}

/*-----*
 * PID_MOTOR_Get_Required_Speed()
 *
 * Get the required speed via the Pot and ADC.
 *-----*/
tByte PID_MOTOR_Get_Required_Speed(void)
{
    // Take sample from A-D

    // Write (value not important) to ADDATL to start conversion
    ADDATL = 0x01;
```

```
// Wait for conversion to complete
// NOTE: This demo software has no timeout...
while (BSY == 1);

// 10-bit A-D result is now available
// return 8-bit result
return ADDATH;
}

/*-----*
PID_MOTOR_Set_New_PWM_Output()

Adjust the PWM output value.

-----*/
void PID_MOTOR_Set_New_PWM_Output(const tByte Controller_output_G)
{
    //
    // Changing value in CCL1 to generate appropriate PWM duty cycle
    CCL1 = Controller_output_G;
}

/*-----*
PID_MOTOR_Read_Current_Speed()

Schedule this function at regular intervals.

Remember: max count is 65536 (16-bit counter)
- it is your responsibility to ensure this count
is not exceeded. Choose an appropriate schedule
interval and allow a margin for error.

For high-frequency pulses, you need to take account of
the fact that the count is stop for a (very brief) period,
to read the counter.

Note: the delay before the first count is taken should
generally be the same as the inter-count interval,
to ensure that the first count is as accurate as possible.

For example, this is OK:

    Sch_Add_Task(PID_MOTOR_Read_Current_Speed, 1000, 1000);

While this will give a very low first count:

    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;

    // Normalized: 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()

Using software to count falling edges on a specified pin
- T0 is *NOT* used here.

/*-----*/
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;
}

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 35.6 Part of an example illustrating PID control of a DC motor

Further reading

- 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 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.
- 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.

Specialized time-triggered architectures

We conclude the collection of patterns in this book by considering four specialized schedulers. These are all co-operative in nature, but are adapted to meet the needs of particular applications where, for example, low power consumption or very accurate scheduler timing are required.

More specifically:

- In Chapter 36, we consider ways of maintaining the advantages of the scheduled architecture while reducing the CPU and / or memory load it imposes.
- In Chapter 37, we discuss simple and cost-effective techniques for improving the stability of the scheduler timing in the face of inevitable fluctuations in ambient temperature.

Reducing the system overheads

Introduction

In this chapter, we consider some of the ways in which the flexible nature of the general-purpose schedulers presented in this book may be exploited, in order to create special-purpose schedulers adapted for the needs of particular applications.

The following patterns are presented in this chapter:

- **255-TICK SCHEDULER** [page 894]

A scheduler designed to run multiple tasks, but with reduced memory (and CPU) overheads. This scheduler operates in the same way as the standard co-operative schedulers, but all information is stored in byte-sized (rather than word-sized) variables: this reduces the required memory for each task by around 30%.

- **ONE-TASK SCHEDULER** [page 911]

A stripped-down, co-operative scheduler able to manage a single task. This very simple scheduler makes very efficient use of hardware resources, with the bare minimum of CPU and memory overheads.

- **ONE-YEAR SCHEDULER** [page 919]

A scheduler designed for very low-power operation: specifically, it is designed to form the basis of battery-powered applications capable of operating for a year or more from a small, low-cost battery supply.

255-TICK SCHEDULER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you reduce the scheduler memory requirements?

Background

Solution

As we discussed in Chapter 14, the majority of the schedulers in this book are based on the following data structure:

```
// Store in DATA area, if possible, for rapid access
// Total memory per task is 7 bytes
typedef data struct
{
    // Pointer to the task (must be a 'void (void)' function)
    void (code * pTask)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tWord Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tWord Period;

    // Incremented (by scheduler) when task is due to execute
    tByte RunMe;
} sTask;
```

This data structure allows initial task delays and task intervals of up to 65,535 ticks. In doing so, it requires a total of seven bytes of memory per task.

The 255-tick scheduler data structure is slightly modified:

```
// Store in DATA area, if possible, for rapid access
// Total memory per task is 5 bytes
typedef data struct
{
    // Pointer to the task (must be a 'void (void)' function)
    void (code * pTask)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tByte Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tByte Period;

    // Incremented (by scheduler) when task is due to execute
    tByte RunMe;
} sTask;
```

This data structure allows initial task delays and task intervals of up to 255 ticks. In doing so, it requires a total of five bytes of memory per task.

Hardware resource implications

In addition to the significant memory savings, use of this scheduler also provides a slight performance improvement, since the processor can manipulate the 8-bit variables in the 255-tick scheduler more rapidly than the 16-bit variables in the conventional version.

Reliability and safety implications

This scheduler has all the reliability and safety features of the co-operative schedulers we have discussed throughout this book.

Portability

This scheduler is as portable as the other co-operative schedulers discussed throughout this book.

Overall strengths and weaknesses

- ☺ Provides a co-operative scheduling environment.
- ☺ Reduced memory requirements (compared with CO-OPERATIVE SCHEDULER [page 255]).

- ☺ Slightly reduced CPU loading.
- ☺ Limited (255-tick) initial delays and task intervals.

Related patterns and alternative solutions

See CO-OPERATIVE SCHEDULER [page 255].

Example: A generic 255-tick scheduler

We provide a complete example of a generic 255-tick scheduler in Listings 36.1 to 36.8.

```
/*-----*
Port.H (v1.00)
-----*
'Port Header' (see Chap 10) for the project SCH_255 (see Chap 36)

-*-----*/
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
#ifndef SCH_REPORT_ERRORS

#define SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif
// ----- LED_Flas.C -----
// Connect LED from +5V (etc) to this pin, via appropriate resistor
// [see Chapter 7 for details]
sbit LED_pin = P1^5;

/*-----*
----- END OF FILE -----
-*-----*/
```

Listing 36.1 Part of a generic 255-tick scheduler example

```
/*-----*
Main.c (v1.00)
-----*
```

```

Demonstration program for:

Generic 255-tick scheduler using T2.

Assumes 12 MHz oscillator (-> 05 ms tick interval).

*** All timing is in TICKS (not milliseconds) ***

Required linker options (see Chapter 14 for details):

OVERLAY (main ~ (LED_Flash_Update),
SCH_Dispatch_Tasks ! (LED_Flash_Update))

----- */

#include "Main.h"
#include "2_05_12g.h"
#include "LED_flas.h"

/* ..... */
/* ..... */

void main(void)
{
    // Set up the scheduler
    SCH_Init_T2();

    // Prepare for the 'Flash_LED' task
    LED_Flash_Init();

    // Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
    // - timings are in ticks (5 ms tick interval)
    // (Max interval / delay is *** 255 *** ticks)
    SCH_Add_Task(LED_Flash_Update, 0, 200);

    // Start the scheduler
    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}

----- */
----- END OF FILE -----
----- */

```

Listing 36.2 Part of a generic 255-tick scheduler example

```

/*-----*
2_05_12g.h (v1.00)

-----
- see 2_05_12g.C for details

-*-----*/
#include "Main.h"
#include "SCH51a.H"

// ----- Public function prototypes -----
void SCH_Init_T2(void);
void SCH_Start(void);

/*-----*
----- END OF FILE -----
-*-----*/

```

Listing 36.3 Part of a generic 255-tick scheduler example

```

/*-----*
2_05_12g.C (v1.00)

-----

*** THIS IS A 255-TICK SCHEUDLER FOR STANDARD 8051 / 8052 ***

*** Uses T2 for timing, 16-bit auto reload ***
*** 12 MHz oscillator -> 5 ms (precise) tick interval ***

-*-----*/
#include "2_05_12g.h"

// ----- Public variable declarations -----
// The array of tasks (see Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable
//
// See Port.H for port on which error codes are displayed
// and for details of error codes
extern tByte Error_code_G;

/*-----*/

```

SCH_Init_T2()

Scheduler initialization function. Prepares scheduler data structures and sets up timer interrupts at required rate. Must call this function before using the scheduler.

```
-----*/
void SCH_Init_T2(void)
{
    tByte i;

    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // Now set up Timer 2
    // 16-bit timer function with automatic reload

    T2CON = 0x04;    // load Timer 2 control register
    T2MOD = 0x00;    // load Timer 2 mode register

    // Crystal is assumed to be 12 MHz
    // The Timer 2 resolution is 0.000001 seconds (1 µs)
    // The required Timer 2 overflow is 0.005 seconds (5 ms)
    // - this takes 5000 timer ticks
    // Reload value is 65536 - 5000 = 60536 (dec) = 0xEC78

    TH2      = 0xEC;    // load Timer 2 high byte
    RCAP2H = 0xEC;    // load Timer 2 reload capture reg, high byte
    TL2      = 0x78;    // load Timer 2 low byte
    RCAP2L = 0x78;    // load Timer 2 reload capture reg, low byte

    // Start timer
    ET2      = 1;    // Timer 2 interrupt is enabled
    TR2      = 1;    // Start Timer 2
}
```

```
-----*/
```

SCH_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

```
- *-----*/  
void SCH_Start(void)  
{  
    EA = 1;  
}  
/*-----*/  
  
SCH_Update()  
  
This is the scheduler ISR. It is called at a rate  
determined by the timer settings in SCH_Init().  
This version is triggered by Timer 2 interrupts:  
timer is automatically reloaded.  
-----*/  
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow  
{  
    tByte Index;  
  
    TF2 = 0; // Have to manually clear this.  
  
    // NOTE: calculations are in *TICKS* (not milliseconds)  
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)  
    {  
        // Check if there is a task at this location  
        if (SCH_tasks_G[Index].pTask)  
        {  
            if (SCH_tasks_G[Index].Delay == 0)  
            {  
                // The task is due to run  
                SCH_tasks_G[Index].RunMe += 1; // Increment the run flag  
  
                if (SCH_tasks_G[Index].Period)  
                {  
                    // Schedule periodic tasks to run again  
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;  
                }  
            }  
        }  
        else  
        {  
            // Not yet ready to run: just decrement the delay  
        }  
    }  
}
```

```

        SCH_tasks_G[Index].Delay -= 1;
    }
}
}

/*
----- END OF FILE -----
*/

```

Listing 36.4 Part of a generic 255-tick scheduler example

```

/*
----- * -
SCH51a.h (v1.00)

-----
- see SCH51a.C for details

-*----- * /
#ifndef _SCH51_H
#define _SCH51_H

#include "Main.h"

// ----- Public data type declarations -----
// Store in DATA area, if possible, for rapid access
// Total memory per task is 5 bytes
typedef data struct
{
    // Pointer to the task (must be a 'void (void)' function)
    void (code * pTask)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tByte Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tByte Period;

    // Incremented (by scheduler) when task is due to execute
    tByte RunMe;
} sTask;

```

```

// ----- Public function prototypes -----
// Core scheduler functions
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);

// ----- Public constants -----
// The maximum number of tasks required at any one time
// during the execution of the program
//
// MUST BE ADJUSTED FOR EACH NEW PROJECT
#define SCH_MAX_TASKS (1)

#endif

/*-----*
----- END OF FILE
-----*/

```

Listing 36.5 Part of a generic 255-tick scheduler example

```

/*-----*
SCH51a.C (v1.00)

-----*
*** THESE ARE THE CORE SCHEDULER FUNCTIONS ***
*** 255-TICK VERSION ***
--- These functions may be used with all 8051 devices ---
*** SCH_MAX_TASKS *must* be set by the user ***
--- see "Sch51.h" ---
*** Includes power-saving mode ***
--- You *MUST* confirm that the power-down mode is adapted ---
--- to match your chosen device (usually only necessary with
--- Extended 8051s, such as c515c, c509, etc ---
-----*/
#include "Main.h"
#include "Port.h"
#include "Sch51a.h"

```

```

// ----- Public variable definitions -----
// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];

// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
tByte Error_code_G = 0;

// ----- Private function prototypes -----
static void SCH_Go_To_Sleep(void);

// ----- Private variables -----
// Keeps track of time since last error was recorded (see below)
static tWord Error_tick_count_G;

// The code of the last error (reset after ~1 minute)
static tByte Last_error_code_G;

/*-----*
 * SCH_Dispatch_Tasks()
 *
 * This is the 'dispatcher' function. When a task (function)
 * is due to run, SCH_Dispatch_Tasks() will run it.
 * This function must be called (repeatedly) from the main loop.
 *-----*/
void SCH_Dispatch_Tasks(void)
{
    tByte Index;

    // Dispatches (runs) the next task (if one is ready)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        if (SCH_tasks_G[Index].RunMe > 0)
        {
            (*SCH_tasks_G[Index].pTask)(); // Run the task

            SCH_tasks_G[Index].RunMe -= 1; // Reset / reduce RunMe flag

            // Periodic tasks will automatically run again
            // - if this is a 'one shot' task, remove it from the array
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }
}

```

```

        }
    }

// Report system status
SCH_Report_Status();

// The scheduler enters idle mode at this point
SCH_Go_To_Sleep();
}

/*-----*
SCH_Add_Task()

```

Causes a task (function) to be executed at regular intervals or after a user-defined delay

Fn_P - The name of the function which is to be scheduled.

NOTE: All scheduled functions must be 'void, void' - that is, they must take no parameters, and have a void return type.

DELAY - The interval (TICKS) before the task is first executed

PERIOD - If 'PERIOD' is 0, the function is only called once, at the time determined by 'DELAY'. If PERIOD is non-zero, then the function is called repeatedly at an interval determined by the value of PERIOD (see below for examples which should help clarify this).

RETURN VALUE:

Returns the position in the task array at which the task has been added. If the return value is SCH_MAX_TASKS then the task could not be added to the array (there was insufficient space). If the return value is < SCH_MAX_TASKS, then the task was added successfully.

Note: this return value may be required, if a task is to be subsequently deleted - see SCH_Delete_Task().

EXAMPLES:

Task_ID = SCH_Add_Task(Do_X,1000,0);

Causes the function Do_X() to be executed once after 1000 sch ticks.

Task_ID = SCH_Add_Task(Do_X,0,1000);

Causes the function Do_X() to be executed regularly, every 1000 sch ticks.

```

Task_ID = SCH_Add_Task(Do_X,300,1000);
Causes the function Do_X() to be executed regularly, every 1000 ticks.
Task will be first executed at T = 300 ticks, then 1300, 2300, etc.

*-----
tByte SCH_Add_Task(void * pFunction(),
                   const tByte DELAY,
                   const tByte PERIOD)
{
    tByte Index = 0;

    // First find a gap in the array (if there is one)
    while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
    {
        Index++;
    }

    // Have we reached the end of the list?
    if (Index == SCH_MAX_TASKS)
    {
        // Task list is full
        //
        // Set the global error variable
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;

        // Also return an error code
        return SCH_MAX_TASKS;
    }

    // If we're here, there is a space in the task array
    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; // return position of task (to allow later deletion)
}

*-----
SCH_Delete_Task()

Removes a task from the scheduler. Note that this does
*not* delete the associated function from memory:
it simply means that it is no longer called by the scheduler.

```

TASK_INDEX - The task index. Provided by SCH_Add_Task().

RETURN VALUE: RETURN_ERROR or RETURN_NORMAL

- *-----*/

```
bit SCH_Delete_Task(const tByte TASK_INDEX)
{
    bit Return_code;

    if (SCH_tasks_G[TASK_INDEX].pTask == 0)
    {
        // No task at this location...
        //
        // Set the global error variable
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;

        // ...also return an error code
        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;           // return status
}
```

/ *-----*/

SCH_Report_Status()

Simple function to display error codes.

This version displays code on a port with attached LEDs:
adapt, if required, to report errors over serial link, etc.

Errors are only displayed for a limited period
(60000 ticks = 1 minute at 1ms tick interval).
After this the the error code is reset to 0.

This code may be easily adapted to display the last
error 'for ever': this may be appropriate in your
application.

See Chapter 10 for further information.

```
-----*/
void SCH_Report_Status(void)
{
#ifdef SCH_REPORT_ERRORS
    // ONLY APPLIES IF WE ARE REPORTING ERRORS
    // Check for a new error code
    if (Error_code_G != Last_error_code_G)
    {
        // Negative logic on LEDs assumed
        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; // Reset error code
            }
        }
    }
#endif
}

/*-----*/
SCH_Go_To_Sleep()

This scheduler enters 'idle mode' between clock ticks
to save power. The next clock tick will return the processor
to the normal operating state.
```

Note: a slight performance improvement is possible if this function is implemented as a macro, or if the code here is simply pasted into the 'dispatch' function.

However, by making this a function call, it becomes easier - during development - to assess the performance of the scheduler, using the 'performance analyser' in the Keil hardware simulator. See Chapter 14 for examples of this.

*** May wish to disable this if using a watchdog ***

*** ADAPT AS REQUIRED FOR YOUR HARDWARE ***

```
/*-----*/
void SCH_Go_To_Sleep()
{
    PCON |= 0x01;      // Enter idle mode (generic 8051 version)

    // Entering idle mode requires TWO consecutive instructions
    // on 80c515 / 80c505 - to avoid accidental triggering
    // PCON |= 0x01;      // Enter idle mode (#1)
    // PCON |= 0x20;      // Enter idle mode (#2)
}

/*-----*
----- END OF FILE -----
*-----*/
```

Listing 36.6 Part of a generic 255-tick scheduler example

```
/*-----*
----- LED_flas.H (v1.00)
-----*/

- See LED_flas.C for details.

/*-----*
----- Public function prototypes -----
-----*/
void LED_Flash_Init(void);
void LED_Flash_Update(void);

/*-----*
----- END OF FILE -----
*-----*/
```

Listing 36.7 Part of a generic 255-tick scheduler example

```
/*-----*
 LED_flas.C (v1.00)

-----
 Simple 'Flash LED' test function for scheduler.

-*-----*/
#include "Main.h"
#include "Port.h"
#include "LED_flas.h"

// ----- Private variable definitions -----
static bit LED_state_G;

/*-----*
 LED_Flash_Init()

 - See below.

-*-----*/
void LED_Flash_Init(void)
{
    LED_state_G = 0;
}

/*-----*
 LED_Flash_Update()

 Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

 Must schedule at twice the required flash rate: thus, for 1 Hz
 flash (on for 0.5 seconds, off for 0.5 seconds) must schedule
 at 2 Hz.

-*-----*/
void LED_Flash_Update(void)
{
    // Change the LED from OFF to ON (or vice versa)
    if (LED_state_G == 1)
    {
        LED_state_G = 0;
        LED_pin = 0;
    }
    else
```

```
{  
    LED_state_G = 1;  
    LED_pin = 1;  
}  
}  
  
/*-----*  
--- END OF FILE -----*  
-*-----*/
```

Listing 36.8 Part of a generic 255-tick scheduler example

Further reading

ONE-TASK SCHEDULER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application is to have a time-triggered architecture, constructed using a scheduler.

