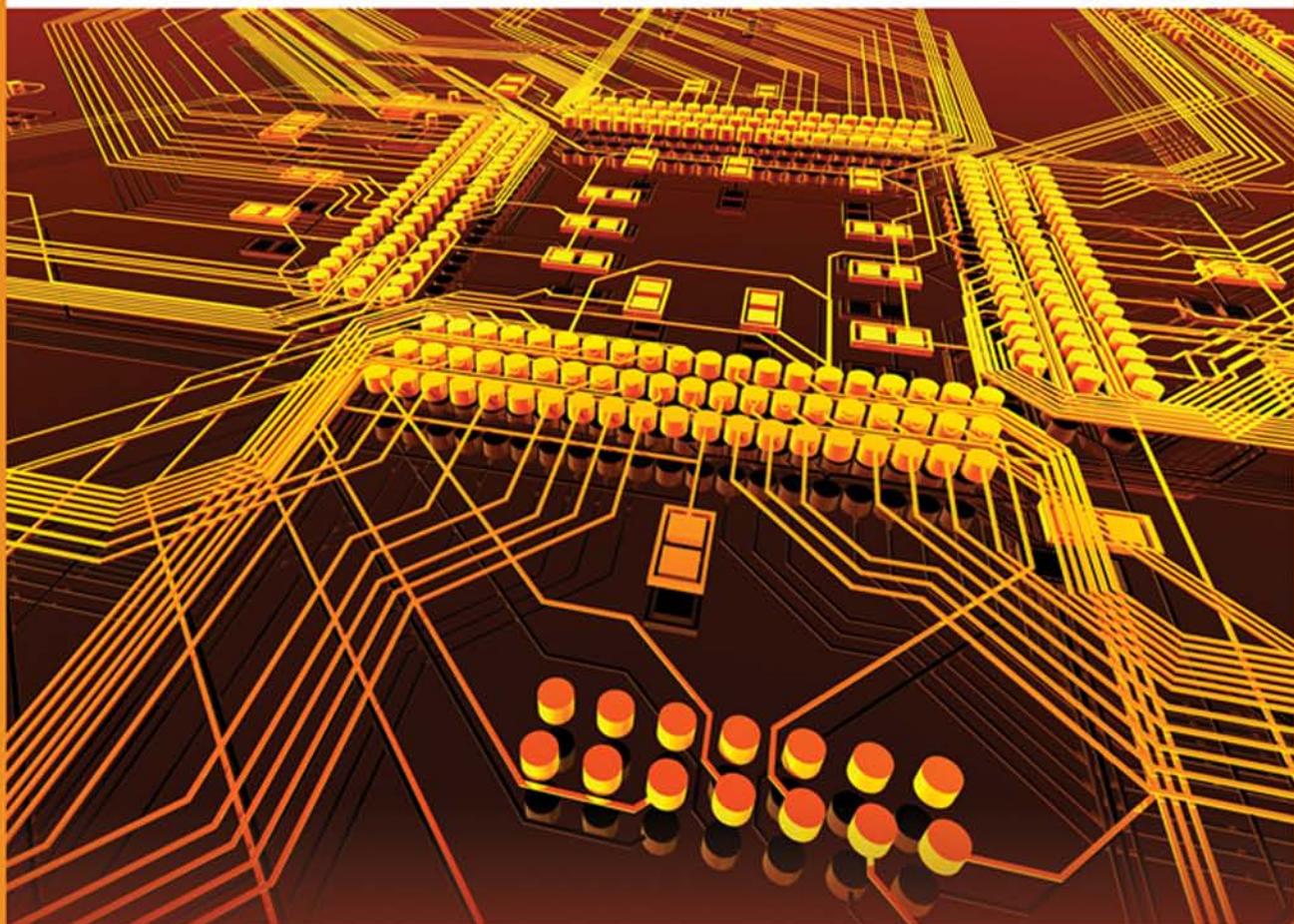


THIRD EDITION

# EMBEDDED MICROCOMPUTER SYSTEMS

## REAL TIME INTERFACING



JONATHAN W. VALVANO

### Relationship between bits, bytes and alternatives as units of precision

Binary bits	Bytes	Alternatives
8	1	256
10		1024
12		4096
16	2	65536
20		1,048,576
24	3	16,777,216
30		1,073,741,824
32	4	4,294,967,296
n	$[\lceil n/8 \rceil]$	$2^n$

### Definition of decimal digits as a unit of precision

Decimal digits	Alternatives
3	1000
$3\frac{1}{2}$	2000
$3\frac{1}{4}$	4000
4	10000
$4\frac{1}{2}$	20000
$4\frac{1}{4}$	40000
5	100000
n	$10^n$

### Data types in C

Data type	Precision	Range
unsigned char	8-bit unsigned	0 to +255
char	8-bit signed	-128 to +127
unsigned int	compiler-dependent	
int	compiler-dependent	
unsigned short	16-bit unsigned	0 to +65535
short	16-bit signed	-32768 to +32767
unsigned long	unsigned 32-bit	0 to 4294967295L
long	signed 32-bit	-2147483648L to 2147483647L
float	32-bit float	$\pm 10^{-38}$ to $\pm 10^{+38}$
double	64-bit float	$\pm 10^{-308}$ to $\pm 10^{+308}$

The 9S12C32 has nine external I/O ports

48-pin	80-pin	Shared Functions
PA0	PA7-PA0	Address/Data Bus
PB4	PB7-PB0	Address/Data Bus
PE7,4,1,0	PE7-PE0	IRQ, XIRQ, bus
—	PJ7, PJ6	Key wakeup
PM5-PM0	PM5-PM0	SPI, CAN
PP5	PP7-PP0	PWM, key wakeup
PS1-PS0	PS3-PS0	SCI
PT7-PT0	PT7-PT0	Timer, PWM
PAD7-PAD0	PAD7-PAD0	ADC, Digital I/O

### Common digital logic gates

Chip	Function
'00	2 input NAND
'02	2 input NOR
'04	Digital NOT
'05	Open collector NOT
'06	High voltage open collector NOT
'07	High voltage open collector buffer
'08	2 input AND
'10	3 input NAND
'11	3 input AND
'14	Schmitt trigger NOT
'20	4 input NAND
'21	4 input AND
'30	8 input NAND
'32	2 input OR
'125	4 tristate buffers
'138	3 to 8 line decoder
'139	2 to 4 line decoder
'148	8 to 3 line encoder
'163	4-bit counter
'164	8-bit shift register
'165	8-bit shift register
'244	8 tristate buffers
'245	8 bidirectional drivers
'374	8 D flip-flops
'573	8 transparent latches
'589	8-bit shift register
'595	8-bit shift register
'676	16-bit shift register

The 9S12DP512 has 91 I/O pins.

Pin	Shared Functions
PAD15-PAD8	ADC
PAD7-PAD0	ADC
PA7-PA0	Address/Data bus
PB7-PB0	Address/Data bus
PE7-PE0	IRQ, XIRQ, bus
PH7-PH4	SPI2, key wakeup
PH3-PH0	SPI1, key wakeup
PJ7-6, PJ1-0	CAN4, I <sup>2</sup> C, CAN0, key
PK7, PK5-PK0	Address bus
PM7-PM6	CAN3, CAN4
PM5-PM4	CAN2, CAN0, CAN4, SPI0
PM3-PM2	CAN1, CAN0, SPI0
PM1-PM0	CAN0, BDLC
PP7-PP4	PWM, SPI2, key wakeup
PP3-PP0	PWM, SPI1, key wakeup
PS7-PS4	SPI0
PS3-PS0	SCI1, SCI0
PT7-PT0	Timer

Output parameters for various open collector gates

Family	Example	V <sub>OL</sub> (V)	I <sub>OL</sub> (A)
Standard TTL	7405	0.4	0.016
Schottky TTL	74S05	0.5	0.02
Low power Schottky TTL	74LS05	0.5	0.008
High speed CMOS	74HC05	0.33	0.004
High voltage output TTL	7406	0.7	0.040
High voltage output TTL	7407	0.7	0.04
Silicon monolithic IC	75492	0.9	0.25
Silicon monolithic IC	75451–75454	0.5	0.3
Darlington switch	ULN-2074	1.4	1.25
MOSFET	IRF-540	varies	28

9S12 interrupt vectors and CodeWarrior numbers

Address	Number	Interrupt Source
\$FFFE	0	Reset
\$FFF8	3	Unimplemented Instruction Trap
\$FFF6	4	SWI
\$FFF4	5	XIRQ
\$FFF2	6	IRQ
\$FFF0	7	Real Time Interrupt, RTIF
\$FFEE	8	Timer Channel 0, C0F
\$FFEC	9	Timer Channel 1, C1F
\$FFEA	10	Timer Channel 2, C2F
\$FFE8	11	Timer Channel 3, C3F
\$FFE6	12	Timer Channel 4, C4F
\$FFE4	13	Timer Channel 5, C5F
\$FE2	14	Timer Channel 6, C6F
\$FFE0	15	Timer Channel 7, C7F
\$FFDE	16	Timer Overflow, TOF
\$FFDC	17	Pulse Acc. Overflow, PAOVF
\$FFDA	18	Pulse Acc. Input Edge, PAIF
\$FFD8	19	SPI0, SPIF or SPTEF
\$FFD6	20	SCI0, TDRE TC RDRF IDLE
\$FFD4	21	SCI1, TDRE TC RDRF IDLE
\$FFD2	22	ATD Sequence Complete, ASCIF
\$FFCE	24	Key Wakeup J, PIFJ
\$FFCC	25	Key Wakeup H, PIFH
\$FFC0	31	I2C, IAAS TCF IBAL
\$FFBE	32	SPI1, SPIF or SPTEF
\$FFBC	33	SPI2, SPIF or SPTEF
\$FFB2	38	CAN receive
\$FFB0	39	CAN transmit
\$FF8E	56	Key Wakeup P, PIFP[7:0]

Output parameters for various open emitter gates

Family	Example	V <sub>CE</sub> (V)	I <sub>CE</sub> (A)
Silicon monolithic IC	75491	0.9	0.05
Darlington switch	ULN-2074	1.4	1.25
MOSFET	IRF-540	varies	28

H-bridge drivers

Chip	Current	Comment
MC3479	0.35 A	Stepper driver
L293D	0.6 A	Dual, diodes
L293	1 A	Dual
TPIC0107	3 A	Direction, fault status
L6203	5 A	Dual

Parameters of typical transistors used by microcomputer to source or sink current

Type	NPN	PNP	Package	V <sub>be(SAT)</sub>	V <sub>ce(SAT)</sub>	h <sub>fe</sub> min/max	I <sub>c</sub>
General purpose	2N3904	2N3906	TO-92	0.85 V	0.2 V	100	10 mA
General purpose	PN2222	PN2907	TO-92	1.2 V	0.3 V	100	150 mA
General purpose	2N2222	2N2907	TO-18	1.2 V	0.3 V	100/300	150 mA
Power transistor	TIP29A	TIP30A	TO-220	1.3 V	0.7 V	15/75	1 A
Power transistor	TIP31A	TIP32A	TO-220	1.8 V	1.2 V	25/50	3 A
Power transistor	TIP41A	TIP42A	TO-220	2.0 V	1.5 V	15/75	3 A
Power Darlington	TIP120	TIP125	TO-220	2.4 V	2.0 V	1000 min	3 A

General specification of various types of resistor components<sup>1</sup>

Type	Range	Tolerance	Temperature coef	Max power
Carbon composition	1 Ω to 22 MΩ	5 to 20%	0.1%/°C	2 W
Wire-wound	1 Ω to 100 kΩ	>0.0005%	0.0005%/°C	200 W
Metal film	0.1 Ω to 10 <sup>10</sup> Ω	>0.005%	0.0001%/°C	1 W
Carbon film	10 Ω to 100 MΩ	>0.5%	0.05%/°C	2 W

<sup>1</sup> Wolf and Smith, *Student Reference Manual*, Prentice Hall, pg. 272, 1990.