Problem

How do you create an application that is required to run only a single task, using a minimum of memory and CPU resources?

Background

Solution

CO-OPERATIVE SCHEDULER [page 255] can be used as the basis of single-task applications and we have frequently used it for this purpose. However, sometimes you have a simple application and need to squeeze all you can from the limited CPU and memory resources available in an 8051 microcontroller; in these circumstances, use of a full-blown co-operative scheduler environment may not be appropriate.

There is a simple solution that is compatible with the majority of the patterns in this book: this involves basing your application on a single task and calling this task directly by implementing it as a timer ISR.

Listing 36.9 illustrates the approach.

```
/*-----*  
Main.c  
-----*  
  
Simple timer ISR demonstration program  
*-----*/  
  
#include <AT89S53.h>  
  
#define INTERRUPT_Timer_2_Overflow 5  
  
// Function prototype  
// NOTE: ISR is not explicitly called and does not require a prototype  
void Timer_2_Init(void);
```

```

/* -----
void main(void)
{
    Timer_2_Init(); // Set up Timer 2

    EA = 1;          // Globally enable interrupts

    while(1);        // An empty Super Loop
}

/* -----
void Timer_2_Init(void)
{
    // Timer 2 is configured as a 16-bit timer,
    // which is automatically reloaded when it overflows
    //
    // This code (generic 8051/52) assumes a 12 MHz system osc.
    // The Timer 2 resolution is then 1.000 µs
    // (see Chapter 11 for details)
    //
    // Reload value is FC18 (hex) = 64536 (decimal)
    // Timer (16-bit) overflows when it reaches 65536 (decimal)
    // Thus, with these setting, timer will overflow every 1 ms
    T2CON = 0x04;    // Load Timer 2 control register
    T2MOD = 0x00;    // Load Timer 2 mode register

    TH2     = 0xFC;   // Load Timer 2 high byte
    RCAP2H = 0xFC;   // Load Timer 2 reload capt. reg. high byte
    TL2     = 0x18;   // Load Timer 2 low byte
    RCAP2L = 0x18;   // Load Timer 2 reload capt. reg. low byte

    // Timer 2 interrupt is enabled, and ISR will be called
    // whenever the timer overflows - see below.
    ET2     = 1;

    // Start Timer 2 running
    TR2    = 1;
}

/* -----
void X(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // This ISR is called every 1 ms
}

```

```

    // Place required code here...
}

/*-----*
 *----- END OF FILE -----*
 *-----*/

```

Listing 36.9 The framework of an application using a timer ISR to invoke a regular task

Hardware resource implications

We consider the hardware resource implications under three main headings: timers, memory and CPU load.

Timer

This pattern requires one hardware timer. If possible, this should be a 16-bit timer, with auto-reload capabilities (such as Timer 2).

Memory and CPU load

The scheduler will consume no significant CPU resources: short of implementing the application as a **SUPER LOOP** [page 162] (with all the disadvantages of this rudimentary architecture), there is generally no more efficient way of implementing your application in a high-level language.

Reliability and safety implications

This approach can be both safe and reliable, provided that you do not attempt to ‘shoe-horn’ a multitask design into this single-task framework.

Overall strengths and weaknesses

- ☺ An efficient environment for running a single task at regular intervals.
- ☺ Only appropriate for applications which can be implemented cleanly using a single task.

Related patterns and alternative solutions

The main alternative to this ‘slim’ scheduler is a **SUPER LOOP** [page 162] architecture. **ONE-TASK SCHEDULER** can be particularly effective if used in combination with **MULTI-STATE TASK** [page 322].

Example: Assessing the load of a one-task scheduler

We illustrate the use of a one-task scheduler in Listings 36.10 and 36.11.

```
/* -----
Port.H (v1.00)

-----
'Port Header' (see Chap 10) for the project ONE_TASK (see Chap 36)

----- */
// ----- Sch51.C -----
// Comment this line out if error reporting is NOT required
// #define SCH_REPORT_ERRORS

#ifndef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif

// ----- LED_Flas.C -----
// Connect LED from +5V (etc) to this pin, via appropriate resistor
// [see Chapter 7 for details]
sbit LED_pin = P1^5;

/* -----
----- END OF FILE -----
----- */
*/
```

Listing 36.10 Part of an implementation of ONE-TASK SCHEDULER

```
/* -----
Main.c (v1.00)

-----
One-task scheduler demonstration program
- See Chapter 36 for details.

----- */
#include "Main.H"
#include "Port.H"
```

```
#define INTERRUPT_Timer_2_Overflow 5

// Global variable
static tByte LED_state_G;

// Function prototypes
// NOTE: ISR is not explicitly called and does not require a prototype
void Timer_2_Init(void);
void LED_Flash_Init(void);
void Go_To_Sleep(void);

/* ----- */

void main(void)
{
    Timer_2_Init();      // Set up Timer 2
    LED_Flash_Init();   // Prepare to flash LED
    EA = 1;              // Globally enable interrupts
    while(1)             // Super Loop
    {
        Go_To_Sleep();  // Enter idle mode to save power
    }
}

/* ----- */

void Timer_2_Init(void)
{
    // Timer 2 is configured as a 16-bit timer,
    // which is automatically reloaded when it overflows
    //
    // This code (generic 8051/52) assumes a 12 MHz system osc.
    // The Timer 2 resolution is then 1.000 µs
    // (see Chapter 11 for details)
    //
    // Reload value is FC18 (hex) = 64536 (decimal)
    // Timer (16-bit) overflows when it reaches 65536 (decimal)
    // Thus, with these setting, timer will overflow every 1 ms
    T2CON    = 0x04;    // Load Timer 2 control register
    T2MOD    = 0x00;    // Load Timer 2 mode register
    TH2      = 0xFC;    // Load Timer 2 high byte
    RCAP2H   = 0xFC;    // Load Timer 2 reload capt. reg. high byte
    TL2      = 0x18;    // Load Timer 2 low byte
    RCAP2L   = 0x18;    // Load Timer 2 reload capt. reg. low byte
```

```

// Timer 2 interrupt is enabled, and ISR will be called
// whenever the timer overflows - see below.
ET2      = 1;

// Start Timer 2 running
TR2     = 1;
}

/*-----*/
LED_Flash_Init()
- See below.

/*-----*/
void LED_Flash_Init(void)
{
    LED_state_G = 0;
}

/*-----*/
LED_Flash_Update()
Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

Code assumes this function will called every 1 ms.
The LED will flash at 0.5Hz (on for 1 second, off for 1 second)

/*-----*/
void LED_Flash_Update(void) interrupt INTERRUPT_Timer_2_Overflow
{
    // This ISR is called every 1 ms
    // - only want to update the LED every second
    static data tWord Call_count;

    TF2 = 0; // Reset the T2 flag

    if (++Call_count < 1000)
    {
        return;
    }

    Call_count = 0;

    // Change the LED from OFF to ON (or vice versa)
    // (Do this every second)
    if (LED_state_G == 1)
    {

```

```

        LED_state_G = 0;
        LED_pin = 0;
    }
else
{
    LED_state_G = 1;
    LED_pin = 1;
}
}

/* -----
Go_To_Sleep()

This one-task scheduler enters 'idle mode' between clock ticks
to save power. The next clock tick will return the processor
to the normal operating state.

Note: a slight performance improvement is possible if this
function is implemented as a macro, or if the code here is simply
pasted into the 'dispatch' function.

However, by making this a function call, it becomes easier
- during development - to assess the performance of the
scheduler, using the 'performance analyser' in the Keil
hardware simulator. See Chapter 14 for examples of this.

*** May wish to disable this if using a watchdog ***

*** ADAPT AS REQUIRED FOR YOUR HARDWARE ***

- *----- */
void Go_To_Sleep(void)
{
    PCON |= 0x01;      // Enter idle mode (generic 8051 version)

    // Entering idle mode requires TWO consecutive instructions
    // on 80c515 / 80c505 - to avoid accidental triggering
    //PCON |= 0x01;    // Enter idle mode (#1)
    //PCON |= 0x20;    // Enter idle mode (#2)
}

/* -----
--- END OF FILE ---
- *----- */

```

Listing 36.11 Part of an implementation of ONE-TASK SCHEDULER

If you compare the scheduler load in this example (Figure 36.1) with the corresponding figures for single-task, 12-MHz 8051s in Chapter 14 (e.g. Figure 14.4), you will see that this one-task version is considerably more efficient. Specifically, this one-task scheduler would – even on a very basic 8051 – allow the use of 0.1 ms tick intervals.

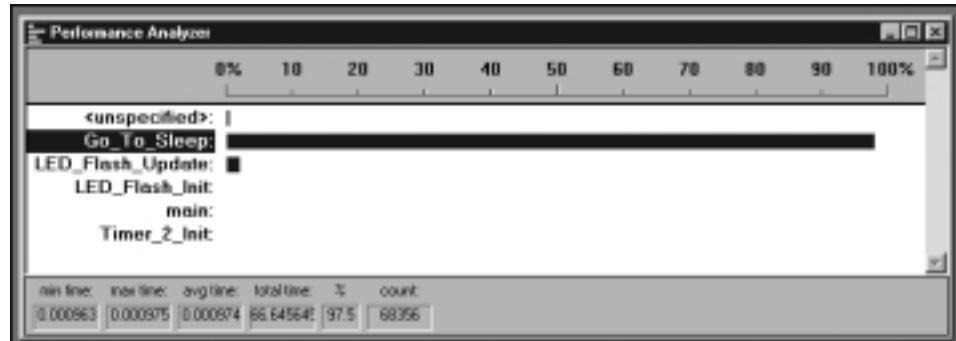


FIGURE 36.1 Using the Keil hardware simulator to evaluate the CPU load imposed by a one-task scheduler with 1 ms ticks, running on a 12 MHz (12 oscillations per instruction) 8051

[Note: The test reveals that the CPU is 97% idle and that the maximum possible task duration is therefore approximately 0.97 ms.]

Further reading

ONE-YEAR SCHEDULER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application is to have a time-triggered architecture, constructed using a scheduler.
- The application is to be powered by batteries.

Problem

How do you create a battery-powered application that can operate for up to 12 months between battery changes?

Background

In this pattern, we will consider how we can use a scheduler to create a battery-powered embedded application that can operate for up to 12 months between battery changes.

We provide some key background details in this section.

Primary and secondary cells

When we talk about a battery, we mean an electrochemical device that converts chemical energy into electricity, by means of one or more galvanic cells.¹ A galvanic cell is a comparatively simple device consisting of two electrodes (an anode and a cathode) and an electrolyte solution. The first form of such a cell was created by the Italian Alessandro Volta in 1800 (Figure 36.2). This cell consists of a copper rod cathode (the positive electrode), a zinc rod anode (the positive electrode) and an electrolyte consisting of a weak solution of sulphuric acid.

Briefly, the cell operates as follows. The metal in the zinc anode oxidizes (that is, it ‘rusts’ or ‘dissolves’) into the electrolyte solution, while hydrogen molecules from the electrolyte are deposited on the cathode. When the anode is fully oxidized (or the cathode is fully reduced) the chemical reaction will stop and the battery is considered to be discharged.

In some cases, it is possible to ‘recharge’ the battery. Recharging usually involves applying a voltage across the plates to reverse the battery’s chemical process. Some chemical reactions, however, are difficult or impossible to reverse. Cells with irreversible reactions are commonly known as *primary cells*, while cells with reversible

1. Strictly, a battery consists of at least two galvanic cells (a ‘battery of cells’). However, in popular usage, the term is used to refer to devices containing one or more cells.

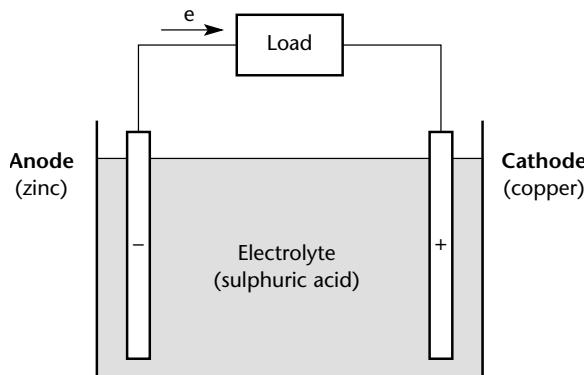


FIGURE 36.2 The first form of 'battery': a galvanic cell, as created by Alessandro Volta in 1800

reactions are known as *secondary cells*. Although, with care, it is sometimes possible to perform a partial recharge of devices designed to be used as primary cells, this is seldom very effective and is frequently dangerous.

Battery characteristics

Numerous different combinations of electrode and electrolyte are used in modern battery designs. These combinations result in cells which have different capacities, often presented in 'milli-Amp-hours' (mAh). For example, a cell might have a capacity of 1,000 mAh. In an ideal world, this would mean that the cell could generate a current of 1,000 mA for 1 hour before becoming discharged to the point that it was no longer capable of powering our application: alternatively, the same ideal battery could generate 1 mA for 1,000 hours before reaching this state. In reality, a battery does not have a single capacity rating that may be applied at all current levels. For example, a 1.5V (size D) alkaline 'battery' might typically have a rated capacity of 10,000 mAh at 10 mA (continuous) and reduced capacities of 8,000 mAh at 100 mA and 4,000 mAh at 1,000 mA. Of course, many embedded systems do not operate continuously and have varying current requirements while they are being used. As a result, predicting the likely lifetime of batteries used in equipment with widely varying current requirements is particularly difficult and the only practical solution is to estimate the likely needs from published manufacturer data sheets and then to test a prototype of the system under real conditions.

A second key characteristic of batteries is the discharge curve (Figure 36.3). Ideally, with microcontroller applications, we wish to have a flat discharge curve, so that the battery maintains its output voltage throughout its working life. If, instead of this flat discharge characteristic, we have a sloping characteristic, the cell voltage can very quickly fall below the level at which we will be able to operate our microcontroller.

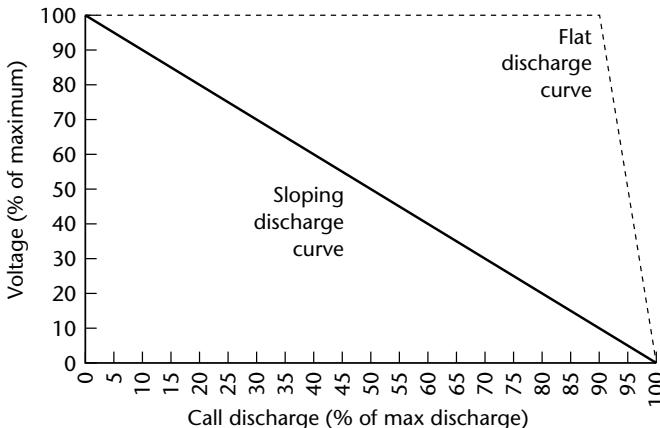


FIGURE 36.3 Idealized 'flat' and 'sloping' discharge characteristics

[Note: that, in reality, most batteries have characteristics that lie somewhere between these two extremes.]

Alkaline cells

A particularly popular type of cell used in modern embedded applications is the 'alkaline' cell. This battery is popular because it has – compared with conventional Leclanche dry cells ('ordinary batteries') – a flatter discharge curve and a higher cell capacity.

Typical figures for the service hours you can expect for common 1.5V alkaline cells (adapted from Duracell data sheets) are given in Table 36.1.

TABLE 36.1 Typical operating life for alkaline cells

Cell type	Average current	Operating life (hours)
Alkaline AAA	245	3
"	40	30
"	15	80
Alkaline AA	275	5
"	50	50
"	15	200
Alkaline D	475	20
"	235	50
"	115	130

Two useful rules of thumb:

- You can expect a shelf life of 2–4 years from alkaline batteries.
- AA cells are particularly popular and are widely available throughout the world. These are appropriate for many applications. The ubiquitous Duracell MN1500, for example, has a rating of 1850 mAh. At low currents (an average of around 0.3 mA), you can expect to get at least a year of life from such cells.

Obtaining the required voltage

Each cell gives 1.5V (nominal): four cells will therefore give you 6V. Note that this does not alter the current rating: 1 MN1500 AA cell has a rating of 1.5V, 1,850 mAh, while four have a rating of 6V, 1,850 mAh.

Solution

We will particularly focus on the use of AA-sized alkaline cells as the power source, as this is a popular and effective option. To do this then – for the reasons outlined in ‘Background’ – we need to reduce the average current consumption to 0.3 mA.

As we noted in Chapter 3 (see Table 3.1, page 36), the typical current consumption for a Standard 8051 microcontroller (operating normally) is around 15 mA. At this current level, our batteries will last for around 200 hours – around 8 days. Even if the application spends most of its time in idle mode, the current consumption will still be around 5 mA: with luck, our application may operate for almost a month.

However, these figures do not tell the full story. In particular, they hide the fact that current consumption is heavily dependent on the oscillator frequency, and operating voltage. For example, if you consider the datasheet for the Atmel 89S53 (included on the CD), it will be apparent that the current consumption varies with voltage and oscillator frequency. More specifically, by considering the datasheets for the 89S53 or a wide range of other 8051 devices, it becomes apparent that, to obtain our average of 0.3mA current consumption, we must run at a voltage of around 3V, and at a very low oscillator frequency (around 1 MHz, preferably less). In addition, we must keep the application in idle mode most of the time.

A cost-effective way of obtaining the required oscillator frequency is to use a watch crystal. These are cheap, small and widely available: they have frequencies of 32.768 kHz. Note that to use this frequency you must work with an 8051 that can operate in this frequency range: check your data sheet.

Working with AAA cells

The AA cells considered in detail in this pattern have a weight of 23.6 g or 0.83 oz. (Duracell MN1500 data sheet). To reduce the weight (and size) of your application, you may wish to consider using AAA cells, which have a weight of 11.0 g. or 0.39 oz. (Duracell MN2400 data sheet). These cells have a capacity of 1,150 mAh (Duracell MN2400 data sheet).

1,150 mAh translates into an average current of around 0.13 mA over a year. It is now possible to obtain this level of current consumption from an 8051 device using the Small 8051s and particularly the modern Philips devices. For example, the Philips 87LPC764 has very low current consumption in both active and idle states, particularly at low frequencies, as the data sheet (included on the CD ROM) makes clear.

From the data sheet, it is apparent that even in active mode (again assuming use of a 32 kHz watch crystal), the 87LPC768 draws only around 0.1 mA, while in idle mode this figure falls to around 0.05 mA. Provided we do not need to drive high-power external components, then these devices are ideal as the basis of lightweight embedded applications.

Working with 9V batteries

Another popular lightweight battery is the 9V 'radio' battery, with useful push-on connector terminals. These have a capacity of around 500 mAh (Duracell MN1604 data sheet), which translates into 0.06 mA average current over a year. The recent Small 8051s (described earlier) can, almost match these requirements. However, in most 8051-based applications, you will be unlikely to obtain more than around six months of life from such a battery.

Working with D cells

If you require a longer operating life and / or higher current than AA cells can provide, D cells may be an alternative. These comparatively large cells have a capacity of around 15,000 mAh (Duracell MN1300 data sheet). Over one year, this translates into an average current of 1.7 mA; over two years, this becomes 0.8 mA.

You can reliably obtain two years of life from D cells in embedded applications and may even manage 3 years if your current requirements are very limited. For applications where the user is not expect to carry the product around this can be a good solution.

D cells are particularly attractive in applications such as long-term, unattended data logging. Although the D cells are large and heavy, the alternative may be a lead-acid secondary cell and a solar-powered battery charger, which will be even heavier, more complex and considerably more expensive.

Working with lithium cells

Lithium cells have been introduced comparatively recently and have many characteristics that make them very attractive for use as primary cells in many battery-powered embedded systems.

Key characteristics of this type of cell or battery:

- ☺ **Lithium cells have very low self-discharge rates: this means they can last for many years (a decade is not uncommon) 'on the shelf'.**
- ☺ **When being used, lithium batteries can last around three times longer than alkaline batteries of the same size. This makes them ideal as, for example, CMOS backup batteries in desktop computers.**
- ☺ **Lithium cells weigh around 30% less than equivalent alkaline cells.**
- ☺ **Unlike most other battery technology, lithium cells operate effectively over a wide temperature range. Alkaline cells operate at around 50–60% of their normal level at 0°C, whereas lithium cells operate normally at this temperature. Lithium cells continue to provide an output around 80% of their normal level even at temperatures below -20°C, whereas alkaline cells are unusable at such temperatures.**
- ☺ **Lithium primary cells are available in standard (AA) sizes, 1.5V.**
- ☺ **Lithium reacts violently when exposed to water and there were initial safety concerns about batteries based on this metal. Recent designs have greatly reduced such problems, but you need to be aware of this potential safety concern.**

- (?) Lithium cells are not suitable for high-current applications (for example, driving motors, or other such loads).
- (?) Lithium batteries cost around twice as much as alkaline batteries of the same size, so may not always prove cost effective.
- (?) Lithium batteries are, currently, not as widely available as alkaline equivalents.

Overall, given these characteristics, it seems inevitable that, as prices fall, lithium cells will become increasingly common in embedded applications.

Working with secondary cells

Rechargeable batteries are useful in many embedded applications, such as mobile phones, but cannot retain their charge for long enough to match the needs of the present pattern.

Hardware resource implications

This pattern is only suitable for use in applications which require limited CPU activity and are able to spend most of their operational life in idle mode.

Reliability and safety implications

Typical one-year applications will be awakened from idle mode every 200 ms (say) to check a switch or perform some other processing. While in idle mode, they will not be able to respond to external events.

This type of architecture is common, for example, in TV remote-control applications and is appropriate for this purpose. This architecture may not be appropriate for safety-related applications where very rapid responses are required.

Portability

This is a form of **CO-OPERATIVE SCHEDULER** [page 255] and is as portable as the original pattern.

Overall strengths and weaknesses

- (?) Reduces battery consumption to a low level.
- (?) Limited processing ability as a result of low clock frequencies.
- (?) May not be appropriate for safety-related applications.

Related patterns and alternative solutions

This pattern can often be combined with **255-TICK SCHEDULER** [page 894] or **ONE-TASK SCHEDULER** [page 911] to reduce the resource requirements further.

Example: Automatic light

We are required to develop a new battery-powered cupboard (closet) light. It is required to operate as follows:

- Press the switch once: a small bulb is lit.
- Press the switch again: the bulb goes out.

Of course, these requirements may be met very easily without resorting to the use of a microcontroller. However, many people placing this type of device in a cupboard forget to turn out the light after use: as a result, the batteries are drained in the course of a couple of hours. In this type of application the cost of the batteries will represent a substantial percentage of the purchase price and the frequent need to replace the batteries is a cause for concern.

We wish to create an application that operates as follows:

- Press the switch while the light is off: the light turns on.
- Press the switch while the light is on: the light turns off.
- If the bulb is lit and the switch is not pressed again, the light will go out after 30 seconds.
- The battery life should be ‘as long as possible’.

Figure 36.4 shows a possible hardware design.

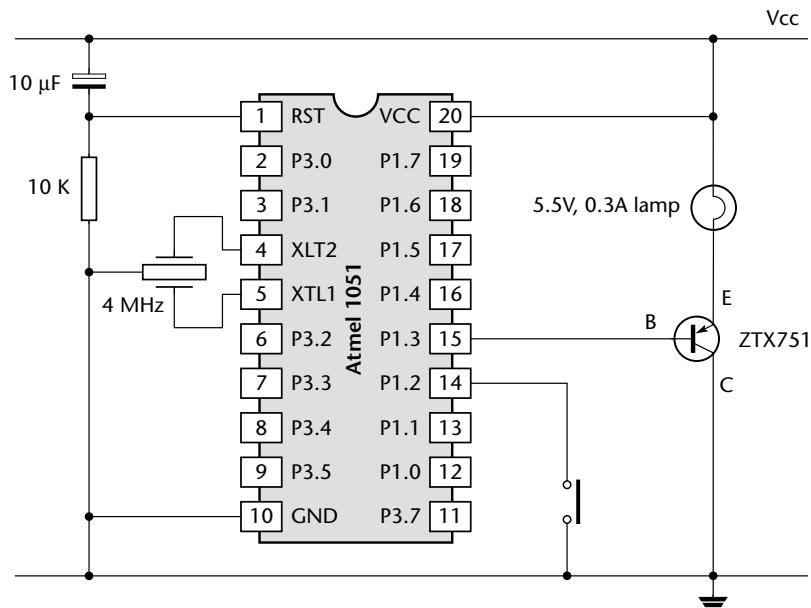


FIGURE 36.4 A hardware schematic for an automatic-light example.

The code for the completed system is given in Listing 36.12.

```
/*-----*
Main.c (v1.00)

-----
Automatic light example
-----*/
#include "Main.h"

// ----- Port pins -----
// Omit Port.H in this simple (one-file) example
// Don't use pins 1^0, 1^1 - NO INTERNAL PULL-UP RESISTORS
sbit Switch_pin_G = P1^2;
sbit Light_pin_G = P1^3;

// ----- Private function prototypes -----
// Function prototypes
// NOTE: ISR is not explicitly called and does not require a prototype
void Timer_1_Init(void);
void Timer_1_Manual_Reload(void);
void Light_Init(void);

// ----- Private constants -----
#define SWITCH_PRESSED 0
#define LIGHT_ON 0
#define LIGHT_OFF 1

// ----- Private variable definitions-----
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(); // Set up Timer 2
    Light_Init(); // Prepare to flash the LED
    EA = 1; // Globally enable interrupts
}
```

```

while(1)
{
    PCON |= 0x01; // Go to sleep (idle mode)
}
}

/* ----- */

void Timer_1_Init(void)
{
// Timer 1 is configured as a 16-bit timer,
// which is manually reloaded when it overflows
TMOD &= 0x0F; // Clear all T1 bits (T0 left unchanged)
TMOD |= 0x10; // Set required T1 bits (T0 left unchanged)

// Sets up timer reload values
Timer_1_Manual_Reload();

// Interrupt Timer 1 enabled
ET1 = 1;
}

/* ----- */

Timer_1_Manual_Reload()

This 'One-Year Scheduler' uses a (manually reloaded) 16-bit timer.
The manual reload means that all timings are approximate.
THIS SCHEDULER IS NOT SUITABLE FOR APPLICATIONS WHERE
ACCURATE TIMING IS REQUIRED!!!
Timer reload is carried out in this function.

/* ----- */

void Timer_1_Manual_Reload(void)
{
// Stop Timer 1
TR1 = 0;

// This code (generic 8051/52) assumes a 4 MHz system osc.
// The Timer 1 resolution is then 0.000003 seconds
// (see Chapter 11 for details)
//
// We want to generate an interrupt every 200 ms (approx):
// this takes 0.2 / 0.000003 timer increments
// i.e. 66666 timer increments
//
// Reload value of 0x00 gives 65536 increments, which is
// sufficiently close for our purposes here (around 2% out)
TL1 = 0x00;
TH1 = 0x00;
}

```

```
// Start Timer 1
TR1 = 1;
}

/*-----*/
Light_Init()

/*-----*/
void Light_Init(void)
{
    Switch_count_G = 0;
    Auto_switch_off_count_G = 0;
    Switch_blocked_G = 0;
    // Write 1 to switch pin (to set it up for reading)
    Switch_pin_G = 1;
}

/*-----*/
Check_Switch()

/*-----*/
void Check_Switch(void) interrupt INTERRUPT_Timer_1_Overflow
{
    // This function is an implementation of the pattern On-Off Switch

    // If the light is on, 'Auto_switch_off_count' will be > 0
    // Decrement here - and switch the light off when it reaches zero.
    if (Auto_switch_off_count_G > 0)
    {
        Auto_switch_off_count_G--;
        if (Auto_switch_off_count_G == 0)
        {
            Light_pin_G = LIGHT_OFF;
        }
    }

    // The switch is 'blocked' after each switch press,
    // to give the user time to remove their finger:
    // If this is not done, the light will switch off again
    // when the user presses the switch for more than 0.4 seconds.
    //
    // If the switch is blocked, decrement the block count and return
    // without checking the switch pin status.
    if (Switch_blocked_G > 0)
    {
        Switch_blocked_G--;
    }
}
```

```
    return;
}

// Now read switch pin
if (Switch_pin_G == SWITCH_PRESSED)
{
    // If the switch pin is pressed, increment the switch count.
    if (++Switch_count_G == 2)
    {
        // If Switch_count_G == 2, this means that the pin has been
        // active
        // for two consecutive calls to this task, i.e. it is a
        // genuine switch press rather than a bounce.
        // The variable Auto_switch_off_count_G acts both as
        // an indication the light is on (if it is non-zero)
        // and a counter of the number of task calls the
        // light will remain on for.
        if (Auto_switch_off_count_G > 0)
        {
            // The light is currently ON
            // -> switch it off.
            Light_pin_G = LIGHT_OFF;
            Auto_switch_off_count_G = 0;
        }
    else
        {
            // The light is currently OFF
            // -> switch it on and set the counter to 150
            // (task is called every 0.2s so this gives 30 seconds
            // delay).
            Light_pin_G = LIGHT_ON;
            Auto_switch_off_count_G = 150;
        }

        // Reset the switch count, and block the switch for the next
        // second (5 calls to this task).
        Switch_count_G = 0;
        Switch_blocked_G = 5;
    }
}
else
{
    Switch_count_G = 0;
}
```

```
}

/* -----
--- END OF FILE -----
*/
```

Listing 36.12 Code for controlling an automatic light

Further reading

chapter **37**

Increasing the stability of the scheduling

Introduction

All crystal oscillators and ceramic resonators have frequency outputs which vary with temperature. As a result, schedulers driven by such clock sources also vary their timing behaviour with temperature. **STABLE SCHEDULER** is a temperature-compensated scheduler that adjusts its behaviour to take into account changes in ambient temperature.

STABLE SCHEDULER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you ensure that your scheduled application continues to operate with correct timing even if the ambient temperature varies?

Background

See **CRYSTAL OSCILLATOR** [page 54] for a discussion of the impact of temperature on the performance of crystal oscillator circuits.

Solution

In most practical applications where accurate timing is required, the system will derive its oscillator from some form of crystal oscillator. Unfortunately, the stability of these oscillators varies with temperature. To obtain stable scheduling, we have two main options:

- Maintain the system at a constant, known temperature and adapt all timer settings to match this. This might be achieved by placing the application in some form of industrial oven or in an ice bath. For most applications, this type of approach is impractical.
- Measure the ambient temperature and adjust the system timing to take into account any temperature changes.

STABLE SCHEDULER adopts this second approach.

Hardware resource implications

The main hardware implication is that we require some form of temperature sensor. In most situations, a digital sensor (such as one of the Dallas Semiconductor 1620 family) is a cost-effective choice, since it is itself cheap and requires at most three microcontroller pins.

A particularly good alternative is an on-chip temperature sensor: some of the Cygnal² and Analog Devices³ ranges of 8051 microcontrollers include such a sensor, making them ideal for this application.

The load imposed by the scheduling software is very small, with one temperature measurement made at infrequent (typically 1-minute) intervals.

Reliability and safety implications

Use of a stable scheduler can improve the reliability of your application if accurate timing over long periods is required.

Portability

Most microcontrollers can be linked to some form of external temperature sensor and, therefore, can apply this pattern.

Overall strengths and weaknesses

- ☺ Improves the scheduler stability.
- ☹ Usually increases costs.

Related patterns and alternative solutions

Example: Using a external Dallas DS1621 I²C temperature sensor

In this example, we present a simple stable scheduler, using an external (DS1621) temperature sensor (Listings 37.1 and 37.2).

The demonstration program displays time on an LED display; refer to Chapter 21 for suitable display hardware and the CD for a complete set of code listings.

```
/* ----- * -  
Main.c (v1.00)  
-----  
Demonstration program for:  
Stable Scheduler  
Drives 4 multiplexed multi-segment LED displays  
- displays elapsed time.
```

2. www.cygnal.com
3. www.analog.com

```
Required linker options (see Chapter 13 for details):  
  
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)  
{  
    // Set up the scheduler  
    SCH_Init_T2();  
  
    // Prepare for temperature measurements  
    I2C_Init_Temperature_DS1621();  
  
    // Add the 'Time Update' task (once per second)  
    // - timings are in ticks (1 ms tick interval)  
    // (Max interval / delay is 65535 ticks)  
    SCH_Add_Task(CLOCK_LED_Time_Update,100,10);  
  
    // Add the 'Display Update' task (once per second)  
    // Need to update a 4-segment display every 3 ms (approx)  
    // Need to update a 2-segment display every 6 ms (approx)  
    SCH_Add_Task(LED_MX4_Display_Update,0,3);  
  
    // This is scheduled once per minute  
    SCH_Add_Task(SCH_Calculate_Ave_Temp_DS1621,33,60000);  
  
    // Start the scheduler  
    SCH_Start();  
  
    while(1)  
    {  
        SCH_Dispatch_Tasks();  
    }  
}
```

```
/*-----*  
---- END OF FILE -----  
-*-----*/
```

Listing 37.1 Part of the implementation of a simple stable scheduler, using an external (DS1621) temperature sensor

```
/*-----*  
2_01_12s.C (v1.00)  
-----  
*** THIS IS A STABLE SCHEDULER FOR STANDARD 8051 / 8052 ***  
*** Uses T2 for timing, 16-bit auto reload ***  
*** 12 MHz oscillator -> 1 ms (precise) tick interval ***  
*** Assumes DS1621 temperature sensor available ***  
  
#include "2_01_12s.h"  
#include "I2C_1621.h"  
  
// ----- Public variable definitions -----  
  
// The current temperature, recorded every hour  
tByte Temperature_G;  
  
// ----- Public variable declarations -----  
  
// The array of tasks (see Sch51.C)  
extern sTask SCH_tasks_G[SCH_MAX_TASKS];  
  
// The error code variable  
//  
// See Port.H for port on which error codes are displayed  
// and for details of error codes  
extern tByte Error_code_G;  
  
// Running total / average temperature (calculated every 24 hours)  
static int Temperature_average_G = 0;  
  
// Called every minute: only take reading once an hour  
// (calling every hour requires changes to scheduler,  
// increasing the required memory for EVERY task).  
static tByte Minute_G;  
static tByte Hour_G;  
  
// The temperature compensation data  
//
```

```

// The Timer 2 reload values (low and high bytes) are varied depending
// on the current average temperature.
//
// NOTE (1):
// Only temperature values from 10 - 30 celsius are considered
// in this version
//
// NOTE (2):
// Adjust these values to match your hardware!
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()

Scheduler initialization function. Prepares scheduler
data structures and sets up timer interrupts at required rate.
Must call this function before using the scheduler.

*-----*/
void SCH_Init_T2(void)
{
    tByte i;

    for (i = 0; i < SCH_MAX_TASKS; i++)
    {
        SCH_Delete_Task(i);
    }

    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    //   (because the task array is empty)
    Error_code_G = 0;

    // Now set up Timer 2
    // 16-bit timer function with automatic reload

```

```

// Crystal is assumed to be 12 MHz
// The Timer 2 resolution is 0.000001 seconds (1 µs)
// The required Timer 2 overflow is 0.001 seconds (1 ms)
// - this takes 1000 timer ticks
// Reload value is 65536 - 1000 = 64536 (dec) = 0xFC18

T2CON = 0x04;    // load Timer 2 control register
T2MOD = 0x00;    // load Timer 2 mode register

TH2      = 0xFC;  // load Timer 2 high byte
RCAP2H = 0xFC;   // load Timer 2 reload capture reg, high byte
TL2      = 0x18;  // load Timer 2 low byte
RCAP2L = 0x18;   // load Timer 2 reload capture reg, low byte

ET2      = 1;    // Timer 2 interrupt is enabled

TR2      = 1;    // Start Timer 2
}

```

/*-----*-

SCH_Start()

Starts the scheduler, by enabling interrupts.

NOTE: Usually called after all regular tasks are added,
to keep the tasks synchronized.

NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

-*-----*/

void SCH_Start(void)

```

{
EA = 1;
}
```

/*-----*-

SCH_Update()

This is the scheduler ISR. It is called at a rate
determined by the timer settings in the 'init' function.

This version is triggered by Timer 2 interrupts:
timer is automatically reloaded.