# **Embedded Microcomputer Systems**

**Real Time Interfacing**  
*Third Edition*

This is an electronic version of the print textbook. Due to electronic rights restrictions, some third party content may be suppressed. Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. The publisher reserves the right to remove content from this title at any time if subsequent rights restrictions require it. For valuable information on pricing, previous editions, changes to current editions, and alternate formats, please visit [www.cengage.com/highered](http://www.cengage.com/highered) to search by ISBN#, author, title, or keyword for materials in your areas of interest.

# **Embedded Microcomputer Systems**

**Real Time Interfacing**  
*Third Edition*

**Jonathan W. Valvano**

University of Texas at Austin



---

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

**Embedded Microcomputer Systems:  
Real Time Interfacing, Third Edition**  
Jonathan W. Valvano

Publisher, Global Engineering:  
Christopher M. Shortt

Acquisitions Editor: Swati Meherishi

Assistant Development Editor:  
Yumnam Ojen Singh

Editorial Assistant: Tanya Altieri

Team Assistant: Carly Rizzo

Marketing Manager: Lauren Betsos

Media Editor: Chris Valentine

Content Project Manager: D. Jean Buttrom

Production Service: RPK Editorial Services, Inc.

Copyeditor: Shelly Gerger-Knechtl

Proofreader: Becky Taylor

Indexer: Shelly Gerger-Knechtl

Compositor: Glyph International Ltd.

Senior Art Director: Michelle Kunkler

Cover Designer: Andrew Adams

Cover Image: © Cybrain/Shutterstock

Internal Designer: Carmela Periera

Senior Rights Acquisitions Specialist:

Mardell Glinski-Schultz

Text and Image Permissions Researcher:

Kristiina Paul

First Print Buyer: Arethea L. Thomas

© 2011, 2007 Cengage Learning

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at  
**Cengage Learning Customer & Sales Support, 1-800-354-9706.**

For permission to use material from this text or product,  
submit all requests online at [www.cengage.com/permissions](http://www.cengage.com/permissions).

Further permissions questions can be e-mailed to  
[permissionrequest@cengage.com](mailto:permissionrequest@cengage.com).

Library of Congress Control Number: 2010938462

ISBN 13: 978-1-111-42625-5

ISBN-10: 1-111-42625-2

**Cengage Learning**

200 First Stamford Place, Suite 400

Stamford, CT 06902

USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:

[international.cengage.com/region](http://international.cengage.com/region).

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your course and learning solutions, visit [www.cengage.com/engineering](http://www.cengage.com/engineering).

Purchase any of our products at your local college store or at our preferred online store [www.cengagebrain.com](http://www.cengagebrain.com).

Printed in the United States of America

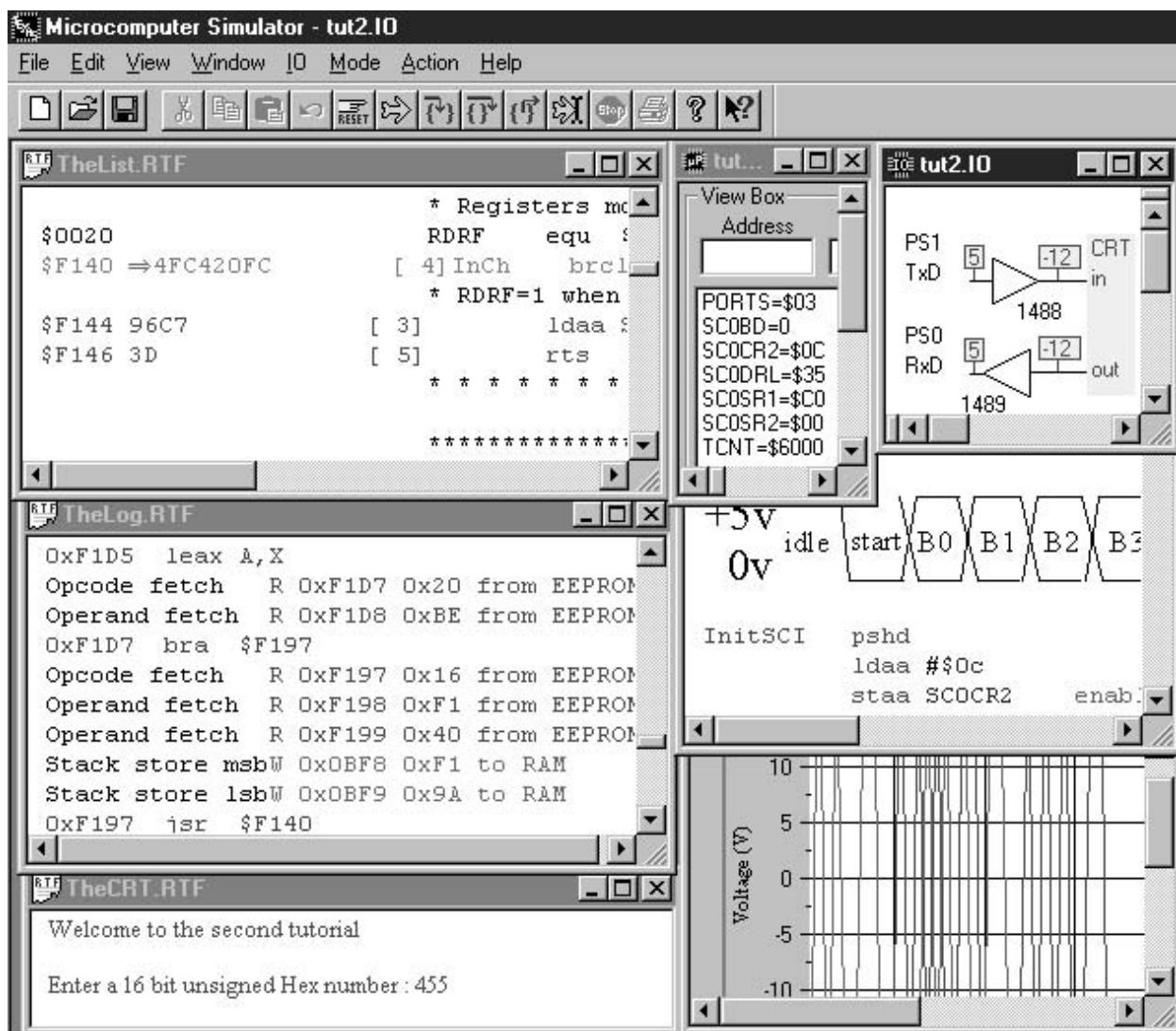
1 2 3 4 5 6 7 14 13 12 11 10

# Preface

Embedded computer systems, which are electronic systems that include a microcomputer to perform a specific dedicated application, are ubiquitous. Every week millions of tiny computer chips come pouring out of factories like Freescale, Atmel, Maxim, Texas Instruments, STMicroelectronics, Renesas, Microchip, Silicon Labs, and Mitsubishi and find their way into our everyday products. Our global economy, food production, transportation system, military defense, communication systems, and even quality of life depend on the efficiency and effectiveness of these embedded systems. As electrical and computer engineers we play a major role in all phases of this effort: planning, design, analysis, manufacturing, and marketing.

This book is unique in several ways. Like any good textbook, it strives to expose underlying concepts that can be learned today and applied later in practice. The difference lies in the details. You will find that this book is rich with detailed case studies that illustrate the basic concepts. After all, engineers do not simply develop theories but rather continue all the way to an actual device. During my years of teaching, I have found that the combination of concepts and examples is an effective method of educating student engineers. Even as a practicing engineer, I continue to study actual working examples whenever I am faced with learning new concepts.

Also unique to this book is its simulator, called *Test Execute and Simulate* (TExaS). This simulator, like all good applications, has an easy learning curve. It provides a self-contained approach to writing and testing microcomputer hardware and software. It differs from other simulators in two aspects. If enabled, the simulator shows you activity internal to the chip, like the address/data bus, the instruction register, and the effective address register. In this way the application is designed for the educational objectives of understanding how a computer works. On the other end of the spectrum, you have the ability to connect such external hardware devices as switches, keyboards, LEDs, LCDs, serial port devices, motors, and analog circuits. Logic probes, voltmeters, oscilloscopes, and logic analyzers are used to observe the external hardware. The external devices together with the microcomputer allow us to learn about embedded systems. The simulator supports many of the I/O port functions of microcomputers like interrupts, serial port, output compare, input capture, key wake up, timer overflow, and the ADC. You will find the simulator on the CD that accompanies this book. Get the CD out now, run the *Readme.htm*, install TExaS, and follow the tutorial example. In particular, double-click the **tut.uc** file in the MC9S12 subdirectory and run the tutorial on the simulator. If you are still having fun, then run the other four tutorials: **tut2.\*** shows the simple serial I/O functions, **tut3.\*** is an ADC data acquisition example, **tut4.\*** shows interrupting serial I/O functions, and **tut5.\*** is an interrupting square-wave generator. Although many programs are included in the application, these five give a broad overview of the capabilities of the simulator. The screen shot in Figure P.1 was obtained when running the **tut2.rtf** example. Notice these features in the figure: (1) address/data bus activity, (2) embedded figures in the source code, (3) external hardware, (4) voltmeters and logic probes, and (5) an oscilloscope.