-*-----*/

void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow

```

{
tByte Index;
```

```

    TF2 = 0; // Have to manually clear this.

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        //
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            //
            if (SCH_tasks_G[Index].Delay == 0)
            {
                //
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Inc. the 'RunMe' flag

                if (SCH_tasks_G[Index].Period)
                {
                    //
                    // Schedule periodic tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
            else
            {
                //
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }

    /*-----*/
    SCH_Calculate_Ave_Temp_DS1621()

    This function should be scheduled once per minute.

    Based on 1 measurement per hour,
    this function updates a variable (Temperature_average_G)
    with the average temperature over the last 24 hours.

    The updates are carried out every 24 hours.

    /*-----*/
    void SCH_Calculate_Ave_Temp_DS1621(void)
    {
        if (++Minute_G == 60)
        {
            Minute_G = 0;
        }
    }
}

```

```

// An hour has elapsed - take temperature reading
I2C_Read_Temperature_DS1621();

// Add current reading to running total
Temperature_average_G += Temperature_G;

if (++Hour_G == 24)
{
    // 24 hours have elapsed - get average temperature
    Hour_G = 0;
    Temperature_average_G /= 24;

    // Update the scheduler
    SCH_Perform_Temperature_Adjustment();
}

}

/*-----*/
SCH_Perform_Temperature_Adjustment()

This scheduler adjusts its timing to take into account
changes in ambient temperature.

/*-----*/
void SCH_Perform_Temperature_Adjustment(void)
{
    static int Previous_temperature_average_G;

    if ((Previous_temperature_average_G - Temperature_average_G) != 0)
    {
        // Only consider temperatures in range 10 - 30 Celsius in this
        // version (easily adjusted)
        if (Temperature_average_G < 10)
        {
            Temperature_average_G = 10;
        }
        else
        {
            if (Temperature_average_G > 30)
            {
                Temperature_average_G = 30;
            }
        }
    }
}

```

```
ET2 = 0; // Disable interrupt
TR2 = 0; // Stop T2

// Reload the timer
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;
}

/*-----
----- END OF FILE -----
----- */
```

Listing 37.2 Part of the implementation of a simple stable scheduler, using an external (DS1621) temperature sensor

Further reading



Conclusions

We draw the book to a close with two final chapters.

- In Chapter 38, we review what we have tried to achieve through presentation of this pattern collection.
- In Chapter 39, we present a collected list of books and papers either directly referenced in the text or of related interest.

chapter **38**

What this book has tried to do

38.1 Introduction

In this chapter we review what we have tried to achieve through presentation of this pattern collection.

38.2 What this book has tried to do

Many current embedded applications are developed in one of two ways:

- Using no specific system architecture and, often, multiple interrupt-service routines.
- Using a commercial, pre-emptive operating system.

In many ways, this echoes what happens in most parts of the research community and in the rest of the world, in that:

- Most published code for embedded systems (on the WWW, for example), as well as manufacturer application notes for microcontrollers and associated components, tends to be informally written, with great emphasis placed on the use of interrupts.
- Much academic research in this area tends to focus on technical and theoretical analyses of complex, pre-emptive, applications (see Chapter 13 for further details). Where case studies are presented, these often seem to bear little relation to the types of problems faced in real projects.

Throughout this book, we have sought to steer a middle path between these two areas. Specifically, we have sought to demonstrate – for a range of real-world embedded applications – that the use of a co-operative scheduler with a single interrupt source (per microcontroller) is a practical proposition, even for distributed applications based on multiple, 8-bit microcontrollers with very limited CPU and memory resources.

We have also tried to demonstrate, informally, that applications based on this co-operative architecture have very predictable behaviour and, hence, are an appropriate basis for safety-related applications. We have argued that predictable and reliable behaviour is not just an essential requirement in safety-related applications, but is also appropriate in all embedded systems where designers and developers are concerned with the quality of the products they are producing.

The simple nature of these schedulers also provides other benefits. For example, it means that developers themselves can, very rapidly, port the scheduler onto a new microcontroller environment. It also means that the basic architecture may be readily adapted to meet the needs of a particular application, without altering the basic (co-operative) approach (as we sought to demonstrate in Part H).

However, perhaps the most important side-effect of these simple schedulers is that – unlike a traditional ‘real-time operating system’ – they become part of the application itself (Figure 38.1), rather than forming a separate code layer between the user code and the application (Figure 38.2).

In our experience, this tight integration of scheduler and application means that developers quickly understand and claim ownership of the scheduler code. This is important, since it avoids a ‘not invented here’ or ‘blame it on the OS’ philosophy,

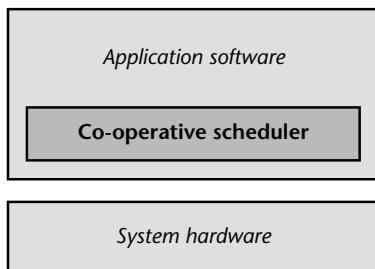


FIGURE 38.1 A co-operative scheduler (of the type discussed in this book) becomes part of the application

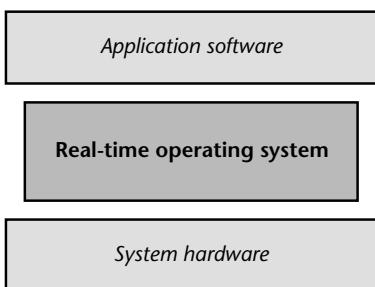


FIGURE 38.2 A ‘real-time operating system’ remains a separate application

which can arise when developers must interface their code to a large and complex real-time operating system, the features and behaviour of which they may never fully understand.

38.3 Conclusions

In this chapter we have briefly reviewed what we have tried to achieve through presentation of this pattern collection.

This brings us to the end of the book. To explore further the topics discussed in this chapter and in earlier chapters, refer to the sources of information listed in Chapter 39.

Collected references and bibliography

In this chapter we present a collected list of books and papers which are either directly referenced in the text, or of related interest.

39.1 Complete list of publications

- Acornley, 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, F.P. (1975) *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, MA.
- Brooks, F.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) *Modern 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.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, 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 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]
- 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 Jöhnk, 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 Övergaard, 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 José, 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 Mellor, 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 Other pattern collections

- 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 Design techniques for real-time / embedded systems

- 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 Mellor, 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 Design techniques for high-reliability systems

- 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 The 8051 microcontroller

- 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 Related publications by the author

- 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.



Appendices

There are three appendices:

- Appendix A provides details of the design notation used throughout this book.
- Appendix B provides information about the contents of the CD.
- Appendix C provides information about the WWW site associated with the book.

The design notation and CASE tool

Overview

This appendix will provide a brief description of the design notation used throughout the book and the CASE tool used to produce the associated figures.

The CASE tool

The diagrams throughout this book were created using the Select 'Yourdon' CASE tool. This product is now distributed by Aonix Corporation.¹

Note that a copy of this CASE tool is included with the book *Software Engineering with C++ and CASE Tools* (Pont, 1996). This is a full copy of the product, but has an educational licence and may not be used for commercial purposes.

The notation

The notation used in this book is fully described elsewhere (Pont, 1996). Briefly, the design is described in a series of layers, starting from an initial high-level design (in the form of a context diagram) and ending with some form of process specification (which will, typically, be implemented as a C function).

The design process also covers aspects such as the creation of the user interface and testing of the components; these issues are not considered here.

The following series of figures (Figures A1.1–A1.6), taken from Pont (1996) illustrate some of the key documents for the design of a bank auto-teller machine (Figure A1.1), culminating in a code framework (in desktop C++ in this case: Listing A1.1).

1. www.aonix.com

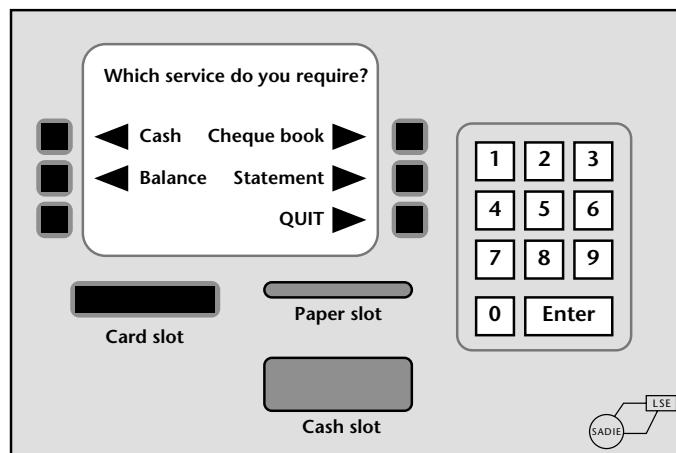


FIGURE A1.1 The ATM interface

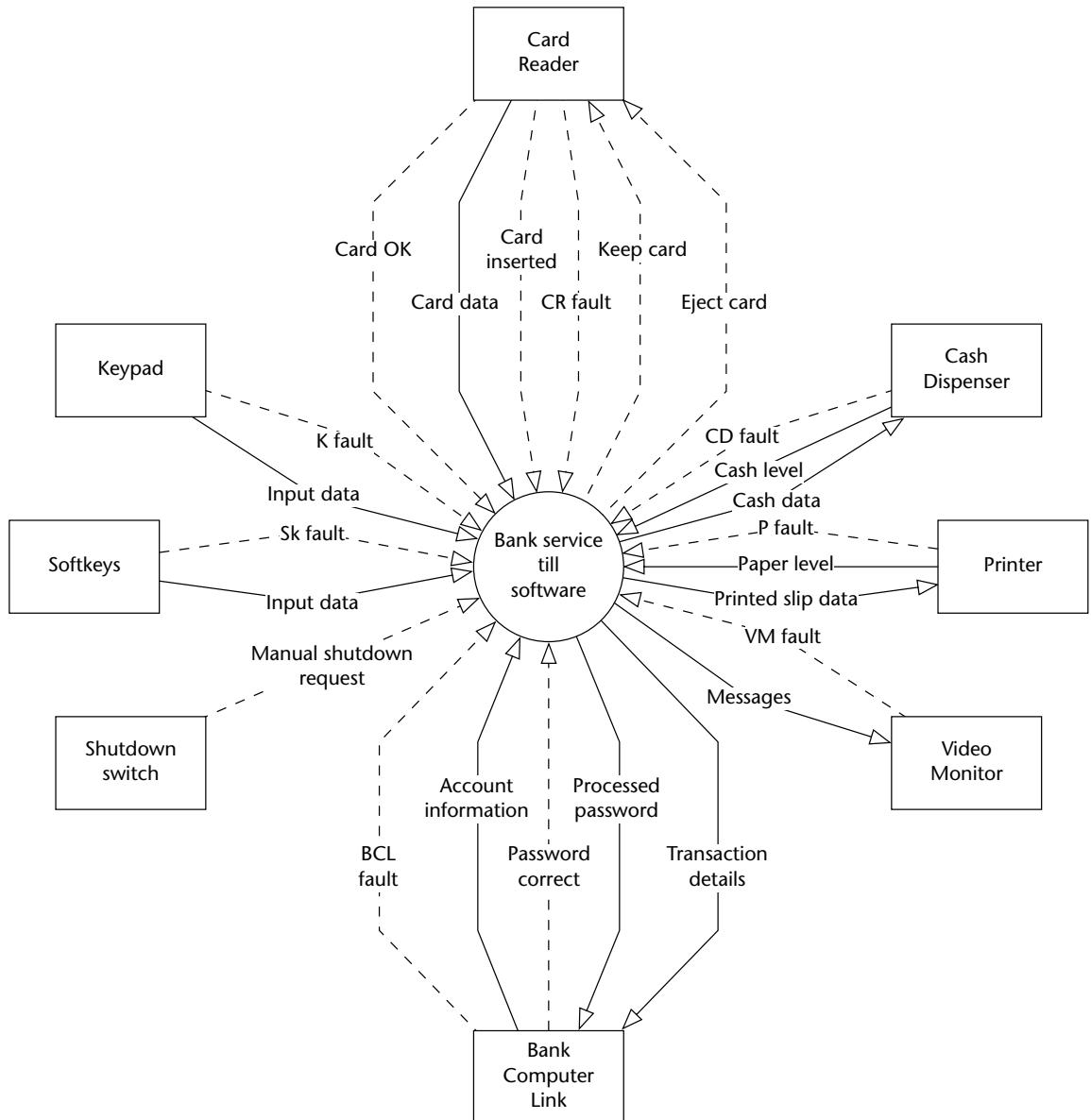


FIGURE A1.2 The context diagram

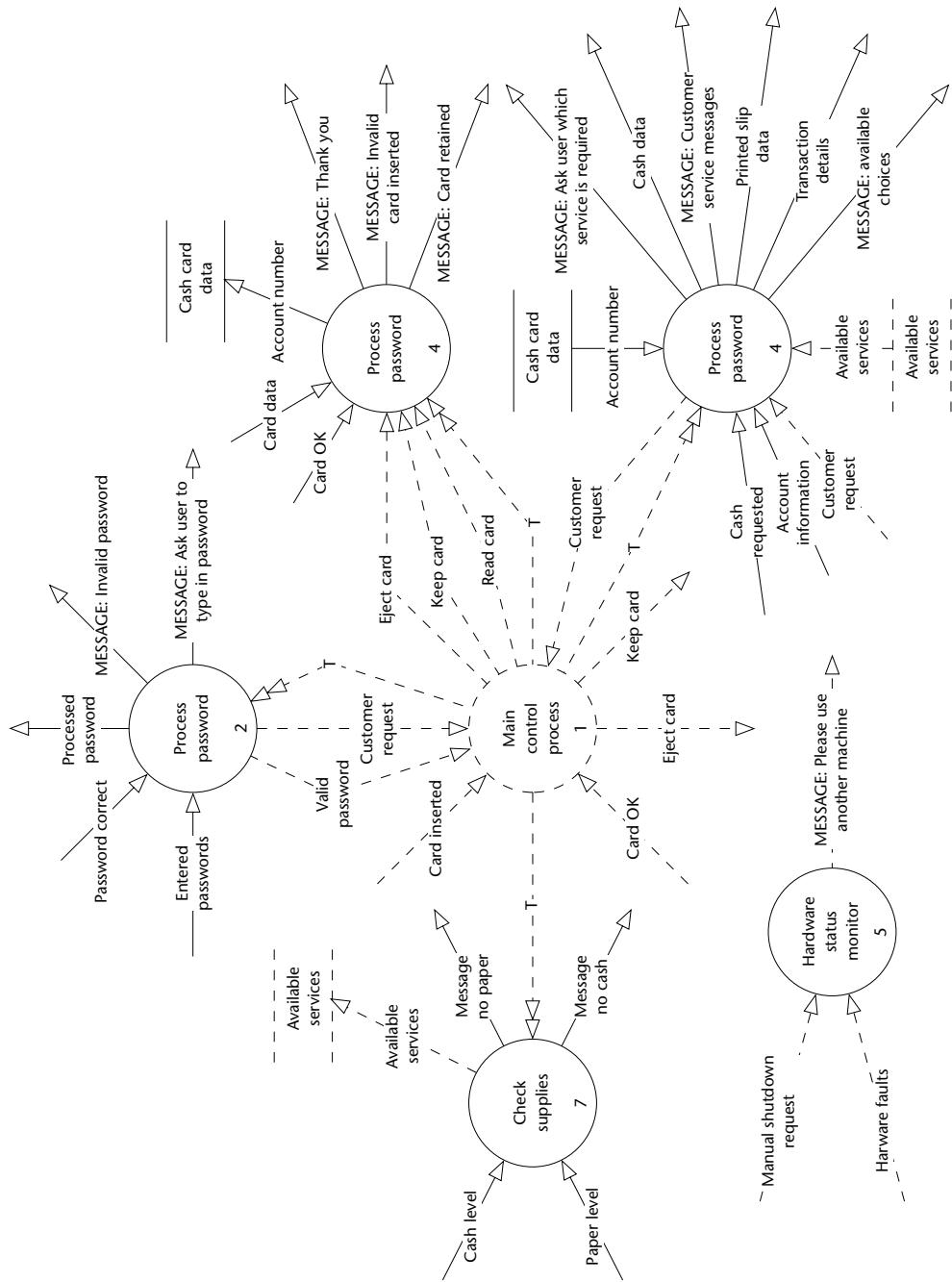


FIGURE A1.3 The Level 1 DfD

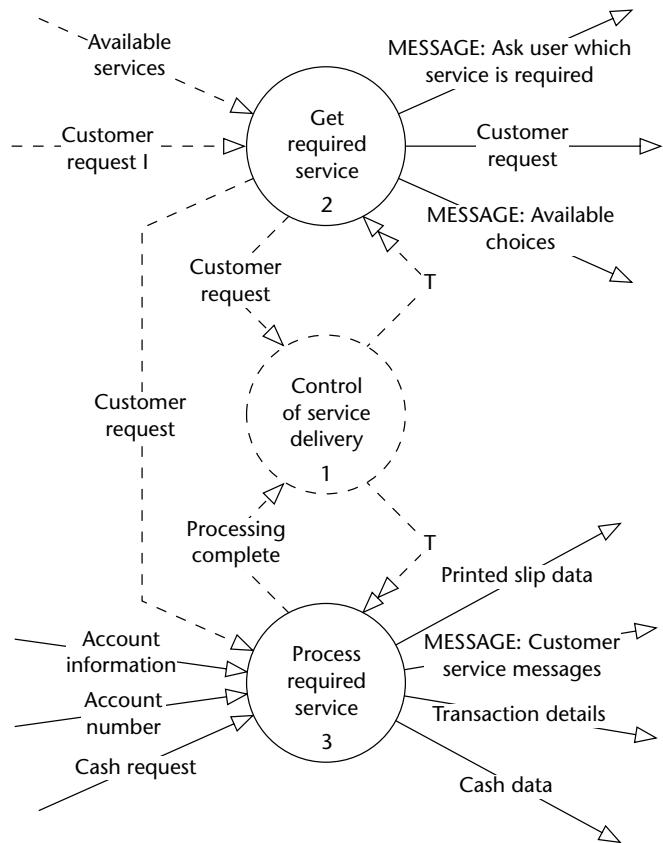


FIGURE A1.4 DfD 1-3

**FIGURE A1.5 DfD 1-3-3**

```

@IN   = Available services
@IN   = Customer request I
@OUT  = Customer request
@OUT  = MESSAGE: Ask user which service is required
@OUT  = MESSAGE: Available choices

@PSpec Get required service

// Generates simple menu (containing only the available options)
// and returns a valid choice

Pre-condition:
    None

(MESSAGE: Ask user which service is required) = 'Which service do you require?'
(MESSAGE: Available choices) = list of (Available services)
(Customer request) = (Customer request I)