**Figure P.1** The TExaS simulator can be used to design, implement, and test embedded systems.

## P.1 General Objectives

This book constitutes an in-depth treatment of the design of embedded microcomputer systems. It includes the hardware aspects of interfacing, advanced software topics like interrupts, and a systems approach to typical embedded applications. The book is unique to other microcomputer books because of its in-depth treatment of both the software and hardware issues important for real-time embedded applications. The specific objectives of the book include the understanding of:

1. Advanced architecture
  - Timing diagrams
  - Memory and I/O device interfacing to the address/data bus
2. Interfacing external devices to the computer
  - Switches and keyboards
  - LED and LCD displays

- DC motors, servos, relays, and stepper motors
  - Amplifiers, analog filters, DAC, and ADC
  - Synchronous and asynchronous serial ports
  - Microphones, speakers, and thermostors
3. Advanced programming
    - Hardware/software synchronization
    - Debugging of real time embedded systems
    - Interrupts and real time events
    - Signal generation and timing measurements
    - Threads and semaphores
  4. Embedded applications
    - Data acquisition systems with digital filters
    - Sound recording and playback
    - Linear and fuzzy logic control systems
    - Simple communication networks

---

## P.2 Changes in the 3<sup>rd</sup> Edition

Removing the 6811 material has significantly improved how students can use this book. Because the book focuses on just the 9S12, reading and using the book is very easy. In other words, the organizational flow is much smoother than the 2<sup>nd</sup> Edition. Students can read entire chapters because all parts apply to the 9S12. Conversely, students can pick and choose individual sections about specific topics. Once a student has selected a topic, the student can read the entire section, because the entire section is a complete well-contained treatment of a particular topic.

One of the strengths of this book has been its wealth of practical examples. In the 3<sup>rd</sup> Edition, material is clearly separated into fundamental concepts and example designs. In particular, each example design begins with a visual marker, followed by a concise but clear problem specification. The solution is then developed in a logical top-down manner. After the complete solution is presented and explained, another visual marker shows the reader that the example has ended and the subsequent material is fundamental again. Many new example designs are included.

There are 200 new homework exercises and 14 new labs. Exercises are divided into simple short answer and detailed design problems. Short answer questions are meant to reinforce the reading. Design questions involve design, creativity, application, and integration and are meant to be solved with paper and pencil. Lab assignments also involve design, creativity, application, and integration, but also include debugging and analysis.

There are many new topics added to the 3<sup>rd</sup> edition. The section on real time operating systems has more fundamentals and more examples. There is discussion on a design and analysis of file systems for embedded systems. There is a section on how to use the flash EEPROM for dynamic storage. The requirements document is described as an important component of the design process. MOS circuit models are used to explain interfacing concepts. There are many new sections on low power design, including regulators and battery chargers. Transmission line theory is incorporated in the discussion of serial interfaces. The section on flow charts is expanded to explain parallel and concurrent programming. There is a new section on recursion. There are new debugging techniques. For example, there are many figures of actual logic analyzer measurements to show students how the logic analyzer can be used. There is a new section on theoretical aspects of modular programming. Synchronization and interthread communication is presented in a formal way using semaphores, mailboxes, and FIFO queues. Pseudo vectors are explained, along with the impact they have on interrupt latency. There are new sections on time jitter, how to measure it, and how to reduce it. There is a new section on the physics of motor electromagnetics, and its implications on interfacing. There are new sections on brushless DC motors and servo motors.

There are sections on interfacing microphones and speakers. There are new audio labs. A PWM DAC is presented. There are both fundamental theories and practical examples of wireless communication for embedded systems. The DFT is presented and used for designing FIR digital filters.

A number of sections have been moved to provide a more logical reading flow. The LCD interface was moved from Chapter 8 to Chapter 3 because it is simple and matches educational goals of Chapter 3. Tuning equations and timing diagrams were moved from Chapter 9 into Chapter 3, because many schools skip chapter 9, but timing is an important part of interfacing. EC was moved from Chapter 14 to Chapter 7, because I2C is like SPI and used to interface the microcontroller to a peripheral. USB was moved from Chapter 14 to Chapter 7, because USB is like SCI and used to interface the microcontroller to the PC.

## P.3 Prerequisites and Supplements

This book is intended for a junior- or senior-level course in microcomputer programming and interfacing. We assume the student has knowledge of:

- MC programming functions, variables, numbers, and control structures
- Basic digital logic (multiplexers, Karnaugh maps, tristate logic)
- Data structures in C (queues, stacks, linked lists)
- Test equipment like multimeters and oscilloscopes
- Discrete analog electric circuits (resistors, capacitors, inductors, and transistors)

Although this book focuses on how to design embedded systems, extensive tutorials on the CD help students with the important issues of how to program in assembly language and how to program in C. A section later in this preface contains more information about what is on the CD.

To access all **additional course materials** for both instructors and students, please visit [www.cengagebrain.com](http://www.cengagebrain.com). At the CengageBrain.com home page, search for the ISBN of your title (from the back cover of your book) using the search box at the top of the page. This will take you to the product page where these resources can be found.

## P.4 Structure of the Book

We will use this font whenever displaying either assembly language or high-level code written in C. For the most part, software that is encapsulated in a box has equivalent examples for assembly and C. The program number in the caption will assist you in finding the software on the CD. All programs in an entire chapter are grouped together into one file. For example, the assembly and C programs in Chapter 1 can be found on the CD as files Chap1.asm and Chap1.c.

The C code in this book was written compiled and tested using the Metrowerks Code-Warrior for the 9S12. We have been very successful transporting these software examples to other compilers with only minor syntax modifications required. The major differences between the compilers are the syntax for embedded assembly and defining interrupt handlers; we have tried using various free SmallC compilers, but the software in this book is too complex to be compiled using SmallC. The biggest limitation of SmallC is its lack of support for data structures. In other words, substantial editing would be required to use SmallC in a course based on this book. Gnu C and Imagecraft ICC12 compilers are available for the 9S12.

We assume you have access to the Freescale programming reference guides for your particular microcomputer that show the details of each assembly instruction. You should also have the microcomputer technical reference manuals, which detail the I/O ports you will be using. For example, if you are using input capture to measure periods on the Freescale 9S12C32, you will find basic principles and examples in this book, but you need to access the 9S12C32 Technical Summary for a complete list of all the details. In other words, you should use this book

along with the manuals from Freescale. Although these reference manuals are available as pdf documents on the CD, it might be better to order physical documents from Freescale's literature center, or to download the latest version from the Freescale website.

Although software development is a critical aspect of embedded system design, this book is not intended to serve as an introduction to C programming. Consequently, you will find it convenient to have a C programming reference available. Fortunately, located on the CD that accompanies this book, there is a small reference called *Developing Embedded Software in C using Metrowerks* written as an HTML document. Although this reference is not as complete a programming guide as some books on C, it is specific for writing embedded software for the 9S12.

In this book we will discuss programming style, and develop debugging strategies specific to embedded real time systems from both an assembly language and a C perspective. Due to the nature of single-chip computers (they are very slow and have very little RAM when compared to today's desktop microcomputers) most of the available C compilers for the single-chip microcomputers used in this book do not support objects, or floating-point. Floating-point should be used only in situations where the range of values spans many orders of magnitude or where the range of values is unknown at software design time. Our experience is that numbers used in embedded systems usually have a narrow and known range, so integer math is sufficient. On the other hand, there is some interest for applying object-oriented approaches of C++ to embedded system design. A few illustrations of object-oriented design can be found in this book.

Chapters 1 and 2 can serve as an introduction to assembly language programming. Most embedded systems engineers agree that a working knowledge of assembly is necessary even when virtually all of our software is written in C. In specific, we believe we must know enough assembly language to be able to follow the assembly listing files generated by our compiler. This understanding of assembly language is vital when debugging, writing interrupt handlers, calculating real time events, and considering reentrancy. Consequently, detailed information on assembly language programming is included on the CD. In particular you will find microcomputer data sheets, interactive on-line help as part of the TExaS simulator, and a short introduction to assembly language programming as an HTML document.

The electrical components used in this book span a wide range. Examples include the 2764 PROM, 1N914 diode, 2N2222 transistor, 7406 open collector TTL driver, 74LS74 LSTTL flip flop, 74HC573 high speed CMOS transparent latch, Max494 op amp, L293 interface driver, Hitachi 44780 LCD driver, IRF540 MOSFET, 6N139 optocoupler, Max232 driver, and Analog Devices DAC8043 digital-to-analog converters. It is unrealistic for each student to have a personal library that contains the data books for all these devices. On the other hand it is appropriate for the company or university to establish a reference library that can be accessed by all the students. The circuit diagrams in this book usually include chip numbers and component values, but not pin numbers or circuit board layout information. Consequently, with the appropriate data sheets available, most circuits can be readily built.

## P.5 How to Teach a Course Based on This Book

The first step in the design of any course is to create a list of educational objectives, along with an understanding of the topics taught in previous classes and the topics needed as prerequisites to subsequent classes. Some important topics like computer architecture and modular software design may be included in multiple courses. Armed with this list, the instructor (or department committee) searches for a book that covers most of the topics in an effective manner. It is unrealistic to teach the entire contents of this book in a single one-semester class, but the rationale in writing this book was to cover a wide range of possible classes. There are two approaches to selecting a subset of material from this book to cover. The first approach is to pick a microcomputer and programming language. For

example, one could teach just assembly language or just C programming. In this situation, one simply skips the other cases.

The other approach to selecting the appropriate subset is to pick and choose topics. For example, a junior-level laboratory class might introduce the student to microcomputer interfacing. This class might focus on interfacing techniques and may cover Chapters 1 to 4, 6 to 8 and a little bit of Chapters 11 to 13. For these students, the remaining parts of the book become a resource to them for projects later in school or on the job. Another possibility is a senior-level project laboratory class. The objectives of this class might focus on the systems aspect of real time embedded systems. In this situation, some microcomputer programming has been previously taught, so this course might cover the advanced interfacing techniques in Chapters 5, 9, 10, and 15 and the applications in Chapters 12 to 14. For these students, the first half of the book becomes a review, allowing them to properly integrate previously learned concepts to solve complex embedded system applications.

In most departments analog circuit design (for example, op amps and analog filters) is taught in separate classes, so Chapter 11 will be a review chapter. Specific and detailed information about analog circuit design was included in this book to emphasize the system integration issues when designing embedded systems. In other words, developing embedded systems does not rely solely on the tools of computer and software engineering but rather involves all of electrical, computer, and software engineering.

The next important decision to make is the organization of the student laboratory. As engineering educators we appreciate the importance of practical “hands on” experiences in the educational process. On the other hand, space, staff, and money constraints force all of us to compromise, doing the best we can. Consequently, we present two laboratory possibilities that range considerably in cost. Indeed, you may wish to mix two or more approaches into a hybrid simulated/physical laboratory configuration. We do believe that the role of simulation is becoming increasingly important as the race for technological superiority is run with shorter and shorter design cycle times. On the other hand, we should expose our students to all the phases of engineering design, including problem specification, conceptualization, simulation, construction, and analysis.

In the first laboratory configuration, we use the traditional approach to an interfacing laboratory. A physical microcomputer development board is made available for each laboratory group of two students. There are numerous possibilities here. Companies such as Technological Arts, Axiom, Wytec, and National Instruments produce development systems. In addition to the microcomputer board, each group will need a power supply, a prototyping area to build external circuits, and the external I/O devices themselves. A number of shared development/debugging stations will also have to be configured. It is on these dedicated PC-compatible computers that the assembler or compiler is installed. If you develop software in assembly, then the TExaS simulator can be used to edit, assemble, download, and debug on a real 9S12 board. In most cases, when programming in C, a current version of a C cross-compiler is greatly preferable. As mentioned earlier, the one possibility ImageCraft’s ICC12 (ImageCraft Inc. 2625 middlefield Rd. #685, Palo Alto, CA 94306 <http://www.imagecraft.com>). The Metrowerks CodeWarrior with educational license is also an excellent choice for developing 9S12 software (Freescale, [www.freescale.com](http://www.freescale.com)). Test equipment like an oscilloscope, a digital multimeter, and a signal generator are required at each station. Expensive equipment like logic analyzers and printers can be shared. Some mail-order companies sell used or surplus electronics that can be configured into laboratory experiments (for possibilities, see my website <http://users.ece.utexas.edu/~valvano>). Many laboratory assignments are available using this traditional configuration. For universities that adopt this book, you will be allowed to download these assignments in Microsoft Word format, then rewrite, print out, and distribute to your students laboratory assignments based on these example laboratory assignments. Because of the detailed and specific nature of the laboratory setup, rewriting will certainly be necessary. The 9S12C32 board from Technological Arts and the Metrowerks cross-compiler is the specific configuration presented in the

example laboratory assignments, but the assignments are appropriate for most microcomputer development boards based on the 9S12.

The second laboratory configuration is based entirely on the TExaS simulator. Each book comes with a CD that allows the student to install the application on a single computer. Students, for the most part, work off campus and come to a teaching assistant station for help or laboratory grading. In this configuration you can either develop software in assembly using the TExaS assembler or develop C programs using the free version of CodeWarrior. The simulator itself becomes the platform on which the laboratory assignments are developed and tested. The educational license of Metrowerks CodeWarrior supports code up to 32 K. Laboratory assignments are also available using the simulator. Again, for universities that adopt this book, you will be allowed to download these assignments in Microsoft Word format, then rewrite, print out, and distribute to your students laboratory assignments based on these example laboratory assignments.

The exercises at the end of each chapter can be used to supplement the laboratory assignments. In actuality, these exercises, for the most part, were collected from old quizzes and final examinations. Consequently, these exercises address the fundamental educational objectives of the chapter, without the overwhelming complexity of a regular laboratory assignment.

## P6 What's on the CD?

The Readme.htm contains a few introductory tutorials about TExaS. These movies do not need to be copied to your hard drive; you can simply watch the movies from the CD itself.

TExaS is a complete editor, assembler, and simulator for the 9S12 microcomputer. It simulates external hardware, I/O ports, interrupts, memory, and program execution. It is intended as a learning tool for embedded systems. This software is not freeware, but the purchase of the book entitles the owner to install one copy of the program. The **Texas** directory contains the installer. You must install the software before using the application. Performing a typical installation, TExaS creates the following three subdirectories:

- MC9S12      containing 9S12C32 assembly examples
- MC9S12DP      containing 9S12DP512 assembly examples
- 9S12C32      containing 9S12C32 CodeWarrior C examples

The Custom installation allows you to pick and choose among many choices. The subdirectory ICC11 also contains the ImageCraft freeware compiler. You can run the existing ICC12 examples, but to edit and recompile you will need the commercial C compiler. The Metrowerks CodeWarrior or the Gnu GCC12 require separate installations.

The **PDF** directory contains many data sheets in Adobe pdf format. This information does not need to be copied to your hard drive; you can simply read the data sheets from the CD itself. In particular there are data sheets for microcomputers, digital logic, memory chips, op amps, ADCs, DACs, timer chips, and interface chips.

The **example** directory contains software from the book. For example, all the assembly language programs from Chapter 1 can be found in the file "Chap1.asm." Similarly, all the C language programs from Chapter 1 can be found in the file "Chap1.c."

The **assembly** directory contains an HTML document describing how to program in assembly for embedded systems using the TExaS application. This document does not need to be copied to your hard drive; you can simply read the HTML document from the CD itself. (Note also that the TExaS application itself contains a lot of information about assembly language development as part of its on-line help).

The **embed** directory contains an HTML document describing how to program in C for embedded systems. This document does not need to be copied to your hard drive; you can simply read the HTML document from the CD itself.

The **lab** directory contains software that could be used in a laboratory setting.

The **Metrowerks** directory contains an educational version of the 9S12 C compiler. This limited version can be used to develop small programs that are less than 32 K bytes of object code. This application must be installed before it can be used.

## P.7 Acknowledgments

Many shared experiences contributed to the development of this book. First, I would like to acknowledge the many excellent teaching assistants I have had the pleasure of working with. Some of these hard-working, underpaid warriors include Pankaj Bishnoi, Rajeev Sethia, Adson da Rocha, Bao Hua, Raj Randeri, Santosh Jodh, Naresh Bhavaraju, Ashutosh Kulkarni, Bryan Stiles, V. Krishnamurthy, Paul Johnson, Craig Kochis, Sean Askew, George Panayi, Jeehyun Kim, Vikram Godbole, Andres Zambrano, Ann Meyer, Hyunjin Shin, Anand Rajan, Anil Kottam, Chia-ling Wei, Jignesh Shah, Icaro Santos, David Altman, Nachiket Kharalkar, Robin Tsang, Byung Geun Jun, John Porterfield, Daniel Fernandez, James Fu., Deepak Panwar, Jacob Egner, Sandy Hermawan, Usman Tariq, Sterling Wei, Seil Oh, Antonius Keddis, Lev Shuhatovich, Glen Rhodes, Geoffrey Luke, and Karthik Sankar. I dreamed of writing this book the first time I taught microcomputer interfacing on the old Motorola MC6809 in 1981. Over the intervening years my teaching assistants have contributed greatly to the contents of this book, particularly to its laboratory assignments. In a similar manner, my students have recharged my energy each semester with their enthusiasm, dedication, and quest for knowledge.

I would also like to thank the reviewers of the second edition who provided such excellent feedback including N. Alexandridis, George Washington University; David W. Capson, McMaster University; Lee D. Coraor, The Pennsylvania State University; Subra Ganesan, Oakland University; Voicu Groza, University of Ottawa; and William R. Murray, California Polytechnic State University, San Luis Obispo.

I would like to thank the reviewers of this third edition who also provided excellent feedback including John M. Acken, Oklahoma State University; William Bishop, University of Waterloo; David W. Capson, McMaster University; James M. Conrad, University of North Carolina, Charlotte; and John Seng, California Polytechnic State University, San Luis Obispo.

Next, I appreciate the patience and expertise of my fellow faculty members at the University of Texas at Austin. From a personal perspective, Dr. John Pearce has provided much needed encouragement and support throughout my career. Also, Dr. John Cogdell and Dr. Francis Bostick helped me with analog circuit design, and Dr. Baxter Womack and Dr. Robert Flake provided good information about control systems. The book and accompanying software include many finite-state machines derived from the digital logic examples explained to me by Dr. Charles Roth. I continue to appreciate the encouragement and support of Dr. G. Jack Lipovski. An outside observer might conclude that Dr. Jack and I enjoy taking opposite sides of every issue. In actuality, this friendly competition makes us organize our otherwise erratic thoughts, and in the process everything we do is the better for it.

Over the last few years, I have enjoyed teaching embedded systems with Dr. William Bard. Bill has contributed to both the excitement and substance of our laboratory based on this book. With pushing from Bill and TAs Robin, Glen, Lev, and John, we have added low power, PCB layout, systems level design, surface mount soldering, and wireless communication to our lab experience. You can see descriptions and photos of our EE345L design competition at <http://users.ece.utexas.edu/~valvano/>.

Last, I appreciate the valuable lessons of character and commitment taught to me by my grandparents and parents. Most significantly, I acknowledge the love, patience, and support of my entire family, especially my wife Barbara and my children, Ben, Dan, and Liz.

By the grace of God, I am truly the happiest man on the planet, because I am surrounded by these fine people.

Good luck!