```

Post-condition:
 (Customer request) is a valid choice

@

FIGURE A1.6 PSpec 1-3-2

```

/*****
*
* A Bank Service Till (ATM) Simulation (Process-Oriented)
*
*/
#include <assert.h>
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

// Maximum number of tries allowed to enter correct password
const int MAX_NUM_PASS_TRIES = 3;

// Various "flags"
const int TRUE = 1;
const int FALSE = 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;

// Function prototypes
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_to_quit, tries;

    // Set up non-repeatable random numbers for simulation
    srand((unsigned)time(NULL));

    // Simulate (up to) five customers in queue then
    // manual shut down
```

```

for (int customer = 1; customer <= 5; customer++)
{
    // Welcome banner
    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))
        {
            // Keep delivering services till user says quit
            // (or hardware fault)
            Deliver_services(cheque_OK, balance_OK,
                            statement_OK, cash_OK);
            Process_cash_card(EJECT_CARD);
        }
        else
        {
            // No valid password
            if (user_wants_to_quit)
            {
                // Polite ejection
                Process_cash_card(EJECT_CARD);
            }
        }
    }

    // Simulate manual shutdown
    Shutdown();
    return 0;
}

```

```

*****
*
*   FUNCTION: Shutdown()
*
*   OVERVIEW: Exit message, and exit program.
*
*   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()
*
*   OVERVIEW: Check cash and paper reserves &c.
*
*   PRE:      No hardware faults.
*
*   POST:     Got valid data.
*
*   RETURNS:  void.
*
*****/
void Check_supplies(int& cash_OK_REF,
                    int& balance_OK_REF,
                    int& statement_OK_REF,
                    int& cheque_OK_REF)
{
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }

    int paper_level_OK; // Is paper level okay? (flag)
}

```

```

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
{
    // Reset these for new customer
    balance_OK_REF = TRUE;
    statement_OK_REF = TRUE;
}

// Reset flag for new customer (always available)
cheque_OK_REF = TRUE;

// Post - got valid data
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()
*
*   OVERVIEW: Make sure hardware is not damaged.
*
*   PRE:      None.
*
*   POST:     None.
*
*   RETURNS:  TRUE (1) if fault, FALSE (0) otherwise.
*
********************************************/

```

```
int Hardware_fault_detected()
{
// Pre - none

// Simulating a hardware fault with a prob. of 1/100
if ((rand() % 100) == 0)
{
    cerr << "**** SIMULATING HARDWARE FAULT ***\n";
    return 1;
}
else
{
    return 0;
}

// Post - none
}

/******************
*
*   FUNCTION: Check_cash_reserves()
*
*   OVERVIEW: Check cash level.
*
*   PRE:      No hardware faults.
*   POST:     Got valid data.
*
*   RETURNS:  void.
*
*************************/
void Check_cash_reserves(int& cash_OK_REF)
{
// Pre - no hardware faults
if (Hardware_fault_detected()) { Shutdown(); }

// Simulating a 1/6 chance of a cash shortage
if ((rand() % 6)==0)
{
    cash_OK_REF = FALSE;
    cerr << "**** SIMULATING LOW CASH LEVEL ***\n";
}
else
{
    cash_OK_REF = TRUE;
}
```

```

// Post - got valid data
assert((cash_OK_REF == FALSE)
       || (cash_OK_REF == TRUE));
}

/***** *
*      *
*  FUNCTION: Check_paper_reserves()      *
*      *
*  OVERVIEW: Check that there is paper to print on.      *
*      *
*  PRE:      No hardware faults.      *
*  POST:     Got valid data.      *
*      *
*  RETURNS: void.      *
*      *
*****/
void Check_paper_reserves(int& paper_level_OK_REF)
{
    //
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }

    // Simulating a 1/6 chance of a paper shortage
    if ((rand() % 6)==0)
    {
        paper_level_OK_REF = FALSE;
        cerr << "**** SIMULATING LOW PAPER LEVEL ***\n";
    }
    else
    {
        paper_level_OK_REF = TRUE;
    }

    // Post - got valid data
    assert((paper_level_OK_REF == FALSE)
           || (paper_level_OK_REF == TRUE));
}

/***** *
*      *
*  FUNCTION: Process_cash_card()      *
*      *
*  OVERVIEW: Validate, eject or retain cash card.      *
*      *
*****/

```

```
*          *
*      PRE:      No hardware faults.          *
*                  Valid operation requested.      *
*      POST:     None.          *
*
*      RETURNS: void.          *
*
*****void Process_cash_card(int required_op, int* card_OK_PTR)
{
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }

    // Pre - valid operation requested
    assert((required_op == CHECK_CARD) ||
           (required_op == KEEP_CARD) ||
           (required_op == EJECT_CARD));

    /* NOTE: Process_cash_card() is a function with a
       default argument. Can call as:
       Process_cash_card(CHECK_CARD, card_OK);
       or
       Process_cash_card(KEEP_CARD);
       Process_cash_card(EJECT_CARD); */

    switch (required_op)
    {
        case CHECK_CARD:
            // Using "extra" argument

            cout << "Please insert your card.\n";
            // Validate user's card
            *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";
    cerr << "*** SIMULATING EJECTING CARD ***\n\n";
    break;
}

// Post - none
}

/*
*   FUNCTION: Get_password()
*
*   OVERVIEW: Get and validate user's password
*
*   PRE:      No hardware faults.
*   POST:     Got valid data.
*
*   RETURNS:  void.
*
*/
void Get_password(int& password_OK_REF,
                  int& want_to_quit_REF,
                  int& tries_REF)
{
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }

    int password;

    tries_REF++;
    cout << "Please enter your password"
}

```

```
<< "\n(or 9999 to simulate QUIT button): ";
cin >> password;

if (!(password == 9999))
{
    // User did not ask to quit

    // Simulate call to bank computer
    cerr << "**** SIMULATING CALL TO BANK COMPUTER ****\n";

    // Simulating one single valid password...
    // In reality, send account number and password
    // to bank computer
    // Computer returns "YES" or "NO".

    if (password == 1234)
    {
        password_OK_REF = TRUE;
    }
    else
    {
        password_OK_REF = FALSE;
    }

    if (!(password_OK_REF))
    {
        if (tries_REF < MAX_NUM_PASSTRIES)
        {
            cout << "Password incorrect.\n";
            // Recursive call to Get_password()
            Get_password(password_OK_REF, want_to_quit_REF, tries_REF);
        }
        else
        {
            // Too many tries - could be a stolen card?
            Process_cash_card(KEEP_CARD);
        }
    }
    else
    {
        cout << "Thank you.\n\n";
    }
}
else
{
```

```

        // User wants to quit
        want_to_quit_REF = TRUE;
    }

    // Post - got valid data
    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_PASSTRIES);
}

/***** *
*      *
* FUNCTION: Deliver_services()      *
*      *
* OVERVIEW: Ask user for reqd. service and deliver it.      *
*      *
* PRE:      No hardware faults.      *
*          Got valid data.      *
* POST:     None.      *
*      *
* RETURNS: void.      *
*      *
*****/
void Deliver_services(int& cheque_OK_REF,
                      int& balance_OK_REF,
                      int& statement_OK_REF,
                      int& cash_OK_REF)
{
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }

    // Pre - got valid data
    assert((cash_OK_REF == FALSE)
           || (cash_OK_REF == TRUE));
    assert((cheque_OK_REF == FALSE)
           || (cheque_OK_REF == TRUE));
    assert((balance_OK_REF == FALSE)
           || (balance_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 (user_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()
*
*   OVERVIEW: Find out what service user requires.
*
*   PRE:      No hardware faults.
*   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 - no hardware faults
    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 - got valid choice
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()  *
*      *
*  OVERVIEW: Perform user's requested service.  *
*      *
*  PRE:      No hardware faults.  *
*          Got valid choice of service.  *
*  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 - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }

    // Pre - got valid choice
    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()
*
*   OVERVIEW: Order cheque book.
*
```

```

*   PRE:      No hardware faults.          *
*           Got valid data.            *
*   POST:     Returning valid data.        *
*           *
*   RETURNS:  void.                      *
*           *
***** */

void Order_cheque_book_from_bank(int& cheque_OK_REF)
{
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }
    // Pre - got valid data
    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 - returning valid data
    assert((cheque_OK_REF == FALSE));
}

***** */

*           *
*   FUNCTION: Print_customer_balance()          *
*           *
*   OVERVIEW: Print customer's balance.        *
*           *
*   PRE:      No hardware faults.          *
*           Got valid data.            *
*   POST:     Returning valid data.        *
*           *
*   RETURNS:  void.                      *
*           *
***** */

void Print_customer_balance(int& balance_OK_REF)
{
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }
    // Pre - got valid data
    assert((balance_OK_REF == TRUE));

    cout << "Your balance is: 99.99 (*** SIMULATED ***)\n\n";

```

```
balance_OK_REF = FALSE;

// Post - returning valid data
assert((balance_OK_REF == FALSE));
}

/********************* *
*      FUNCTION: Print_customer_mini_statement()
*
*      OVERVIEW: Print mini statement.
*
*      PRE:      No hardware faults.
*              Got valid data.
*      POST:     Returning valid data.
*
*      RETURNS: void.
*
***** */

void Print_customer_mini_statement(int& statement_OK_REF)
{
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }

    // Pre - got valid data
    assert((statement_OK_REF == TRUE));

    cerr << "**** SIMULATING PRINTING OF MINI STATEMENT ****";
    cout << "\nPlease take your mini statement.\n\n";

    statement_OK_REF = FALSE;

    // Post - returning valid data
    assert((statement_OK_REF == FALSE));
}

/********************* *
*      FUNCTION: Dispense_cash_to_customer()
*
*      OVERVIEW: Deliver cash.
*
*      PRE:      No hardware faults.
*              Got valid data.
*      POST:     Returning valid data.
*
***** */
```

```
*      RETURNS: void.          *
*
***** */
void Dispense_cash_to_customer(int& cash_OK_REF)
{
    // Pre - no hardware faults
    if (Hardware_fault_detected()) { Shutdown(); }
    // Pre - got valid data
    assert((cash_OK_REF == TRUE));

    cerr << "**** SIMULATING CASH DELIVERY ***\n";
    cout << "Please take your cash.\n\n";

    cash_OK_REF = FALSE;

    // Post - returning valid data
    assert((cash_OK_REF == FALSE));
}

/*
*** END OF PROGRAM ***
***** /
```

Listing A1.1 The prototype ATM system software

Guide to the CD

Overview

This short appendix provides a guide to the contents of the CD.

The basis of the CD

The CD is based on information kindly provided by Keil Software; this includes the evaluation version of all the Keil compilers, plus numerous data sheets on 8051 microcontrollers.

The CD has a menu which will guide you through the installation process.

The source code for this book

The source code for this book is NOT automatically installed with the Keil compiler.

The source files are included on the CD in the directory \Pont. For example, if your CD-ROM drive is mapped to D: you will find the source files in the directory D:\Pont.

Use 'Windows Explorer' (or equivalent) to copy these files from the CD to your hard drive.

Copyright Restrictions

The code included in this book took many years to produce. It is not 'free ware' and is subject to some simple copyright restrictions. These are as follows:

- Having purchased a copy of this book, you are entitled to use the code listed in this book and included on the CD in your projects, should you choose to do so. If you use the code in this way, then no run-time royalties are due. However, the author would appreciate it if you acknowledged the source of the code in the product documentation.
- If there are ten developers in your team using code adapted from this book, please purchase ten copies of the book.
- You may not, **under any circumstances**, publish or otherwise distribute any of the source code included in the book or on the CD, in any form or by any means, without explicit written authorization from the author. If you wish to publish limited code fragments then, in most circumstances, this permission will be granted, subject only to an appropriate acknowledgement accompanying the published material. If you wish to publish more substantial code listings, payment of a fee may be required. Please contact the author for further details.

Guide to the WWW site

Overview

This brief appendix provides information about the WWW site associated with this book.

The URL

There is a WWW site associated with this book, at the following URL:

<http://www.engg.lse.ac.uk/books/Pont>

Contents of the WWW site

On this WWW site you will find:

- A set of detailed case studies describing the application of the techniques discussed in this book in a series of small and large projects
- Bug reports and code updates
- Further code samples
- Links to other relevant sites

Bug reports and code updates

There is huge amount of code involved in this project, both in the book itself and on the associated CD. The author has personally tested all the code that appears here. Nonetheless, errors can creep in.

If you think you have found a bug, please first check the WWW site to see if anyone else has picked up the error: if they have, a code correction will have been made available.

If you have found a bug not listed on the WWW site, please send an e-mail to the author explaining the bug (M.Pont@le.ac.uk) and he will do his best to help.

Anyone who spots a bug will be acknowledged (if they wish) in subsequent editions of the book.

Index

- 3-LEVEL PWM** 822–30
4-bit interface LCD character panel 471
7-segment LED 450–1
8XC520 (Dallas) 89, 98, 99, 276, 872
8XC552 (Philips) 91–2, 298
9V radio batteries 923
24C64 (Atmel) 496
80C390 (Dallas) 46, 48, 89
80c751 (Philips) 41, 90
87C550 (Dallas) 809
87LPC764 (Philips) 41, 61, 90, 922
89C52 (Atmel) 118
89C420 (Dallas) 55, 84
89C1051 (Atmel) 89
89C2051 (Atmel) 62, 75, 89
89C4051 (Atmel) 60, 89
89S53 (Atmel) 31–2, 89, 223–7, 524–36, 763, 922
255-TICK SCHEDULER 893, 894–910
1232 external timer 217, 219–22
8048 microcontroller 30
8051 microcontroller
 STANDARD 8051 30–4
 SMALL 8051 39, 41–5
 EXTENDED 8051 39, 46–52
 alternative solutions 39
 building your own 51–2
 clock speeds 34
 hardware components 35, 37, 38
 idle operating mode 36–7
 linking together 50, 540–1, 543–52
 memory architecture 34–5
 naming of family members 33
 oscillator cycles 33–4, 55
 performance levels 33–4
 pin count 35–6
 portability 38
 power consumption 36–7
power-down operating mode 37
reliability and safety 37–8
8052 microcontroller 30–1
80251 microcontroller 39
A-A FILTER (anti-aliasing filter) 794–801
 alternative solutions 800
 continuous-time filters 799
 over-sampling the signal 800
 portability 800
 reliability and safety 799
 switched-capacitor filters 798–9
AC loads 148–58
EMR DRIVER (electromagnetic relay) 149–55
inductive kick 152
inrush currents 152
RC snubber 153–4
SSR DRIVER (solid-state relay) 156–8
switching on/off 152–3
ACKNOWLEDGE signal (I2C) 499–500
active low inputs 75–6
AD421 (Analog Devices) 843
AD8517 (Analog Devices) 780
ADC see analogue-to-digital converters
ADC PRE-AMP 777–81
 level shifting circuits 778–9
 microphone pre-amplifier 780
 operational amplifiers 777
 portability 779
 reliability and safety 779
 voltage amplification 777–8, 780
Add Task function 265–6, 339
address bus 95
address bytes 611
address counter 470–1
address latch enable (ALE) 91, 95
air-conditioning systems 873
air-traffic control 861–2, 865
Airbus 543, 549
alarm clock application 20–1
alarm systems 841
ALE (address latch enable) 91, 95
Alexander, C. 22
aliasing
 A-A FILTER 794–801
 and digital-to-analogue converters (DACs) 844
 and **SEQUENTIAL ADC** 786
alkaline batteries 921–2
Allworth, S.T. 12, 251
amplifiers
 BJT-based amplifier circuit 858
 operational amplifiers 777, 797–8, 799, 858
 power amplifiers 858–9
 transconductance amplifier 843–4
 voltage amplification 777–8, 780
 see also **ADC PRE-AMP**
Analog Devices
 AD421 843
 AD8517 780
analogue-digital converter 760
digital-analogue converter 842, 843
EXTENDED 8051 46–7, 48
memory options 81, 89
temperature sensors 933
watchdog chips 217
analogue voltage measurement 757–61
analogue-to-digital converters (ADC) 731, 756–806
 A-A FILTER 794–801
 ADC PRE-AMP 777–81
 CURRENT SENSOR 802–6
 flash ADCs 787
 hardware options 759–61
 ONE-SHOT ADC 757–76
 SEQUENTIAL ADC 782–93
 successive-approximation ADCs 787

anti-windup protection 869–70
 asynchronous data transmission 364, 368, 520
 AT cut 54
 Atmel
 24C64 496
 89C542 118
 89C1051 89
 89C2051 62, 75, 89
 89C4051 60, 89
 89S53 31–2, 89, 223–7,
 524–36, 763, 922
 memory options 89
 SMALL 8051 41, 42
 watchdog timers 223–7
 atomic clocks 60
 audio equipment 858–9
 auditory-evoked responses 785
 auto-reload timers 197, 238–9
 automatic light 925–30
 autopilot applications 6–7
 Awad, M. 16, 17, 20
 Axelson, J. 370, 396
 Ayala, K. 39
 back lighting LCD character
 panels 472
 bandwidth
 and data transfer 712–15
 and **PID CONTROLLER** 871–2
 and **SEQUENTIAL ADC** 784–5
 bank-switched memory
 arrangements 104–7
 bargraph display 187–92
 Barnett, R.H. 251
 Barrensheen, J. 684, 685
 Basic CAN 678
 basic input/output system (BIOS)
 232
 Bates, I. 12, 251
 batteries
 alkaline 921–2
 D cell 923
 discharge curve 920–1
 galvanic cells 919, 920
 lithium cell 923–4
 9V 923
 predicting lifetime of 920
 primary cells 919
 recharging 919–20, 924
 secondary cells 920, 924
 voltage of 922
 baud rates
 PC LINK (RS-232) 364, 366,
 367–8, 386–96, 520