JONATHAN W. VALVANO

# Contents

## 1 Microcomputer-Based Systems 1

<b>1.1</b>	Computer Architecture	2
<b>1.2</b>	Embedded Computer Systems	7
<b>1.3</b>	The Design Process	11
1.3.1	Top-Down Design	11
1.3.2	Bottom-Up Design	15
<b>1.4</b>	Digital Logic and Open Collector	16
<b>1.5</b>	Digital Representation of Numbers	22
1.5.1	Fundamentals	22
1.5.2	8-Bit Numbers	25
1.5.3	Character Information	26
1.5.4	16-Bit Numbers	27
1.5.5	Fixed-Point Numbers	28
<b>1.6</b>	Common Architecture of the 9S12	30
1.6.1	Registers	31
1.6.2	Terminology	32
1.6.3	Addressing Modes	33
1.6.4	Numbering Scheme Used by Freescale	36
<b>1.7</b>	9S12 Architecture Details	36
1.7.1	9S12C32 Architecture	37
1.7.2	9S12DP512 Architecture	39
1.7.3	9S12E128 Architecture	44
1.7.4	Operating Modes	45
<b>1.8</b>	Phase-Lock-Loop (PLL)	46
<b>1.9</b>	Parallel I/O Ports	47
1.9.1	Basic Concepts of Input and Output Ports	47
1.9.2	Introduction to I/O Programming and the Direction Register	49
<b>1.10</b>	Choosing a Microcontroller	55
<b>1.11</b>	Exercises	56
<b>1.12</b>	Lab Assignments	58

## 2 Design of Software Systems 60

<b>2.1</b>	Quality Programming	60
2.1.1	Quantitative Performance Measurements	61
2.1.2	Qualitative Performance Measurements	61
<b>2.2</b>	Assembly Language Programming	62
2.2.1	Introduction	62
2.2.2	Assembly Language Syntax	64
2.2.3	Memory and Register Transfer Operations	66
2.2.4	Indexed Addressing Mode	68
2.2.5	Arithmetic Operations	71
2.2.6	Extended Precision Arithmetic Instructions on the 9S12	76
2.2.7	Shift Operations	77
2.2.8	Logical Operations	79
2.2.9	Subroutines and the Stack	80
2.2.10	Branch Operations	83
2.2.11	Assembler Pseudo-ops	85
2.2.12	Memory Allocation	89
<b>2.3</b>	Self-Documenting Code	92
2.3.1	Comments	92
2.3.2	Naming Convention	95
<b>2.4</b>	Abstraction	96
2.4.1	Definitions	96
2.4.2	9S12 Timer Details	97
2.4.3	Time Delay Software Using the Built-in Timer	98
<b>2.5</b>	Modular Software Development	104
2.5.1	Variables	104
2.5.2	Modules	111
2.5.3	Dividing a Software Task into Modules	115

2.5.4	Rules for Developing Modular Software in Assembly Language	120
<b>2.6</b>	<b>Layered Software Systems</b>	<b>121</b>
<b>2.7</b>	<b>Device Drivers</b>	<b>123</b>
2.7.1	Basic Concept of Device Drivers	123
2.7.2	Design of a Serial Communications Interface (SCI) Device Driver	124
<b>2.8</b>	<b>Object-Oriented Interfacing</b>	<b>126</b>
2.8.1	Encapsulated Objects Using Standard C	126
2.8.2	Object-Oriented Interfacing Using C++	127
2.8.3	Portability Using Standard C and C++	128
<b>2.9</b>	<b>Threads</b>	<b>130</b>
2.9.1	Single-Threaded Execution	130
2.9.2	Multithreading and Reentrancy	130
<b>2.10</b>	<b>Recursion</b>	<b>132</b>
<b>2.11</b>	<b>Debugging Strategies</b>	<b>134</b>
2.11.1	Debugging Tools	135
2.11.2	Debugging Theory	136
2.11.3	Functional Debugging	138
2.11.4	Performance Debugging	140
2.11.5	Profiling	143
<b>2.12</b>	<b>Exercises</b>	<b>144</b>
<b>2.13</b>	<b>Lab Assignments</b>	<b>147</b>

## 3 Interfacing Methods 150

<b>3.1</b>	<b>Introduction</b>	<b>150</b>
3.1.1	Performance Measures	150
3.1.2	Synchronizing the Software with the State of the I/O	151
3.1.3	Variety of Available I/O Ports	154
3.1.4	Timing Equations	156
3.1.5	Timing Diagrams	158
<b>3.2</b>	<b>Key Wake-Up</b>	<b>160</b>
<b>3.3</b>	<b>Blind Cycle Counting Synchronization</b>	<b>162</b>
3.3.1	Blind Cycle Printer Interface	162
<b>3.4</b>	<b>Gadfly or Busy-Wait Synchronization</b>	<b>163</b>
3.3.2	Blind Cycle ADC Interface	163
<b>3.5</b>	<b>Parallel I/O Interface Examples</b>	<b>166</b>
<b>3.6</b>	<b>Serial Communications Interface (SCI) Device Driver</b>	<b>173</b>
3.6.1	Transmitting in Asynchronous Mode	173
3.6.2	Receiving in Asynchronous Mode	174
3.6.3	9S12 SCI Details	175
3.6.4	SCI Device Driver	176
<b>3.7</b>	<b>Parallel Port LCD Interface with the HD44780 Controller</b>	<b>178</b>
<b>3.8</b>	<b>Exercises</b>	<b>180</b>
<b>3.9</b>	<b>Lab Assignments</b>	<b>187</b>

## 4 Interrupt Synchronization 189

<b>4.1</b>	<b>What Are Interrupts?</b>	<b>190</b>
4.1.1	Interrupt Definition	190
4.1.2	Interrupt Service Routines	191
4.1.3	When to Use Interrupts	192
4.1.4	Interthread Communication	192
<b>4.2</b>	<b>Reentrancy and Critical Sections</b>	<b>199</b>
<b>4.3</b>	<b>First-In–First-Out Queue</b>	<b>206</b>
4.3.1	Introduction to FIFOs	206
4.3.2	Two-Pointer FIFO Implementation	207
4.3.3	Two-Pointer/ Counter FIFO Implementation	210
4.3.4	FIFO Dynamics	211
<b>4.4</b>	<b>General Features of Interrupts on the 9S12</b>	<b>212</b>
<b>4.5</b>	<b>Interrupt Vectors and Priority</b>	<b>215</b>
<b>4.6</b>	<b>External Interrupt Design Approach</b>	<b>217</b>
<b>4.7</b>	<b>Polled versus Vectored Interrupts</b>	<b>219</b>
<b>4.8</b>	<b>Pseudo-Interrupt Vectors</b>	<b>221</b>
<b>4.9</b>	<b>Key Wake-Up Interrupt Examples</b>	<b>222</b>
<b>4.10</b>	<b>Power System Interface Using <math>\overline{\text{XIRQ}}</math> Synchronization</b>	<b>227</b>
<b>4.11</b>	<b>Interrupt Polling Using Linked Lists</b>	<b>228</b>
<b>4.12</b>	<b>Interrupt Priority</b>	<b>230</b>
<b>4.13</b>	<b>Round-Robin Polling</b>	<b>233</b>
<b>4.14</b>	<b>Periodic Interrupts</b>	<b>234</b>
<b>4.15</b>	<b>Low Power Design</b>	<b>239</b>
<b>4.16</b>	<b>Exercises</b>	<b>241</b>
<b>4.17</b>	<b>Lab Assignments</b>	<b>249</b>

## 5 Real-Time Operating Systems 251

<b>5.1</b>	<b>Introduction</b>	<b>252</b>
<b>5.2</b>	<b>Round-Robin Scheduler</b>	<b>257</b>
<b>5.3</b>	<b>Semaphores</b>	<b>264</b>
5.3.1	Spin-Lock Semaphore Implementation	265
5.3.2	Blocking Semaphore Implementation	267
<b>5.4</b>	<b>Thread Synchronization and Communication</b>	<b>270</b>
5.4.1	Thread Synchronization or Rendezvous	270
5.4.2	Resource Sharing, Nonreentrant Code or Mutual Exclusion	271
5.4.3	Thread Communication Between Two Threads Using a Mailbox	271
5.4.4	Thread Communication Between Many Threads Using a FIFO Queue	271

<b>5.5</b>	Fixed Scheduling	272
<b>5.6</b>	OS Considerations for I/O Devices	277
5.6.1	Board Support Package	277
5.6.2	Path Expression	278
<b>5.7</b>	Exercises	280
<b>5.8</b>	Lab Assignments	286

## 6 Timing Generation and Measurements 288

<b>6.1</b>	Input Capture	288
6.1.1	Basic Principles of Input Capture	288
6.1.2	Input Capture Details	289
6.1.3	Period Measurement	293
6.1.4	Pulse-Width Measurement	298
<b>6.2</b>	Output Compare	302
6.2.1	General Concepts	302
6.2.2	Output Compare Details	304
6.2.3	Pulse-Width Modulation	307
6.2.4	Delayed Pulse Generation	309
<b>6.3</b>	Frequency Measurement	310
6.3.1	Frequency Measurement Concepts	310
<b>6.4</b>	Conversion Between Frequency and Period	312
6.4.1	Using Period Measurement to Calculate Frequency	312
6.4.2	Using Frequency Measurement to Calculate Period	313
<b>6.5</b>	Pulse Accumulator	316
6.5.1	9S12 Pulse Accumulator Details	316
6.5.2	Frequency Measurement	317
6.5.3	Pulse-Width Measurement	318
<b>6.6</b>	Pulse-Width Modulation on the MC9S12C32	318
<b>6.7</b>	Exercises	322
<b>6.8</b>	Lab Assignments	327

## 7 Serial I/O Devices 330

<b>7.1</b>	Introduction and Definitions	330
<b>7.2</b>	RS232 Specifications	337
<b>7.3</b>	RS422/USB/RS423/RS485 Balanced Differential Lines	339
7.3.1	RS422 Output Specifications	342
7.3.2	RS422 Input Specifications	342
7.3.3	RS485 Half-Duplex Channel	343
<b>7.4</b>	Other Communication Protocols	344
7.4.1	Current Loop Channel	344
7.4.2	Introduction to Modems	344
7.4.3	Optical Channel	345
7.4.4	Digital Logic Channel	345
<b>7.5</b>	Serial Communications Interface	346
7.5.1	Transmitting in Asynchronous Mode	346

7.5.2	Receiving in Asynchronous Mode	347
7.5.3	9S12 SCI Details	350
<b>7.6</b>	Synchronous Transmission and Receiving Using the SPI	357
7.6.1	SPI Fundamentals	357
7.6.2	MC9S12C32 SPI Details	360
7.6.3	9S12DP512 Module Routing Register	363
<b>7.7</b>	Inter-Integrated Circuit ( $I^2C$ ) Interface	368
7.7.1	The Fundamentals of the $I^2C$	368
7.7.2	$I^2C$ Synchronization	371
7.7.3	9S12 $I^2C$ Details	373
7.7.4	9S12 $I^2C$ Single Master Example	376
<b>7.8</b>	Logic Level Conversion	377
<b>7.9</b>	Universal Serial Bus (USB)	378
7.9.1	Introduction	378
7.9.2	Modular USB Interface	382
7.9.3	Integrated USB Interface	383
<b>7.10</b>	Exercises	384
<b>7.11</b>	Lab Assignments	388