SCC SCHEDULER 681
SCI SCHEDULER (TICK) 562
SCU SCHEDULER (LOCAL)
 612–13
 and timer 2 743
 BDATA memory 86
 Beck, K. 22
 Bennett, S. 251
 Bignell, V. 552
 BIOS (basic input/output system)
 232
 bipolar-junction transistor (BJT)
 see **BJT**
 Bishop, R.H. 866, 873
 bit rate
 DAC OUTPUT 842
 SEQUENTIAL ADC 786
 bitwise operators 179–83
 BJT (bipolar-junction transistor),
 amplifier circuit 858
 BJT (bipolar-junction transistor),
 and current-sense resistors
 803–4
BJT DRIVER 124–33, 142
 buffering output 125–7
 inrush currents 128
 load faults 131
 portability 131
 reliability and safety 128–31
 switching off inductive DC
 loads 129–31
 blink control circuit 470
 Booch, G. 16
 bounce behaviour of switches
 399–400, 402, 410–11
 brightness control 823–30,
 834–9
 broadband signals 784–5
 Brooks, F.P. 25
 brownouts 73–4
 buffers
 and **BJT DRIVER** 125–7
 and connecting up LEDs 112
 and **HARDWARE PWM** 810
 IC BUFFER 118–2
 and **KEYPAD INTERFACE** 438,
 439
 logic families 120–2
 and multi-segment LED
 displays 451, 454
 UDN2585A 451–3, 454
 bug reports 982–3
 Burns, A. 543, 546
 Burr-Brown
 RCV420 761
 transconductance amplifier
 843–4
 XTR105 761
 XTR110 843–4
 business information systems
 (BISs) 3–5
 busy flag (BF) 470
 buzzers 116–17, 133
 bytes
 address bytes 611
 format in I2C protocol 499
 message bytes 611
 reading and writing 178–9
C167 (Infineon) 34
C501 (Infineon) 33, 34, 36,
 89–90
C505C (Infineon) 46, 90
C509 (Infineon) 90, 98, 872
C515C (Infineon) 46, 90, 93,
 386–96, 686–710, 760,
 776, 788–92, 809, 812–17
C517 (Infineon) 872
C541 (Infineon) 373
C8051F000 (Cyginal) 842–3
 cable connections (RS-232)
 365–6
 caesium beam clock 60
 CAN (controller area networks)
 545–7, 675–710
 capacitors 57, 65
 CASE tool 957
 central-heating systems 155,
 158, 165–8
HARDWARE WATCHDOG 221–2,
 224–7
CERAMIC RESONATOR 64–6
 connecting 66
 cost 64
 frequency variation and
 temperature 931
 portability 65
 reliability and safety 65
 stability 64
 Character Generator (CG) RAM
 470
 character set of LCD panels 469
 clock frequency/speed
 latch and memory
 combinations 97, 100–1
 and oscillator frequency 55–6
SPI PERIPHERAL 522–3
STANDARD 8051 34
 clock synchronization
 and I2C protocol 497, 500

and shared-clock schedulers 543–5
 clock-outs 741
 closed-loop control systems 864–5
 CMOS logic family 120–2
CO-OPERATIVE SCHEDULER 246–53, 255–96, 716
 compared to pre-emptive scheduler 250–2
 core scheduler library 280–96
 CPU load 274–6
 data structure 260–1
 error reporting 272–4
 function pointers 255–8, 268–70, 277
 functions
 add task 265–6, 339
 delete task 270–1
 dispatch task 266–8
 initialization 262–3
 report status 272–4
 sleep 271
 start 270
 update 263–4, 268, 336–7
 integration of scheduler and application 944–5
 interrupt generation 263
 Keil linker options 268–70
 key components 258
 and memory 274
 oscillator frequency 274–6
 portability 279
 power consumption 271
 reliability and safety 246, 276–9
 resource consumption 260, 274–6
 task array 261, 277
 task jitter 268
 task overlap 277, 278
 tick intervals 263, 278–9
 watchdog support 274
 co-operative thinking 297
 CODE memory 86, 88
 and interrupts 236
OFF-CHIP CODE MEMORY
 100–8
 speed of access 92
 code size
 I2C communication protocol 501
SPI PERIPHERAL 523
 code updates 982–3
 Coen, A. 9

Coleman, D. 16
 common anode packages 450
 common cathode packages 450
 communication systems for hydrofoils 853–5
 condition-monitoring applications 718, 723
 consecutively scheduled tasks 720–4
 context switch 247, 340
 continuous-time filters 799
 contrast adjustments 472
 control algorithms 865–72, 876
 control parameters 871
 controller area networks (CAN) 545–7, 675–710
 Cooling, J.E. 251
 copyright restrictions 981
 cost of licences
 I2C protocol 502
SPI PERIPHERAL 524
 counters
 address counter 470–1
 incrementing 424–32
 switch block counter 415
 see also timers
 critical sections of code 247–50, 338
 cruise-control system 17–20, 551, 874–8
 Crydom MP240D3 SSR 158
CRYSTAL OSCILLATOR 54–63
 connecting 57, 58
 to dual-processor boards 62, 63
 cost 57
 external modules 54, 58–9
 on-chip 60–1
 oscillator frequency 55–6, 57, 61–2
 and **PC LINK** (RS-232) 368–9, 372
 portability 60
 and **RC RESET** 73
 reliability and safety 58–60
 and RS-232 communication 368–9
 stability 56–7, 59–60
 start-up times 58
 temperature-compensated (TCXOs) 59, 931, 932
 Cunningham, W. 22
 current measurement 758–61
CURRENT SENSOR 802–6
 portability 805
 reliability and safety 805
 current sinks 134–6
 current-mode DACs 843–4
 current-mode sensor components 761
 current-sense resistors 802–3
 cursor/blink control circuit 470
 cyclic scheduling 251
 Cygnal
 C8051F000 842–3
 on-chip DACs 842–3
 temperature sensors 933
 Cypress Semiconductor 373
 D cell batteries 923
DAC DRIVER (digital-to-analogue converter driver) 857–9
 portability 857
 reliability and safety 857
DAC OUTPUT (digital-to-analogue converter output) 841–52
 aliasing 844
 alternative solutions 845
 bit rate 842
 external current-mode DAC 843–4
 external voltage-mode DAC 843
 frequency distortion 844
 on-chip DACs 842–3
 port pins and 844
 portability 844
 reliability and safety 844
 sample rate 842
 sinc compensation 854
 software architecture 844
 speech playback 845–52
 transconductance amplifier 843–4
DAC SMOOTHER (digital-to-analogue converter smoother) 853–6
 portability 856
 reliability and safety 855
 Dallas Semiconductor
 8XC520 98, 99, 276, 872
 80C390 46, 48
 87C550 809
 89C420 55, 844
 CAN support 678
 DS1050 809
 DS1620 932, 933–40
 DS1621 515
 Econoreset 77, 78, 79
 EXTENDED 8051 46, 48

Dallas Semiconductor *continued*
 fast 8051 devices 786–7
 high speed devices 34
 memory options 81, 89, 98–9
 pulse-width modulation
 devices 809
STANDARD 8051 34
 temperature sensors 515, 932,
 933–40
 watchdog chips 217
 Darlington arrangement 134,
 858
 data acquisition 541–3, 718
 data bus 95, 472
 Data bytes 611–12
 data lines 522
DATA memory 86–7, 88
OFF-CHIP DATA MEMORY 94–9
 speed of access 92
 data registers 470
 data structure
CO-OPERATIVE SCHEDULER
 260–1
255-TICK SCHEDULER 894–5
 see also message structure
 data transfer
 I2C protocol 498–500
 limited bandwidth 712–15
 shared-clock schedulers 545–6
SPI PERIPHERAL 522
 UART data transfer 235, 608,
 675
 see also message structure
DATA UNION 712–15
 portability 714
 reliability and safety 713–14
 DC loads 109–47
BJT DRIVER 124–33, 142
IC BUFFER 118–23
IC DRIVER 134–8
 inductive kick 129
 inrush currents 128
MOSFET DRIVER 139–43
NAKED LED 110–14
NAKED LOAD 115–17
 SSR driver 144–7
 switching on/off 128–31
 DC motor control 128, 143, 147,
 822, 861–4, 879–88
 DD (Display Data) RAM 467
 De Marco, T. 8, 16
 debouncing switches 399–400,
 402, 410–11
 delays
 generic delay code 200–5

HARDWARE DELAY 194–205
SOFTWARE DELAY 206–14
 Delete Task function 270–1
 design notation 957
 desktop systems 5–6, 162, 231–2
 see also **PC LINK (RS-232)**
 device addresses 496–7
 digital-to-analogue converters
 (DACs) 840–59
 conversion noise 853
DAC DRIVER 857–9
DAC OUTPUT 841–52
DAC SMOOTHER 853–6
 voltage mode DACs 843, 857
 diodes 129–30, 153
 direct addressing 82–3
 disaster recovery 218
 discharge curve of batteries
 920–1
 Dispatch task function 266–8
 Display Data (DD) RAM 467
 distributed networks 608, 646
 node wiring 684
 transceivers 683
 Dolphin Integration 51
DOMINO TASK 720–4
 portability 722
 reliability and safety 722
 Dorf, R.C. 866, 873
 Douglass, B.P. 16
 DRAM (dynamic RAM) 83
 DS1050 (Dallas) 809
 DS1620 (Dallas) 932, 933–40
 DS1621 (Dallas) 515
 Duracell 921
 Dutton, K. 866, 873
 duty cycles
HARDWARE PRM 742
HARDWARE PWM 808–9
 dynamic RAM (DRAM) 83
 EEPROM (electrically erasable
 programmable read-only
 memory) 85, 496, 503,
 510–15, 530–6
 electromagnetic interference
 (EMI) 151, 811, 820
 electromagnetic relays 144, 145
 electromechanical relays 149,
 152, 155
 electrostatic discharge (ESD) 403
 embedded systems 8–10, 162
EMR DRIVER 149–55
 portability 153
 reliability and safety 150–3
 switching on/off inductive
 loads 152–3
 zero-crossing detection 151
 enable inputs 648
 encoding data 363
 error checking/handling
 I2C protocol 502–3
 network and node errors
 547–50
 RS-232 protocol 372
SCI SCHEDULER (TICK) 561
 error code displays 137–8, 183,
 272–4
 event-triggered systems 10–11
 examples
255-TICK SCHEDULER
 896–910
 1232 external watchdog timer
 219–22
A-A FILTER 800–1
 active low resets 75–6
 amplifiers 780
 automatic lights 925–30
 baud rate generator 386–96
 brightness of light bulbs 834–9
 buzzers 116–17, 133
 CAN-based scheduler 686–710
 central-heating pump control
 with an EM relay 155
 with an SSR 158
 with **SUPER LOOP** 165–8
CERAMIC RESONATOR
 connections 66
 condition monitoring and
 control 723
 counter 424–32
 cruise-control system 874–8
CRYSTAL OSCILLATOR
 attaching to an Atmel
 89C2051 62
 attaching to a dual-processor
 board 62
 data acquisition and FFT 718
 DC motor control
MOSFET DRIVER 143
PID CONTROLLER 879–88
SSR DRIVER 147
 delays
 in an I2C library 208
 generic code 200–5
 detecting a blown bulb 805–6
 Econoresets 79
 error code displays 137–8
 in a scheduler 183
 external I2C ADC 767–72

- external parallel ADC 772–6
 external SPI ADC 763–7
HARDWARE PRM on the 8052 744–7
HARDWARE PULSE-COUNT library 731–5
HYBRID SCHEDULER 341–57
I²C PERIPHERAL
 with ADC converters 519, 767–72
 core library 503–10
 delays in an I²C library 208
 EEPROM interface 510–15
 temperature sensors 515–19, 933–40
 internal ADC 776, 788–92
KEYPAD INTERFACE 439–48
 large buzzers 133
 LCDs
 controlling an LCD 183
 and **KEYPAD INTERFACE** 484–90
 time displays on an LCD 473–84
 updating displays 321
 LEDs
 bargraph display 187–92
 buffering three LEDs with a 74HC04 123
 driving a high-power IR LED transmitter 132
 flashing with **HARDWARE DELAY** 200–5
 flashing with **ON-OFF SWITCH** 416–22
 flashing with **SOFTWARE DELAY** 208–14
 flashing with **SWITCH INTERFACE SOFTWARE** 404–9
 flashing with Timer 1 239–42
 low-current LEDs 114
 time displays on an **MX LED DISPLAY** 458–64
 light bulbs
 automatic lights 925–30
 controlling brightness of 834–9
 detecting a blown bulb 805–6
 lighting with **MOSFET DRIVER** 142–3
 Max489 transceivers 650–74
 Max810M 79
- memory
 adding more than 64 kbytes of code memory 104–7
 adding ROM and RAM
 memory 103
 external RAM and internal SRAM on the Dallas 8XC520 98–9
 external RAM and internal XRAM on the C509 98
 internal XRAM memory on C515C 93
 Philips 8XC552 internal memory 91–2
 speed of access to memory areas 92
 microphone pre-amplifier 780
 minimal Atmel 89C2051 75
 minimal Dallas circuit 79–80
menu-driven 3-LEVEL PWM 823–9
 network with Max489
 transceivers 650–74
 on-chip ADC and PWM
 hardware 812–17
ONE-TASK SCHEDULER 914–18
 open-loop DC motor control
 (**MOSFET DRIVER**) 143
 open-loop DC motor control
 (**SSR DRIVER**) 147
 output-only library 396
PC LINK (RS-232) library 374–86
PROJECT HEADER 172
 PWM smoothing filter 821
 reading 8 switch inputs in a hostile environment 412–13
 reading and writing bits 179–83
 reading and writing bytes 178–9
 reducing the component count 103–4
 rotational speed measurement 319–20
 schedulers
255-TICK SCHEDULER 896–910
 CAN-based scheduler 686–710
 core scheduler library 280–7
generic CO-OPERATIVE SCHEDULER with 16-bit timing 288–96
- HYBRID SCHEDULER** 341–57
ONE-TASK SCHEDULER 914–18
SCC SCHEDULER 686–710
SCI SCHEDULER (DATA) 595–604
SCI SCHEDULER (TICK) precise timer ticks and standard baud rates 562–3 traffic lights 563–92
SCU SCHEDULER (RS-232) 644
SOFTWARE PRM on the 8051 751–5
SOFTWARE PULSE-COUNT library 737–40
 speaker drivers 858–9
 speech playback 845–52, 856
 speech-recognition system 800–1
SPI PERIPHERAL
 core library 527–30
 SPI-based ADC 536, 763–7
 using an EEPROM 530–6
 SSRs in telecommunication applications 146–7
 temperature sensors 515–19, 933–40
 timeouts
 generating timeout-based delays 314–15
HARDWARE TIMEOUT 308–14
LOOP TIMEOUT 301–4
 traffic lights 328–31
 using **SCI SCHEDULER (DATA)** 595–604
 using **SCI SCHEDULER (TICK)** 563–62
 using **SCU SCHEDULER (RS-232)** 644
 using **SCU SCHEDULER (RS-485)** 650–74
 using **SCC SCHEDULER** 686–710
 transferring floats between microcontrollers 714–15
UART
 adding an additional UART 640–1
 scheduler library 616–39
 watchdogs
 1232 external watchdog timer 219–22
 internal watchdog timer on the Atmel 89S53 223–7

- EXTENDED 8051** 39, 46–52
 alternative solutions 50
 hardware components 48, 49
 memory 48
 performance levels 48
 pin count 48
 portability 49–50
 ports 174
 power consumption 49
 reliability and safety 49
 external code memory 100–8
 external crystal oscillator modules 54, 58–9
 external DACs 843–4
 external data memory 88
 external parallel ADC 761, 772–6
 external serial ADC 761, 763–7, 767–72
 external watchdog chips 217–18
 EZ-USB range 373
- FFT (Fourier transform) 718
 FilterLab 798
 filters
 continuous-time 799
 design packages 798
 high-frequency 786
 low-pass 795–6, 797
 op-amp 777, 797–8, 799, 858
 pulse-width modulation 819–20
 sinc filter 854
 switched-capacitor 798–9
 see also A-A (anti-aliasing) filter
- flash ADCs 787
 flash ROM 85
 flashing an LED
- CO-OPERATIVE SCHEDULER** 259–60, 269, 288–96
HARDWARE DELAY 200
ON-OFF SWITCH interface 416–22
SOFTWARE DELAY 206–7, 208–14
SOFTWARE PRM 748–50
SWITCH INTERFACE (SOFTWARE) 404–9
 timer-driven routine 239–42
 flow control in RS-232
 communication protocol 364–5
 Fortune, J. 552
 Fourier transform (FFT) 718
 Fowler, M. 15, 16
- Franklin, G.F. 873, 874
 frequency distortion 844
 frequency-domain signal representation 818–19
 Fuerst, W. 786, 800
 Full CAN 678
 full-duplex serial communication system 362
 function keys 438, 439
 function pointers 255–8, 268–70, 277
 reliability and safety 269–70
 fuses 131
 fuzzy control 874
- galvanic cells 919, 920
 Gamma, E. 22, 24
 Ganssle, J. 22
 global positioning system (GPS)
 receivers 60
 glucose sensors 873
 graphic displays 465
- Ha, R. 252
 half-duplex serial communication system 362
 ‘hanging’ applications 217, 298
- HARDWARE DELAY** 194–205, 314–15
 portability 198–9
 reliability and safety 198
 timers/counters 194–7, 198–9
- HARDWARE PRM** (pulse-rate modulation) 742–7
 alternative solutions 744
 duty cycles 742
 portability 744
 reliability and safety 744
- HARDWARE PULSE-COUNT** 728–35
 alternative solutions 730–1
 generic library 731–5
 portability 730
 reliability and safety 730
- HARDWARE PWM** (pulse-width modulation) 808–17
 buffer limitations 810
 driver limitations 810
 duty cycle 808–9
 external hardware 809
 on-chip hardware 809
 portability 811
 reliability and safety 811
- smoothing outputs 810
 switching frequency 809–10
- HARDWARE TIMEOUT** 305–15
 portability 308
 reliability and safety 308
 testing 308–14
- HARDWARE WATCHDOG** 215–27
 external watchdog chips 217–18
 1232 external timer 217, 219–22
 portability 218
 reliability and safety 218
- Hatley, D.J. 16, 17, 19
HD44780 (LCD) components 465–6, 467–72
 header files
 PORT HEADER 184–92
 PROJECT HEADER 161, 169–72
 heat sink 156–7
 high-frequency filters 786
 see also A-A (anti-aliasing) filter
- Hill, W. 110
Hitachi LCD panel controller (HD44780) 465–6, 467–72
 Horowitz, P. 110
 Huang, H-W 410
HYBRID SCHEDULER 247, 248, 332–57
 portability 341
 reliability and safety 248, 334, 338–41
 hydrofoil communication systems 853–5
 hydrophones 782
 Hyperterminal application 370
- I2C bus 494–502
ACKNOWLEDGE signal 499–500
 byte format 499
 clock signal generation/synchronization 497, 500
 code size 501
 data transfers 498–500
 device addresses 496–7
 error-checking mechanisms 502–3
 execution speed 501
 external serial (I2C) ADC 767–72
 flexibility 501
 licence fees 502
 load capacitance 495, 501

- main application areas 501
Masters and Slaves 497,
 499–500
NOT ACKNOWLEDGE signal
 499–500
 scalability 501
 serial clock (SCL) lines 495,
 499
 serial data (SDA) lines 495,
 499
 START condition 499
 STOP condition 499
 suitability of 502
 temperature sensor 933–40
I₂C libraries
 core library 503–10
 EEPROM interface 496, 503,
 510–15
LOOP TIMEOUT 303–4
 temperature sensor interface
 498, 515–19
I₂C PERIPHERAL 303–4, 491,
 493–519
 hardware features 494–502
 portability 503
 reliability and safety 502–3
IC BUFFER 118–23
 finding an IC 119–20
 logic families 120–2
 portability 122–3
 reliability and safety 122
IC DRIVER 134–8
 current sinks 134–6
 error code displays 137–8
 portability 137
 reliability and safety 136–7
IDATA memory 86, 92
 idle operating mode 36–7, 271
 indicator light circuits 9–10
 indirect addressing 82–3
 inductive AC loads 152–3
 inductive DC loads 129–31
 inductive kick 129, 152
Infineon
 analogue/digital converter 760
 C167 34
 C501 33, 34, 36
 C505C 46
 C509 98, 872
 C515C 46, 93, 386–96,
 686–710, 760, 776,
 788–92, 809, 812–17
 C517 872
 C541 373
 CAN support 678
EXTENDED 8051 46
 internal ADC 788–92
 memory options 89–90, 93,
 98–9
 on-chip ADC 812–17
 pulse-width modulation signal
 generation 809
 PWM hardware 812–17
 watchdog chips 218
 information systems (ISs) 3–5
 infra-red (IR) LEDs 132
 initialization function 262–3
 inrush currents 128, 152
 instruction registers 470
 insulin delivery systems 873
 Intel
 8048 microcontroller 30
 8052 microcontroller 30–1
 80251 microcontroller 39
 memory options 90
 intelligent data-acquisition
 541–3
 intelligent sensors 541–3
 internal ‘external’ memory 88
 interrupt inputs and **PORT I/O**
 178
 interrupts
 and CODE memory 236
 definition 10–11
 external interrupts 561
 one interrupt per
 microcontroller rule 263
 priority levels 12–13
SCC SCHEDULER 680
SCI SCHEDULER (TICK) 555
 shared-clock schedulers
 (UART-based) 610
 timer-based 235–9
 UART-related 235
 Keil hardware simulator 275–7
KEYPAD INTERFACE 433–48
 buffer arrangements 438, 439
 code library 439–48
 function keys 438, 439
 and **LCD CHARACTER PANEL**
 484–90
 matrix of switches 434–5
 and memory 438
 portability 439
 QWERTY keypad 439
 reliability and safety 439
 scanning function 435–8
 shared-clock scheduler 439
 keypad scanning 435–8
 Knott, G. 8
 Kopetz, H. 11
 Labrosse, J.J. 252
 lamps see light bulbs
 Lander, C.W. 131
 latch combinations and clock
 frequency/speed 97, 100–1
 latching switches 401–2
 latency 613, 681–3
 Lawrenz, W. 675
LCD CHARACTER PANEL (liquid
 crystal display) 321,
 465–90
 address counter 470–1
 back lighting 472
 busy flag (BF) 470
 Character Generator (CG) RAM
 (in HD44780) 470
 character set 469
 contrast adjustment
 connection 472
 cursor/blink control circuit 470
 data bus 472
 Display Data (DD) RAM (in
 HD44780) 467
 4-bit interface 471
 HD44780 components 465–6,
 467–72
 and **KEYPAD INTERFACE**
 484–90
 memory locations 469
 on-board controller 465–6
 portability 473
 power consumption 465
 registers 470
 reliability and safety 472
 software library 472
 time displays 473–84
 updating 321
LCD (liquid crystal displays)
 graphic displays 321, 465
LED (light-emitting diodes)
 bargraph display 187–92
 buffers and connecting up
 LEDs 112
 driving multiple LEDs 118–19,
 123
 error code displays 137–8
 infra-red (IR) LEDs 132
 multi-segment LED displays
 450–1

LED (light-emitting diodes)
continued
MX LED DISPLAY 449–64
NAKED LED 110–14
and ports 453, 456
see also flashing an LED
Leen, G. 252, 539
level-shifting circuits 778–9
level-shifter IC 140
Leveson, N.G. 552
Li, Y. 723
licence fees 502, 524
light bulbs
automatic light 925–30
brightness control 823–30,
 834–9
detecting a blown bulb 805–6
pulse-width modulation 807
switching on 128, 142–3, 152
light-emitting diodes see LED
Linear Technology
switched-capacitor filters 799
watchdog chips 217
linear/non-linear control systems
 863–4
liquid crystal displays see LCD
lithium cell batteries 923–4
Liu, J.W.S. 252
load capacitance in I2C protocol
 495, 501
local networks 608
hardware and wiring 684
Locke, C.D. 251
locking mechanisms 248–50,
 338–9, 341
logic families 120–2
LONG TASK 716–19
portability 718
reliability and safety 718
loop time in **PID CONTROLLER**
 872
LOOP TIMEOUT 299–304
I2C library 303–4
portability 300
reliability and safety 300
test program for 301–3
low-pass filter 795–6, 797
Lynn, P. 786, 800

machine cycle periods see
 performance levels
Mariutti, P. 58
mask read-only memory 84
Master node 610, 611–13, 679–81

matrix arrangements of switches
 434–5
Maxim
continuous-time filters 799
LED drivers 456
Max127 ADC 767
Max150 772
Max232 365
Max270 799
Max275 799
Max541 857
Max1110 ADC 763
Max7408 799
ROBUST RESET 77, 78, 79
switch debouncing 411
switched-capacitor filters 799
voltage-mode DACs 857
watchdog chips 217
measuring speed see speed
measurement
memory 81–108
areas of memory 85–90
bank-switched arrangements
 104–7
Character Generator (CG) RAM
(in HD44780) 470
and clock frequency/speed 97,
 100–1
and co-operative scheduling
 274
Display Data (DD) RAM (in
HD44780) 467
EEPROM 85, 496, 503, 510–15,
 530–6
and **EXTENDED 8051** 48
and **KEYPAD INTERFACE** 438
locations in **LCD CHARACTER**
 PANEL 469
OFF-CHIP CODE MEMORY
 100–8
OFF-CHIP DATA MEMORY 94–9
ON-CHIP MEMORY 82–93
and **ONE-TASK SCHEDULER**
 913
in **PC LINK (RS-232)** 371
reducing requirements 894
SMALL 8051 42
speed of access 92
STANDARD 8051 34–5
types of memory 83–5
memory access and **PORT I/O**
 177
message bytes 611

message structure
SCC SCHEDULER 680–1
shared-clock schedulers
(UART-based) 610–12
Microchip
filter-design packages 798
MCP601 777
PIC 12CE673 as alternative to
SMALL 8051 44
microphone pre-amplifier 780
Microwire interface 523
MISO (Master in Slave out) data
line (SPI) 522
MISRA 12
modems 146–7
MOSFET DRIVER 139–43, 803–4
portability 142
reliability and safety 141–2
MOSI (Master out Slave in) data
line (SPI) 522
Motorola 521
MP240D3 SSR (Crydon) 158
multi-drop communication 373,
 646
multi-point communication 373,
 646
multi-segment LED displays
 450–1
MULTI-STAGE TASK 317–21
LCD library 321
portability 319
reliability and safety 319
rotational speed measurement
 319–20
temperature monitoring system
 317–19
MULTI-STATE SWITCH 397,
 423–32
incrementing a counter
 424–32
portability 424
reliability and safety 424
MULTI-STATE TASK 322–31
portability 328
reliability and safety 328
System Update task 324
traffic light system 328–31
multiplexed LED displays see
MX LED DISPLAY
multiprocessor applications
 711–24
DATA UNION 712–15
DOMINO TASK 720–4
LONG TASK 716–19

- multitasking 243–5, 247–8, 338–40
 see also multiprocessor applications; **MULTI-STAGE TASK; MULTI-STATE TASK**
- MX LED DISPLAY** 449–64
 hardware requirements 451–4
 multi-segment LED displays 450, 451, 454
 portability 456
 reliability and safety 455–6
 software code 454–5
 time displays 455–64
 updating modules 455, 456
- NAKED LED** 110–14
 connecting up LEDs 112
 portability 113
 pull-up resistors 111–12
 reliability and safety 112
- NAKED LOAD** 115–17
 portability 116
 reliability and safety 116
- National Semiconductor 523
 analogue-to-digital converters (ADC) 731
 LM12CL 858–9
 power amplifiers 858–9
- networks
 CAN (controller area networks) 545–7, 675–710
 distributed networks 608, 646, 683, 684
 local networks 608, 684
 Max489 transceivers 650–74
 node errors 547–50
 resetting networks 550
 shutting down 549
 wiring 614, 684
- Nichols, N.B. 871
- nine volt radio batteries 923
- Nise, N.S. 866, 873
- Nissanke, N. 12, 250
- nodes
 errors 547–50
 hardware 613–14
 Master node 610, 611–13, 679–81
 redundant nodes 551
 Slave node 610, 611, 613, 680–1
 wiring 684
- non-linear control systems 863–4
- normally closed (NC) switches 402
- normally open (NO) switches 402
- NOT ACKNOWLEDGE signal (I2C) 499–500
- NPN transistor switches 124–5, 127
- Nyquist frequency 784, 794
- object databases 5
- OBSERVER** 22, 23–4
- OFF-CHIP CODE MEMORY** 100–8
 bank-switched memory arrangements 104–7
 portability 102
 reliability and safety 102
- OFF-CHIP DATA MEMORY** 94–9
 portability 97–8
 reliability and safety 96–7
- on-chip DACs 842–3
- ON-CHIP MEMORY** 82–93
 areas of memory 85–90
 controlling access to 88–9
 direct addressing 82–3
 indirect addressing 82–3
 portability 91
 reliability and safety 91
 speed of access 92
 types of memory 83–5
- on-chip oscillators 60–1
- on-chip reset circuits 78, 79–80
- on-chip (voltage-mode) ADC 759–60
- ON-OFF SWITCH** 397, 414–22
 for AC loads 152–3
 for DC loads 128–31
 portability 416
 reliability and safety 416
 switch block counter 415
- ONE-SHOT ADC** (analogue-to-digital converter) 757–76
- alternative solutions 762–3
 analogue voltage measurement 757–61
 current measurement 758–61
 current-mode sensor components 761
 external parallel ADC 772–6
 external parallel (voltage-mode) ADC 761
 external serial (I2C) ADC 767–72
 external serial (SPI) ADC 763–7
- external serial (voltage-mode)
 ADC 761
- on-chip (voltage-mode) ADC 759–60
- port pins and 761
 portability 762
- potentiometers 757–8, 762–3
 and power consumption 762
 reliability and safety 762
- one-shot tasks 234, 243
- ONE-TASK SCHEDULER** 893, 911–18
 alternative solutions 913
 and CPU load 913
 load assessment 914–18
 and memory 913
 reliability and safety 913
 and timers 913
- ONE-YEAR SCHEDULER** 893, 919–30
 alternative solutions 924
 automatic light 925–30
 portability 924
 reliability and safety 924
 see also batteries
- Ong, H.L.R. 811
- open-loop control systems 861–4, 873, 876
- operating systems 5, 231–2, 944
- operational amplifiers 777, 797–8, 799, 858
- Oppenheim, A.V. 786
- oscillator cycles 33–4, 55
- oscillator drift 559
- oscillator failure 560
- oscillator frequency 55–6, 57, 274–6
 0 MHz 56
 battery-powered applications 922
 choosing 61–2
- oscillator hardware
CERAMIC RESONATOR 64–6
CRYSTAL OSCILLATOR 54–63
 on-chip oscillators 60–1
 RC oscillators (relaxation oscillators) 61
RC RESET 73
- over-sampling signals 800
- OVERLAY** directive 268
- P-only controllers 867
- parallel ADC 772–6
- parallel (voltage-mode) ADC 761

Parikh, C.R. 723
 payroll systems 4
PC LINK (RS-232) 361–96
 baud rate generation 366,
 367–8, 386–96, 520
 cable connections 365–6
CRYSTAL OSCILLATOR 368–9,
 372
 error checking 372
 link library 374–86
 memory problems 371
 multi-drop communications
 373
 output-only library 396
 PC software 370
 portability 372
 reliability and safety 371–2
SCON special function register
 367
 serial port control 366–7
 software architecture 366
 transceiver chip 365
 USB (universal serial bus) ports
 373
 voltage level conversion 365
 see also RS-232 protocol
PCA82c250 (Philips) 683
PDATA memory 88, 92
 performance levels
EXTENDED 8051 48
 and oscillator frequency 55
SMALL 8051 42
STANDARD 8051 33–4
 Perier, L. 9
 periodic tasks 234, 243
 personal computers see desktop
 systems
 Philips
 8XC552 1–2, 298
 80c751 41
 87LPC764 41, 61, 922
 CAN support 679, 683
EXTENDED 8051 48
 extended memory devices 81
 memory options 90, 91–2, 103
PCA82c250 683
SMALL 8051 41, 42
 XA-family 52
PID CONTROLLER 252, 860–89
 bandwidth 871–2
 closed-loop control systems
 864–5
 control algorithms 865–72,
 876

control parameters 871
 cruise-control system 874–8
 DC motor speed control
 879–88
 fuzzy control 874
 limitations of 873
 linear/non-linear control
 systems 863–4
 loop time 872
 open-loop control systems
 861–4, 873, 876
 P-only controllers 867
 portability 873
 reliability and safety 872
 sample rate for control systems
 871–2
 tuning the controller 877–8
 windup protection 869–70
 Pierce oscillator 54
 piezoelectric buzzers 116–17
 pins see port pins
 Pirbhai, I.A. 16, 17, 19
 PNP transistor switch 124–5
 pointers see function pointers
 Pont, M.J. 16, 24, 37, 957
PORT HEADER 184–92
 portability 187
 reliability and safety 186
PORT I/O 174–83
 bitwise operators 179–83
 interrupt inputs 178
 and memory access 177
 portability 177
 reading and writing from ports
 174–6
 bits 179–83
 bytes 178–9
 reliability and safety 176–7
 reset values 176–7
 sbit variables 176
 special function registers (SFR)
 174–5
 port pins/ports
 ALE pin 91, 95
 and **DAC OUTPUT** 844
 driving DC loads 109
EXTENDED 8051 48
 header files 184–92
 and LED displays 453, 456
 and matrix of switches 434–5
 and **ONE-SHOT ADC** 761
 RESET pin 68
 SCK pin 525
 and **SEQUENTIAL ADC** 788
 serial port control 366–7
SMALL 8051 42–3, 174
STANDARD 8051 35–6
 USB (universal serial bus) ports
 373
 portability
3-LEVEL PWM 823
255-TICK SCHEDULER 895
A-A FILTER 800
ADC PRE-AMP 779
BJT DRIVER 131
CERAMIC RESONATOR 65
CO-OPERATIVE SCHEDULER 279
CRYSTAL OSCILLATOR 60
CURRENT SENSOR 805
DAC DRIVER 857
DAC OUTPUT 844
DAC SMOOTH 856
DATA UNION 714
DOMINO TASK 722
EMR DRIVER 153
EXTENDED 8051 49–50
HARDWARE DELAY 198–9
HARDWARE PRM 744
HARDWARE PULSE-COUNT 730
HARDWARE PWM 811
HARDWARE TIMEOUT 308
HARDWARE WATCHDOG 218
HYBRID SCHEDULER 341
I₂C PERIPHERAL 503
IC BUFFER 122–3
IC DRIVER 137
KEYPAD INTERFACE 439
LCD CHARACTER PANEL 473
LONG TASK 718
LOOP TIMEOUT 300
MOSFET DRIVER 142
MULTI-STAGE TASK 319
MULTI-STATE SWITCH 424
MULTI-STATE TASK 328
MX LED DISPLAY 456
NAKED LED 113
NAKED LOAD 116
OFF-CHIP CODE MEMORY 102
OFF-CHIP DATA MEMORY 97–8
ON-CHIP MEMORY 91
ON-OFF SWITCH 416
ONE-SHOT ADC 762
ONE-YEAR SCHEDULER 924
PC LINK (RS-232) 372
PID CONTROLLER 873
PORT HEADER 187
PORT I/O 177
PROJECT HEADER 171–2

- PWM SMOOTHING** 820
RC RESET 74
ROBUST RESET 78
SCC SCHEDULER 686
SCI SCHEDULER (DATA) 594
SCI SCHEDULER (TICK) 561
SCU SCHEDULER (LOCAL) 615
SCU SCHEDULER (RS-232) 643
SCU SCHEDULER (RS-485) 649
SEQUENTIAL ADC 788
SMALL 8051 44
SOFTWARE DELAY 207
SOFTWARE PRM 751
SOFTWARE PULSE-COUNT 736
SOFTWARE PWM 833
SPI PERIPHERAL 525
SSR DRIVER (AC) 157
SSR DRIVER (DC) 146
STABLE SCHEDULER 933
STANDARD 8051 38
SUPER LOOP 165
SWITCH INTERFACE (HARDWARE) 412
SWITCH INTERFACE (SOFTWARE) 403
potentiometers 757–8, 762–3
power amplifiers 858–9
power consumption
 and **CO-OPERATIVE SCHEDULER** 271
 EXTENDED 8051 49
LCD CHARACTER PANEL 465
ONE-SHOT ADC 762
SEQUENTIAL ADC 788
SMALL 8051 43
STANDARD 8051 36–7
power supply design/disruption 72–3, 73–4, 77
power-down operating mode 37
pre-emptive scheduler 246–53, 338–40
 code complexity 252
 compared to **CO-OPERATIVE SCHEDULER** 250–2
pre-emptive tasks 334–5, 340–1
printf() function 371–2
programmable read-only (PROM) memory 84
PROJECT HEADER 161, 169–72
 portability 171–2
 reliability and safety 171
 typedef statements 171–2
proportional-integral-differential (PID) control see **PID CONTROLLER**
PSEN (program store enable) 95
pull-up resistors 111–12, 122, 125, 136, 141, 145, 149, 156
pulse counting 728–30
pulse stream 741
pulse-rate modulated output 748–50
pulse-rate modulation 741–55
 HARDWARE PRM 742–7
 SOFTWARE PRM 748–55
pulse-rate sensing 727–40
 HARDWARE PULSE-COUNT 728–35
 SOFTWARE PULSE-COUNT 731, 736–40
pulse-width modulation 807–39
 filters 819–20
 frequency-domain signal representation 818–19
 HARDWARE PWM 808–17
 noise removal 819
PWM SMOOTHING 818–21
SOFTWARE PWM 831–9
3-LEVEL PWM 822–30
time-domain signal representation 818
push-button double-pole, double-throw (PB-DPDT) switch 403
push-button switch 401–3
PWM (pulse-width modulation)
 filters 819–20
PWM SMOOTHING 818–21
 portability 820
 reliability and safety 820
quantized sine wave 854–5
quartz crystal oscillator see **CRYSTAL OSCILLATOR**
quiescent state 363
QWERTY keypad 439
radar control system 861–2, 865
radio battery 923
RAM (random access memory) 83–5
Rashid, M.H. 131
RC oscillator (relaxation oscillator) 61
RC RESET 68–76
 active low inputs 75–6
brownouts 73–4
and oscillation 73
portability 74
power supply design/
 disruption 72–3, 73–4, 77
reliability and safety 72
RESET buttons 71–2
reset cycle 73
values of R and C 69–71
RC snubber 153–4
RCV420 (Burr-Brown) 761
RD (data read) 95
read-write memory 83–5
reading and writing from ports 174–6
real-time system 6–8
recharging batteries 919–20, 924
rectangular-wave output 748
redundant networks/nodes 551–2
reed relay 149
refresh function 219
register
 and **LCD CHARACTER PANEL** 470
 SPI control register 525–6
 see also special function register (SFR)
relational database system 4–5
relaxation oscillator 61
relay
 electromagnetic 144, 145
 electromechanical 149, 152, 155
reed relays 149
SSR DRIVER (AC) 156–8
SSR DRIVER (DC) 144–7
reliability and safety
 function pointers 269–70
 shared-clock schedulers 550–2
3-LEVEL PWM 823
255-TICK SCHEDULER 895
A-A FILTER 799
ADC PRE-AMP 779
BJT DRIVER 128–31
CERAMIC RESONATOR 65
CO-OPERATIVE SCHEDULER 246, 276–9
CRYSTAL OSCILLATOR 58–60
CURRENT SENSOR 805
DAC DRIVER 857
DAC OUTPUT 844
DAC SMOOTHING 855
DATA UNION 713–14
DOMINO TASK 722