## 8 Parallel Port Interfaces 390

<b>8.1</b>	Input Switches and Keyboards	390
8.1.1	Interfacing a Switch to the Computer	390
8.1.2	Hardware Debouncing Using a Capacitor	392
8.1.3	Software Debouncing	395
8.1.4	Basic Approaches to Interfacing Multiple Keys	401
<b>8.2</b>	Output LEDs	410
8.2.1	Single LED Interface	411
8.2.2	Seven-Segment LED Interfaces	413
<b>8.3</b>	Liquid Crystal Displays	422
8.3.1	LCD Fundamentals	422
8.3.2	Simple LCD Interface with the MC14543	423
8.3.3	Scanned LCD Interface with the MC145000, MC145001	424
<b>8.4</b>	Transistors Used for Computer-Controlled Current Switches	427
<b>8.5</b>	Computer-Controlled Relays, Solenoids, and DC Motors	429
8.5.1	Introduction to Relays	429
8.5.2	Electromagnetic Relay Basics	430
8.5.3	Reed Relays	432
8.5.4	Solenoids	432
8.5.5	Solid-State Relays	442
<b>8.6</b>	Stepper Motors	443
8.6.1	Basic Operation	446
8.6.2	Stepper Motor Hardware Interfaces	449
8.6.3	Stepper Motor Shaft Encoder	451
<b>8.7</b>	Servo Motors	452
<b>8.8</b>	Exercises	454
<b>8.9</b>	Lab Assignments	456

## 9 Memory Interfacing 458

<b>9.1</b>	Introduction	458
<b>9.2</b>	Address Decoding	461
9.2.1	Full-Address Decoding	461
9.2.2	Minimal-Cost Address Decoding	463
9.2.3	Special Cases When Address Decoding	466
<b>9.3</b>	General Memory Bus Timing	467
9.3.1	Synchronous Bus Timing	468
9.3.2	Partially Asynchronous Bus Timing	469
9.3.3	Fully Asynchronous Bus Timing	469
<b>9.4</b>	External Bus Timing	471
9.4.1	Synchronized Versus Unynchronized Signals	471
9.4.2	Freescale MC9S12C32 External Bus Timing	472
<b>9.5</b>	General Approach to Interfacing	480
9.5.1	Interfacing to a 9S12 in Expanded Narrow Mode	480
9.5.2	Interfacing to a 9S12 in Expanded Wide Mode	480
<b>9.6</b>	9S12 Paged Memory	489
<b>9.7</b>	Programming Flash EEPROM	492
<b>9.8</b>	Dynamic RAM (DRAM)	496
<b>9.9</b>	Exercises	497
<b>9.10</b>	Lab Assignments	498

## 10 High-Speed I/O Interfacing 500

<b>10.1</b>	The Need for Speed	500
<b>10.2</b>	High-Speed I/O Applications	501
10.2.1	Mass Storage	501
10.2.2	High-Speed Data Acquisition	502
10.2.3	Video Displays	503
10.2.4	High-Speed Signal Generation	503
10.2.5	Network Communications	503
<b>10.3</b>	General Approaches to High-Speed Interfaces	504
10.3.1	Hardware FIFO	504
10.3.2	Dual Port Memory	505
10.3.3	Bank-Switched Memory	505
<b>10.4</b>	Fundamental Approach to DMA	506
10.4.1	DMA Cycles	506
10.4.2	DMA Initiation	507
10.4.3	Burst Versus Cycle Steal DMA	507
10.4.4	Single-Address versus Dual-Address DMA	508
10.4.5	DMA Programming	510
<b>10.5</b>	LCD Graphics	511
10.5.1	LCD Graphics Controller	511
10.5.2	Practical LCD Graphics Interface	514
<b>10.6</b>	Secure Digital Card Interface	515

## 10.7 File System Management 519

10.7.1	Introduction	519
10.7.2	File System Allocation	519
10.7.3	Simple File System	521
10.7.4	File Allocation Table	523
10.7.5	Internal Fragmentation	524
10.7.6	External Fragmentation	524

## 10.8 Exercises 525

## 10.9 Lab Assignments 529

## 11 Analog Interfacing 531

### 11.1 Resistors and Capacitors 531

11.1.1	Resistors	531
11.1.2	Capacitors	532

### 11.2 Operational Amplifiers (Op Amps) 534

11.2.1	Op Amp Parameters	534
11.2.2	Threshold Detector	537
11.2.3	Simple Rules for Linear Op Amp Circuits	537
11.2.4	Linear Mode Op Amp Circuits	539
11.2.5	Instrumentation Amplifier	544
11.2.6	Current-to-Voltage Circuit	545
11.2.7	Voltage-to-Current Circuit	545
11.2.8	Integrator Circuit	545
11.2.9	Derivative Circuit	546
11.2.10	Voltage Comparators with Hysteresis	546
11.2.11	Analog Isolation	547

### 11.3 Analog Filters 548

11.3.1	Simple Active Filter	548
11.3.2	Butterworth Filters	549
11.3.3	Bandpass and Band-Reject Filters	550

### 11.4 Digital-to-Analog Converters 551

11.4.1	DAC Parameters	551
11.4.2	DAC Using a Summing Amplifier	553
11.4.3	Three-Bit DAC with an R-2R Ladder	554
11.4.4	Twelve-Bit DAC with a DAC8043	556
11.4.5	DAC Selection	557
11.4.6	DAC Waveform Generation	560
11.4.7	PWM DAC	563

### 11.5 Analog-to-Digital Converters 564

11.5.1	ADC Parameters	564
11.5.2	Two-Bit Flash ADC	565
11.5.3	Successive Approximation ADC	566
11.5.4	Sixteen-Bit Dual Slope ADC	567
11.5.5	Sigma Delta ADC	568
11.5.6	ADC Interface	569

### 11.6 Sample and Hold 570

### 11.7 BiFET Analog Multiplexer 571

### 11.8 ADC System 573

11.8.1	ADC Block Diagram	573
11.8.2	Power and Grounding for the ADC System	575
11.8.3	Input Protection for CMOS Analog Inputs	575

<b>11.9</b>	<b>Power</b>	<b>575</b>
11.9.1	Regulators	575
11.9.2	Low Power Design	576
11.9.3	Battery Power	578
<b>11.10</b>	<b>Multiple-Access Circular Queue</b>	<b>580</b>
<b>11.11</b>	<b>Internal ADCs</b>	<b>582</b>
11.11.1	9S12 ADC System	582
11.11.2	ADC Software	585
<b>11.12</b>	<b>Exercises</b>	<b>586</b>
<b>11.13</b>	<b>Lab Assignments</b>	<b>589</b>

## **12 Data Acquisition Systems** **591**

<b>12.1</b>	<b>Introduction</b>	<b>591</b>
12.1.1	Accuracy	593
12.1.2	Resolution	595
12.1.3	Precision	595
12.1.4	Reproducibility or Repeatability	596
<b>12.2</b>	<b>Transducers</b>	<b>596</b>
12.2.1	Static Transducer Specifications	596
12.2.2	Dynamic Transducer Specifications	600
12.2.3	Nonlinear Transducers	601
12.2.4	Position Transducers	601
12.2.5	Velocity Measurements	603
12.2.6	Force Transducers	605
12.2.7	Temperature Transducers	606
<b>12.3</b>	<b>DAS Design</b>	<b>611</b>
12.3.1	Introduction and Definitions	611
12.3.2	Using Nyquist Theory to Determine Sampling Rate	612
12.3.3	How Many Bits Does One Need for the ADC?	615
12.3.4	Specifications for the Analog Signal Processing	615
12.3.5	How Fast Must the ADC Be?	620
12.3.6	Specifications for the S/H	620
<b>12.4</b>	<b>Analysis of Noise</b>	<b>621</b>
12.4.1	Thermal Noise	621
12.4.2	Shot Noise	624
12.4.3	1/f, or Pink, Noise	624
12.4.4	Galvanic Noise	624
12.4.5	Motion Artifact	624
12.4.6	Electromagnetic Field Induction	625
12.4.7	Techniques to Measure Noise	625
12.4.8	Techniques to Reduce Noise	627
<b>12.5</b>	<b>Data Acquisition Case Studies</b>	<b>629</b>
<b>12.6</b>	<b>Exercises</b>	<b>639</b>
<b>12.7</b>	<b>Lab Assignments</b>	<b>647</b>

## **13 Microcomputer-Based Control Systems** **648**

<b>13.1</b>	<b>Introduction to Digital Control Systems</b>	<b>648</b>
<b>13.2</b>	<b>Open-Loop Control Systems</b>	<b>649</b>

<b>13.3</b>	<b>Simple Closed-Loop Control Systems</b>	<b>655</b>
<b>13.4</b>	<b>PID Controllers</b>	<b>659</b>
13.4.1	General Approach to a PID Controller	659
13.4.2	Design Process for a PID Controller	662
<b>13.5</b>	<b>Fuzzy Logic Control</b>	<b>668</b>
<b>13.6</b>	<b>Exercises</b>	<b>681</b>
<b>13.7</b>	<b>Lab Assignments</b>	<b>684</b>

## **14 Simple Networks** **686**

<b>14.1</b>	<b>Introduction</b>	<b>686</b>
<b>14.2</b>	<b>Communication Systems Based on the SCI Serial Port</b>	<b>690</b>
<b>14.3</b>	<b>Design and Implementation of a Controller Area Network (CAN)</b>	<b>692</b>
14.3.1	The Fundamentals of CAN	692
14.3.2	Details of the 9S12C32 CAN	695
14.3.3	9S12C32 CAN Device Driver	698
<b>14.4</b>	<b>Wireless Communication</b>	<b>701</b>
<b>14.5</b>	<b>Modem Communications</b>	<b>705</b>
14.5.1	FSK Modem	705
14.5.2	Phase-Encoded Modems	708
14.5.3	Quadrature Amplitude Modems	709
<b>14.6</b>	<b>Exercises</b>	<b>710</b>
<b>14.7</b>	<b>Lab Assignments</b>	<b>714</b>

## **15 Digital Filters** **716**

<b>15.1</b>	<b>Basic Principles</b>	<b>717</b>
<b>15.2</b>	<b>Simple Digital Filter Examples</b>	<b>719</b>
<b>15.3</b>	<b>Impulse Response</b>	<b>726</b>
<b>15.4</b>	<b>High-Q 60-Hz Digital Notch Filter</b>	<b>729</b>
<b>15.5</b>	<b>Effect of Time Jitter on Digital Filters</b>	<b>734</b>
<b>15.6</b>	<b>Discrete Fourier Transform</b>	<b>736</b>
<b>15.7</b>	<b>FIR Filter Design</b>	<b>736</b>
<b>15.8</b>	<b>Direct-Form Implementations</b>	<b>739</b>
<b>15.9</b>	<b>Exercises</b>	<b>740</b>
<b>15.10</b>	<b>Lab Assignments</b>	<b>742</b>

## **Appendix 1** **744**

## **Appendix 2** **766**

## **Index** **781**



# 1 Microcomputer-Based Systems

## Chapter 1 objectives are to introduce:

- ❖ Embedded systems
- ❖ Practical aspects of digital logic
- ❖ Architecture of the Freescale MC9S12 family
- ❖ Parallel port input/output operations

The overall objective of this book is to teach the design of embedded systems. It is an effective approach to learn new techniques by doing them. But, the dilemma in teaching a laboratory-based topic such as embedded systems is that there is a tremendous volume of details that first must be learned before microcomputer hardware and software systems can be designed. The approach taken in this book is to learn by doing, starting with very simple problems and building up to more complex systems later in the book.

We will begin with a short section introducing some terminology and the basic components of a computer system. In order to understand the context of our designs, we will overview the general characteristics of embedded systems. It is in these discussions that we develop a feel for the range of possible embedded applications. Because courses taught using this book typically have a lab component, we will review some practical aspects of digital logic. We then introduce the specific architectures of the Freescale MC9S12 family. For more detailed information concerning your specific microcomputer, refer to the respective Freescale manual. Data sheets for the devices we will use can be found in PDF format at <http://users.ece.utexas.edu/~valvano>. At the end of the chapter, we will discuss prototyping methods to build embedded systems and present a simple example with binary inputs and outputs.

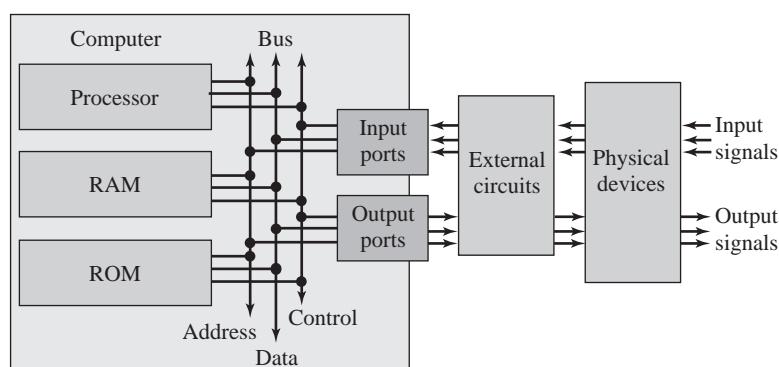
Even though we will design systems based specifically on the MC9S12C32, these solutions can, with little effort, be implemented on other versions of the MC9S12 family. If your overall goal is to develop assembly language software, then the C code can serve to clarify the software algorithms. If your overall goal is to develop C code, then I strongly advise you to learn enough assembly language so that you can understand the machine code that your compiler generates. From this understanding, you can evaluate, debug, and optimize your system.

## 1.1 Computer Architecture

A *computer system* combines a processor, random access memory (RAM), read only memory (ROM), and input/output (I/O) ports, as shown in Figure 1.1. *Software* is an ordered sequence of very specific instructions that are stored in memory, defining exactly what and when certain tasks are to be performed. The *processor* executes the software by retrieving and interpreting these instructions one at a time. A *microprocessor* is a small processor, where small refers to size (i.e., it fits in your hand) and not computational ability. For example, the Intel Pentium and the PowerPC are microprocessors. A *microcomputer* is a small computer, where again small refers to size (i.e., you can carry it) and not computational ability. For example, a desktop PC is a microcomputer. A very small microcomputer, called a *microcontroller*, contains all the components of a computer (processor, memory, I/O) on a single chip. The Freescale MC9S12C32 used in this book is a microcontroller. Because a microcomputer is a small computer, this term can be confusing because it is used to describe a wide range of systems from an 8-bit 6811 running at 2 MHz with 512 bytes of memory to a personal computer with a state-of-the-art 64-bit processor running at multi-GHz speeds having terabytes of storage.

**Figure 1.1**

The basic components of a computer system include processor, memory, and I/O.



The computer can store information in *RAM* by writing to it, or it can retrieve previously stored data by reading from it. Most RAMs are *volatile*, meaning if power is interrupted and restored, the information in the RAM is lost. Information is programmed into *ROM* using techniques more complicated than writing to RAM. On the other hand, retrieving data from a ROM is identical to retrieving data from RAM. ROMs are *nonvolatile*, meaning if power is interrupted and restored, the information in the ROM is retained. Some ROMs are programmed at the factory and can never be changed. A Programmable ROM (*PROM*) can be erased and reprogrammed by the user, but the erase/program sequence is typically 10000 times slower than the time to write data into a RAM. Some PROMs are erased with ultraviolet light and programmed with voltages, whereas electrically erasable PROMs (*EEPROM*) are both erased and programmed with voltages. Flash ROM is a popular type of EEPROM. For most of the systems in this book, we will store instructions and constants in ROM and place variables and temporary data in RAM.

**Checkpoint 1.1:** What are the differences between a microcomputer, a microprocessor, and a microcontroller?

The external devices attached to the computer provide functionality for the system. An *input port* is hardware on the computer that allows information about the external

world to be entered into the computer. The computer also has hardware called an *output port* to send information out to the external world. An *interface* is defined as the collection of the I/O port, external electronics, physical devices, and the software, which combine to allow the computer to communicate with the external world. An example of an input interface is a switch, where the operator moves the switch, and the software can recognize the switch position. An example of an output interface is a light-emitting diode (LED), where the software can turn the light on and off, and the operator can see whether or not the light is shining. There is a wide range of possible inputs and outputs, which can exist in either digital or analog form. In general, we can classify I/O interfaces into four categories

- parallel—binary data is available simultaneously on a group of lines
- serial—binary data is available one bit at a time on a single line
- analog—data is encoded as an electrical voltage, current, or power
- time—data is encoded as a period, frequency, pulse width, or phase shift

**Checkpoint 1.2:** What are the differences between an input port and an input interface?

**Checkpoint 1.3:** List three input interfaces available on a personal computer.

**Checkpoint 1.4:** List three output interfaces available on a personal computer.

In this book, numbers that start with \$ (e.g., \$64) are specified in hexadecimal, which is base 16. In C, we start hexadecimal numbers with 0x (e.g., 0x64). Intel assembly language adds an “H” at the end to specify hexadecimal (e.g., 64H). Texas Instruments uses “h” (e.g., 64h).

In a system with *memory-mapped I/O*, as shown in Figure 1.1, the I/O ports are connected to the processor in a manner similar to memory. I/O ports are assigned addresses, and the software accesses I/O using reads and writes to the specific I/O addresses. The software inputs from an input port using the same instructions as it would if it were reading from memory. Similarly, the software outputs from an output port using the same instructions as it would if it were writing to memory. The processor, memory, and I/O are connected together by an address bus, a data bus, and a control bus. Together, these buses direct the data transfer between the various modules in the computer. A *bus* is defined as a collection of signals, which are grouped for a common purpose. For example, the *address bus* on the 9S12 is 16 signals (A15-A0), which together specify the memory address (\$0000 to \$FFFF) that is currently being accessed. The address specifies both which module (input, output, RAM or ROM) as well as which cell within the module will communicate with the processor. The *data bus* contains the information that is being transferred, which on the 9S12 is 16 bits (D15-D0), but it can transfer either 8-bit or 16-bit data. The *control bus* specifies the timing and the direction of the transfer. We call a complete data transfer a *bus cycle*. In a simple computer system, like the 9S12, only two types of transfers are allowed, as shown in Table 1.1. In this simple system, the processor always controls the address (where to access), the direction (read or write), and the control (when to access.) The MC9S12C32 has 2048 bytes of RAM located at addresses \$3800 to \$3FFF. Figure 1.2 illustrates how the

**Table 1.1**  
Simple computers generate two types of cycles.

Type	Address Driven by	Data Driven by	Transfer
Memory Read Cycle	Processor	RAM, ROM or Input	Data copied to processor
Memory Write Cycle	Processor	Processor	Data copied to Output or RAM

processor fetches the 8-bit contents of location \$3800 using a read cycle. Assume memory at address \$3800 has the value \$98. The processor first places the RAM address \$3800 on the address bus, then the processor issues a read command on the control bus. The memory will respond by placing its \$98 information on the data bus, and lastly the processor will accept the data and terminate the read command.

**Figure 1.2**

A memory read cycle copies data from RAM, ROM, or an input device into the processor.

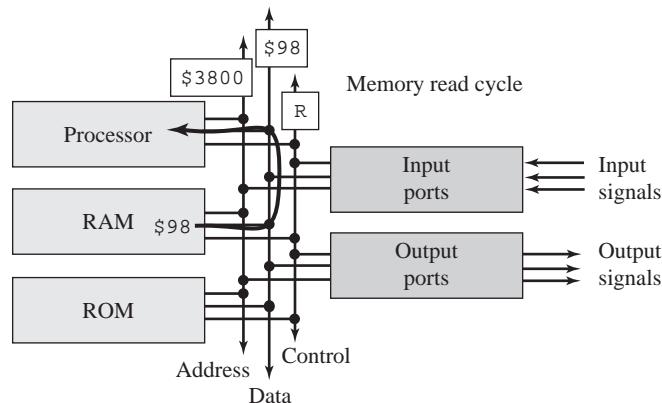
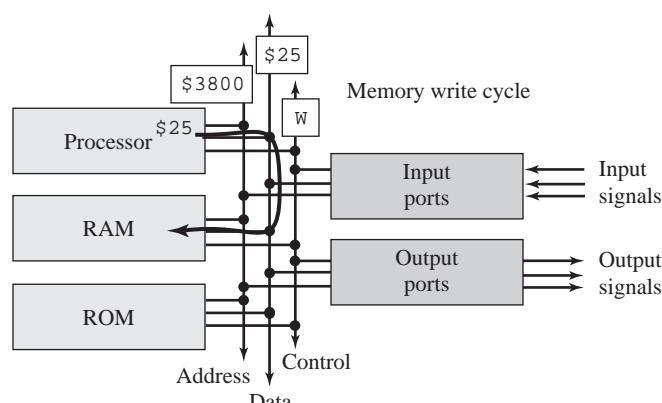


Figure 1.3 illustrates how the processor stores the 8-bit value \$25 into RAM location \$3800 using a write cycle. The processor first places the RAM address \$3800 on the address bus. Next, the processor places the \$25 information on the data bus, and then the processor issues a write command on the control bus. The memory will respond by storing the \$25 information into the proper place, and after the processor is sure the memory has captured the data, it will terminate the write command.