reliability and safety *continued*
EMR DRIVER 150–3
EXTENDED 8051 49
HARDWARE DELAY 198
HARDWARE PRM 744
HARDWARE PULSE-COUNT 730
HARDWARE PWM 811
HARDWARE TIMEOUT 308
HARDWARE WATCHDOG 218
HYBRID SCHEDULER 248, 334,
 338–41
I2C PERIPHERAL 502–3
IC BUFFER 122
IC DRIVER 136–7
KEYPAD INTERFACE 439
LCD CHARACTER PANEL 472
LONG TASK 718
LOOP TIMEOUT 300
MOSFET DRIVER 141–2
MULTI-STAGE TASK 319
MULTI-STATE SWITCH 424
MULTI-STATE TASK 328
MX LED DISPLAY 455–6
NAKED LED 112
NAKED LOAD 116
OFF-CHIP CODE MEMORY 102
OFF-CHIP DATA MEMORY 96–7
ON-CHIP MEMORY 91
ON-OFF SWITCH 416
ONE-SHOT ADC 762
ONE-TASK SCHEDULER 913
ONE-YEAR SCHEDULER 924
PC LINK (RS-232) 371–2
PC LINK (RS-232) 872
PORT HEADER 186
PORT I/O 176–7
PROJECT HEADER 171
PWM SMOOTHER 820
RC RESET 72
ROBUST RESET 78
SCC SCHEDULER 685
SCI SCHEDULER (DATA) 594
SCI SCHEDULER (TICK) 557–61
SCU SCHEDULER (LOCAL) 615
SCU SCHEDULER (RS-232) 642
SCU SCHEDULER (RS-485) 649
SEQUENTIAL ADC 788
SMALL 8051 44
SOFTWARE DELAY 207
SOFTWARE PRM 751
SOFTWARE PULSE-COUNT 736
SOFTWARE PWM 833
SPI PERIPHERAL 525
SSR DRIVER (AC) 156–7

SSR DRIVER (DC) 145
STABLE SCHEDULER 933
STANDARD 8051 37–8
SUPER LOOP 164
SWITCH INTERFACE (HARDWARE)
 411
SWITCH INTERFACE (SOFTWARE) 401–3
repeater boards (repeaters) 608
report status function 272–4
RESET buttons 71–2
reset cycle 73
reset hardware 67–80
 RC RESET 68–76
 ROBUST RESET 77–80
RESET pin 68
reset values 176–7
resistors
 current-sense resistors 802–3
 pull-up resistors 111–12, 122,
 125, 136, 141, 145, 149,
 156
resonators see **CERAMIC RESONATOR**
ROBUST RESET 77–80
 on-chip reset circuits 78, 79–80
 portability 78
 reliability and safety 78
ROM (read-only memory) 83–5
rotary encoders 319
 see also pulse-rate sensing
rotary switch interface 401–2
rotational speed measurement
 319–20, 727
HARDWARE PULSE-COUNT
 728–35
SOFTWARE PULSE-COUNT 731,
 736–40
 see also DC motor control
RS-232 protocol 362–5, 524
 asynchronous data transmission
 364, 368, 520
 baud rates 364, 520
 compared to RS-485 646–8
CRYSTAL OSCILLATOR 368–9
 definition 362–3
 encoding data 363
 error checking 372
 flow control 364–5
 quiescent state 363
 start bit 363
 stop bit 364
 voltage levels 364, 365
 see also **PC LINK (RS-232)**;
 SCU SCHEDULER (RS-232)

RS-485 protocol 608, 646–8
 see also **SCU SCHEDULER (RS-485)**
Rumbaugh, J. 16

safety see reliability and safety
safety monitoring systems 841
sample frequency
 for control systems 871–2
DAC OUTPUT 842
Nyquist criterion 784, 794
over-sampling signals 800
SEQUENTIAL ADC 783–5
sbit variables 176
SBUF 367
scalability
 of I2C protocol 501
 of **SPI PERIPHERAL** 523
scanning keypads 435–8
SCC SCHEDULER 677–710
 architecture 679
 baud rate 681
 Infineon C515c 686–710
 interrupt generation 680
 Master node 679–81
 message structure 680–1
 node wiring 684
 portability 686
 reliability and safety 685
 Slave node 680–1
 software for 685
 tick latency 681–3
 timer overflow 680
 transceivers 683
 Update function 680
Schedulers 231–53
255-TICK SCHEDULER 893,
 894–910
CO-OPERATIVE SCHEDULER
 246–53, 255–96, 716
core scheduler library 280–96
cyclic scheduling 251
definition 245
error code displays 183, 272–4
HYBRID SCHEDULER 247, 248,
 332–57
integration of scheduler and
 application 944–5
memory requirements 894
and **MX LED DISPLAY** 455–6
ONE-TASK SCHEDULER 893,
 911–18
ONE-YEAR SCHEDULER 893,
 919–30

- pre-emptive scheduler 246–53, 338–40
 pulse-rate modulated output 748–50
STABLE SCHEDULER 932–40
 temperature-compensated schedulers 931–40
 see also shared-clock schedulers
SCI SCHEDULER (DATA) 593–607
 hardware requirements 593
 portability 594
 reliability and safety 594
 traffic light control system 595–607
 Master node 595–601
 Slave node 602–7
SCI SCHEDULER (TICK) 554–92
 alternative solutions 561–2
 baud rates 562
 error handling 561
 hardware requirements 558
 interrupt generation 555
 external interrupts 561
 oscillator drift 559
 oscillator failure 560
 portability 561
 reliability and safety 557–61
 tick messages 555, 562
 traffic light control system 563–92
 Master node 582–8
 Slave node 588–92
 tick and acknowledgement messages 582–92
 update function 555
 voltage level change 556
 SCK pin 525
 SCON special function register 367
 Scott, K. 15, 16
SCU SCHEDULER (LOCAL) 609–41
 adding an additional UART 640–1
 Address bytes 611–12
 architecture 609–10
 baud rate 612–13
 Data bytes 611–12
 interrupt generation 610
 Master node 610, 611–13
 message structure 610–12
 network wiring 614
 node hardware 613–14
 portability 615
 reliability and safety 615
 Slave node 610, 611, 613
 tick latency 613
 tick rate 612–13
 timer overflow 610
 UART scheduler library 616–41
 Master software 617–30
 Slave software 630–40
 Update function 610
SCU SCHEDULER (RS-232) 642–5
 portability 643
 reliability and safety 642
SCU SCHEDULER (RS-485) 646–74
 enable inputs 648
 network with Max489
 transceivers 650–74
 Master software 650–64
 Slave software 664–74
 portability 649
 reliability and safety 649
 Selic, B. 16
 semaphore mechanisms 248, 338
 sensors
 CURRENT SENSOR 802–6
 current-mode sensor
 components 761
 glucose sensors 873
 intelligent sensors 541–3
 temperature sensors 59, 498, 515–19, 932–3, 933–40
SEQUENTIAL ADC 782–93
 bandwidth of signals 784–5
 bit rate 786
 conversion times 787
 high-frequency filters 786
 library code 788–92
 port pins and 788
 portability 788
 power consumption 788
 reliability and safety 788
 sample frequency 783–5, 794
 software architecture 786–7
 serial clock (SCL) lines 495, 499
 serial data (SDA) lines 495, 499
 serial (I2C) ADC 767–72
 serial peripheral interface see SPI
 serial port control 366–7
 serial (SPI) ADC 763–7
 serial (voltage-mode) ADC 761
 shared-clock schedulers 539–52
 backup slave 550
 CAN-based 675–710
 clock synchronization 543–5
 data transfer 545–6
KEYPAD INTERFACE 439
 modular design benefits 541–3
 network and node errors 547–50
 reliability and safety 550–2
 resetting networks 550
SCC SCHEDULER 677–710
SCI SCHEDULER (DATA) 593–607
SCI SCHEDULER (TICK) 554–92
SCU SCHEDULER (LOCAL) 609–41
SCU SCHEDULER (RS-232) 642–5
SCU SCHEDULER (RS-485) 646–74
 shutting down networks 549
 task structure 716–17
 watchdog timers 548
 Sharp, R.S. 9
 short task handling 716–19
 simplex serial communication system 362
 sinc compensation 854
 sinc filter 854
 single-pole switches 402–3
 sink drivers 134–6
 SISO (single-input single-output) systems 863, 873
 Sivasothy, S. 608
 Slave node 610, 611, 613, 680–1
 Sleep function 271
SMALL 8051 39, 41–5
 alternative solutions 44
 hardware components 42, 44
 memory 42
 performance levels 42
 pin count 42–3
 portability 44
 ports 174
 power consumption 43
 reliability and safety 44
 Smith, S.W. 854
 smoothing signals
 DAC SMOOTHER 853–6
 PWM SMOOTHER 818–21
 software application labels 3
SOFTWARE DELAY 206–14
 portability 207
 reliability and safety 207
 software design limitations 16–21

- software patterns 22–6
SOFTWARE PRM (pulse-rate modulation) 748–55
 flashing an LED 748–50
 generic code 751–5
 portability 751
 reliability and safety 751
 variable-frequency software PRM 750
SOFTWARE PULSE-COUNT 731, 736–40
 generic library 737–40
 maximum pulse rate 736
 portability 736
 reliability and safety 736
SOFTWARE PWM (pulse-width modulation) 831–9
 frequency increases 834
 portability 833
 reliability and safety 833
 software watchdogs 219
 solid-state relay see **SSR DRIVER (AC); SSR DRIVER (DC)**
 speakers 858–9
 special function registers (SFR) 174–5
 memory 87
 SCON 367
 TCON 194
 TMOD 194
 WCON 223
 spectrum analyzer 785
 speech recognition systems 784–5, 794, 800–1
 speech signals
 digitally transmitted 854
 level shifting 778
 playback using 12-bit parallel DAC 845–52, 856
 speed measurement 319–20, 727
 see also DC motor control; pulse-rate sensing
SPI PERIPHERAL 521–36
 clock polarities 522
 clock rate 522–3
 clock signal generation 522
 code size 523
 control register 525–6
 data lines 522
 data transfer 522
 execution speed 523
 flexibility 523
 history of 521
 libraries
 core library 527–30
 external EEPROM 530–6
 licence fees 524
 main application areas 523
 Microwire interface 523
 portability 525
 reliability and safety 525
 scalability 523
 SCK pin 525
 serial (SPI) ADC 763–7
 stability of 524
 square-wave output 748
SSR DRIVER (AC) 156–8
 heat sink 156–7
 portability 157
 reliability and safety 156–7
SSR DRIVER (DC) 144–7
 portability 146
 reliability and safety 145
 in telecommunications equipment 146–7
STABLE SCHEDULER 932–40
 portability 933
 reliability and safety 933
STANDARD 8051 30–4
 alternative solutions 39
 clock speeds 34
 hardware components 35, 37, 38
 idle operating mode 36–7
 linking together 50, 540–1, 543–52
 memory architecture 34–5
 oscillator cycles 33–4, 55
 performance levels 33–4
 pin count 35–6
 portability 38
 power consumption 36–7
 power-down operating mode 37
 reliability and safety 37–8
 start bit 363
 Start function 270
 static RAM (SRAM) 84
 stop bit 364
 Storey, N. 12, 551
 successive-approximation ADCs 787
SUPER LOOP 161–8, 233–5
 portability 165
 reliability and safety 164
 switch interface 412–13
 switch block counter 415
 switch interface 412–13
 debouncing 399–400, 402, 410–11
 in industrial environments 397, 410–13
 latching switches 401–2
 matrix of switches 434–5
MULTI-STATE SWITCH 397, 423–32
 push-button switches 401–3
 rotary switch interface 401–2
 single-pole switches 402–3
 see also **ON-OFF SWITCH**
SWITCH INTERFACE (HARDWARE) 397, 410–13
 portability 412
 reliability and safety 411
SWITCH INTERFACE (SOFTWARE) 397, 399–409
 electrostatic discharge (ESD) 403
 flashing an LED 404–9
 normally closed (NC) switches 402
 normally open (NO) switches 402
 out-of-range inputs 403
 portability 403
 push-button double-pole, double-throw (PB-DPDT) switch 403
 reliability and safety 401–3
 switched-capacitor filters 798–9
 switching frequency 809–10
 synchronization see clock synchronization
 synchronous communication protocol 520
 System Update task 324
 task array 261, 277
 task duration 234–5, 243–5, 252, 297, 333
 and multiprocessor systems 720–1
 task jitter 268
 task overlap 277, 278
 task structure 716–17
 consecutively scheduled tasks 720–4
 long and short task handling 716–19
 worst case execution time (WCET) 716

- task-oriented design 316–31
MULTI-STAGE TASK 317–21
MULTI-STATE TASK 322–31
- TCON special function register 194
- TDMA (time division multiple access) protocol 543, 546
- Temic CAN support 679
- temperature monitoring system 317–19, 762
- temperature sensors 59, 498, 515–19, 932–3, 933–40
- temperature-compensated crystal oscillator (TCXOs) 59, 931, 932
- temperature-compensated schedulers 931–40
- Texas Instruments, TUSB3200 373
- text displays see LCD (liquid crystal displays) character panel
- thermistors 128, 152
- 3-LEVEL PWM (pulse-width modulation) 822–30
 menu-driven example 823–30
 portability 823
 reliability and safety 823
 Timer 2 and 822
- tick intervals 263, 278–9
- tick latency 613, 681–3
- tick messages 555, 562
- tick rate 612–13
- time displays 455–64, 473–84
- time division multiple access (TDMA) protocol 543, 546
- time-domain signal representation 818
- time-triggered systems 11–13
- timeout patterns
HARDWARE TIMEOUT 305–15
LOOP TIMEOUT 299–304
- timer ISR 219
- timer overflow
SCC SCHEDULER 680
SCU SCHEDULER (LOCAL) 610
- timer-based interrupts 235–9
- timers
 auto-reload timers 197, 238–9
 and co-operative scheduling 274
 and **HARDWARE DELAY** 194–7, 198–9
 increment rates 198
- 1232 external timer 217, 219–22
- and **ONE-TASK SCHEDULER** 913
 and speed measurement systems 320
- timer 0 194–7, 198, 728–30
- timer 1 194–7, 198, 367, 728–30
- timer 2 197, 198, 238–9
 as a baud-rate generator 743
 clock-out mode 743
 and **HARDWARE PWM** 810
 and **3-LEVEL PWM** 822
 see also counters; **HARDWARE WATCHDOG**
- TMOD special function register (SFR) 194
- traffic light control system 328–31, 543–7, 563–92, 595–607
 Master node 582–8, 595–601
 Slave node 588–92, 602–7
 tick and acknowledgement messages 582–92
- transceiver chip 365
- transceivers 683
- transconductance amplifier 843–4
- transistors
 Darlington pair 858
 NPN transistor switch 124–5, 127
 PNP transistor switch 124–5
 see also **BJT DRIVER**
 (bipolar-junction transistor); **MOSFET DRIVER**
- TRIAC switch 156
- TTL logic family 120–2
- Turkish Airlines 551–2
- 255-TICK SCHEDULER** 893, 894–910
 data structure 894–5
 generic code 896–910
 portability 895
 reliability and safety 895
- typedef statements 171–2
- UART-based shared-clock schedulers
 adding an additional UART 640–1
 data transfer 608, 675
 interrupts 235
 scheduler library 616–41
- SCU SCHEDULER (LOCAL)** 609–41
- SCU SCHEDULER (RS-232)** 642–5
- SCU SCHEDULER (RS-485)** 646–74
- UDN2585A buffer 451–3, 454
- UDN2585A driver series 135–6
- ULN2803 driver series 134–5
- UML 15
- underwater pressure transducer 782
- Unified Modelling Language (UML) 15
- union (C/C++) keyword see **DATA UNION**
- Update function 263–4, 268, 336–7, 555, 610, 680
- USB (universal serial bus) ports 373
- user interfaces 359
- UV-erasable programmable read-only (UV-EPROM) 84
- variable-frequency software PRM 750
- vibration monitoring 785
- Volta, Alessandro 919
- voltage
 amplification 777–8, 780
 of batteries 922
 measurement of 757–61
 in RS-232 protocol 364, 365
 in **SCI SCHEDULER (TICK)** 556
- voltage-mode ADC 761
- voltage-mode DAC 843, 857
- Waites, N. 8
- warning devices 113
- washing machine control system 322–8
- watchdog support 274
- watchdog timers 548
- watchdogs
HARDWARE WATCHDOG 215–27
 software watchdogs 219
- waveform storage 5
- WCON special function registers (SFR) 223
- web site 982
- Wellings, A. 543, 546
- whale song 782–3
- Winbond microcontrollers 34
- WINDOW PLACE** 22, 23

windup protection 869–70
worst case execution time
(WCET) 716
WR (data write) 95
writing from ports 174–6

XA-family (Philips) 52
XDATA memory 88, 89, 92
Xilinx Foundation 51
XTR105 (Burr-Brown) 761
XTR110 (Burr-Brown) 843–4

Yalamanchili, S. 51
Yourdon, E.N. 15, 16
zero-crossing detection 151
Ziegler, J.G. 871