**Figure 1.3**

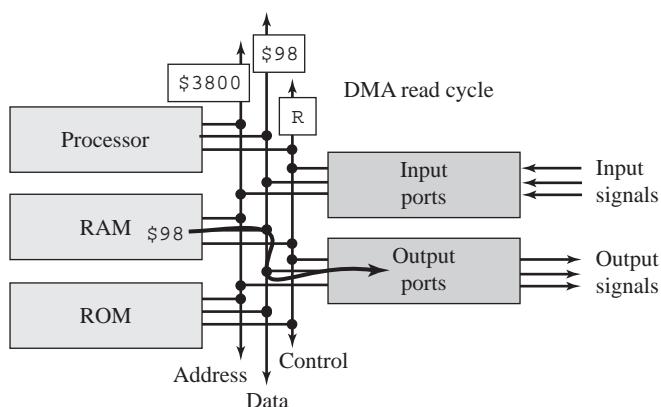
A memory write cycle copies data from the processor into RAM or an output device.



You see that if we wish to transfer data from an input device into RAM, we must first transfer it from input to the processor, then from the processor into RAM. In some microcontrollers, such as the ARM Cortex-M3 and in all desktop PCs, we can transfer data directly from input to RAM or from RAM to output using *direct memory access* (DMA). The *bandwidth* of an I/O device is the number of information bytes/sec that can be transferred. Because DMA is faster, we will use this method to interface high bandwidth devices such as disks and networks. During a read DMA cycle (Figure 1.4), data flows directly from the memory to the output device. Many systems that support DMA also allow data to be transferred from memory to memory (See Chapter 10).

**Figure 1.4**

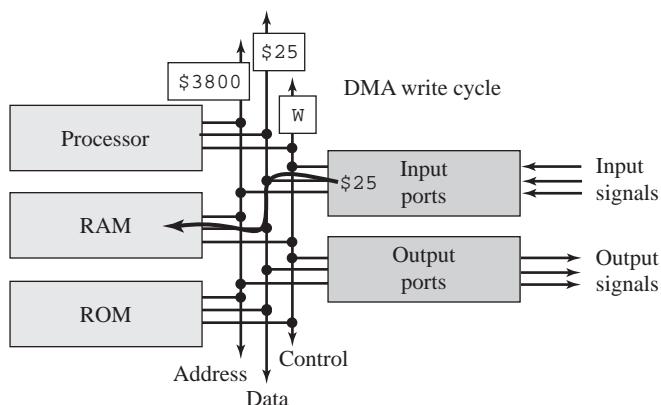
A DMA read cycle copies data from RAM, ROM, or an input device into an output device.



During a write DMA cycle (Figure 1.5) data flows directly from the input device to memory.

**Figure 1.5**

A DMA write cycle copies data from the input device into RAM or an output device.



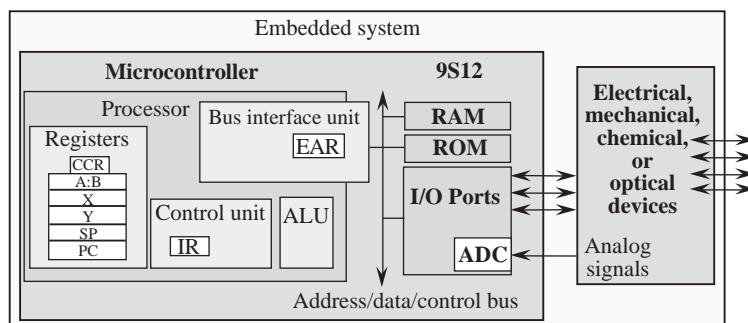
Input/output devices are important in all computers, but they are especially significant in an embedded system. In a computer system with *I/O-mapped I/O*, the control bus signals that activate the I/O are separate from those that activate the memory devices. These systems have a separate address space and separate instructions to access the I/O devices. The original Intel 8086 had four control bus signals: MEMR, MEMW, IOR, and IOW. MEMR and MEMW were used to read and write memory, while IOR and IOW were used to read and write I/O. The Intel x86 refers to any of the processors that Intel has developed based on this original architecture. Even though we do not consider the personal computer (PC) an embedded system, there are embedded systems developed on this architecture. One such platform is called the PC/104 Embedded-PC. The Intel x86 processors continue to implement this separation between memory and I/O. Currently, there are 32 to 64 memory address lines, but only 16 of those lines are used to access I/O devices. The other address lines are not used during an I/O bus cycle. Rather than use the regular memory access instructions, the Intel x86 processor uses special *in* and *out* instructions to access the I/O devices. The advantages of I/O-mapped I/O are that software can not inadvertently access I/O when it thinks it is accessing memory. In other words, it protects I/O devices from common software bugs, such as bad pointers, stack overflow, and buffer overflows. In contrast, systems with memory-mapped I/O are easier to design, and the software is easier to write.

The processor within the 9S12 microcontroller has four major components, as illustrated in Figure 1.6. The *bus interface unit* (BIU) reads data from the bus during a read cycle, and writes data onto the bus during a write cycle. The 9S12 has a single processor and

does not support DMA. Therefore, the BIU always drives the address bus and the control signals of the bus. The *effective address register* (EAR) contains the memory address used to fetch the data needed for the current instruction.

**Figure 1.6**

The four basic components of 9S12 processor.



The *control unit* (CU) orchestrates the sequence of operations in the processor. The CU issues commands to the other three components. The *instruction register* (IR) contains the *operation code* (or *opcode*) for the current instruction. Most 9S12 opcodes are 8 bits wide, but some are 16 bits. Most instructions have two parts, the opcode that defines the function to perform, and an *operand* that specifies the data to be used. In an embedded system the software is converted to machine code, which is a list of instructions, and stored in nonvolatile memory. When the system is running, instructions one at a time are fetched from memory and executed.

The *registers* are high-speed storage devices located in the processor. Registers do not have addresses like regular memory, but rather they have specific functions explicitly defined by the instruction. *Accumulators* are registers that contain data. *Index registers* contain addresses. The *program counter* (PC) points to the memory containing the instruction to execute next. In an embedded system, the PC usually points into nonvolatile memory (e.g., ROM, EPROM, or EEPROM). The information stored in nonvolatile memory (e.g., the instructions) is not lost when power is removed. The *stack pointer* (SP) points to the RAM and defines the stack. The stack is an extremely important component of software development and can be used to pass parameters, save temporary information, and implement local variables. The internal RAM of the 9S12 is volatile memory, meaning its information is lost when power is removed. On some systems, such as calculators and PDAs, a separate battery powers the RAM, creating nonvolatile RAM. The *condition code register* (CCR) contains the status of the previous operation, as well as some operating mode flags such as the interrupt enable bit. This register is called the *flag register* on the Intel computers.

The *arithmetic logic unit* (ALU) performs arithmetic and logic operations. Addition, subtraction, multiplication, and division are examples of arithmetic operations. And, or, exclusive or, and shift are examples of logical operations.

**Checkpoint 1.5:** For what do the acronyms CU DMA BIU ALU stand?

In general, the execution of an instruction goes through four phases. First, the computer fetches the machine code for the instruction by reading the value in memory pointed to by the program counter (PC). Some instructions are only one byte long, while others are two or more bytes. After each byte of the instruction is fetched, the PC is incremented. At this time, the instruction is decoded, and the effective address is determined (EAR). Many instructions require additional data, and during phase 2, the data is retrieved from memory at the effective address. Next, the actual function for this instruction is performed. Often, the computer bus is idle at this time, because no additional data is required. During the last phase, the results are written back to memory. All instructions have a phase 1, but the other three phases

may or may not occur for any specific instruction. Each of the phases may require one or more bus cycles to complete. Each bus cycle reads or writes one piece of data. The 9S12 bus cycle can transfer 8-bit or 16-bit data.

Phase	Function	R/W	Address	Comment
1	Instruction fetch	read	PC++	Put into IR,
2	Data read	read	EAR	Data passes through ALU,
3	Operation	none		ALU operations, set CCR
4	Data store	write	EAR	Results stored in memory

## 1.2 Embedded Computer Systems

An *embedded computer system* is an electronic system that includes a microcomputer such as the Freescale 9S12 that is configured to perform a specific dedicated application, drawn previously as Figure 1.6. To better understand the expression *embedded microcomputer system*, consider each word separately. In this context, the word *embedded* means “hidden inside so one can’t see it.” The software that controls the system is programmed or fixed into ROM and is not accessible to the user of the device. Even so, *software maintenance*, which is verification of proper operation, updates, fixing bugs, adding features, extending to new applications, updating end user configurations, is still extremely important. In this book, we will develop techniques that facilitate this important aspect of system design. Embedded systems have these four characteristics.

First, embedded systems typically perform a single function. Consequently, they solve only a limited range of problems. For example, the embedded system in a microwave oven may be reconfigured to control different versions of the oven within a similar product line. Still, a microwave oven will always be a microwave oven, and you can’t reprogram it to be a dishwasher. What makes each embedded system unique are the I/O ports of the microcontroller and the external devices interfaced to them.

Second, embedded systems are tightly constrained. There are typically very specific performance parameters within which the system must operate. For example, a cell-phone carrier typically gets 832 radio frequencies to use in a city, a hand-held video game must cost less than \$50, an automotive cruise control system must operate the vehicle within 3 mph of the set-point speed, and a portable MP3 player must operate for 12 hours on one battery charge.

Third, many embedded systems must operate in *real time*. In a real-time computer system, we can put an upper bound on the time required to perform the input-calculation-output sequence. A real-time system can guarantee a worst-case upper bound on the response time between when the new input information becomes available and when that information is processed. Another real-time requirement that exists in many embedded systems is the execution of periodic tasks. A periodic task is one that must be performed at equal time intervals. A real-time system can put a small and bounded limit on the interval between when a task should be run and when it is actually run. Because of the real-time nature of these systems, microcontrollers like the 9S12 have a rich set of features to handle all aspects of time.

The fourth characteristic of embedded systems is their small memory requirements. There are exceptions to this rule, such as those that process video or audio, but most have memory requirements measured in thousands of bytes.

There have been two trends in the microcontroller field. The first trend is to make microcontrollers smaller, cheaper, and with lower power. The Microchip PIC and Texas Instruments MSP430 families are good examples of this trend. Size, cost, and power are critical factors for high-volume products, where the products are often disposable. On the other end of the spectrum is the trend of larger RAM and ROM, faster processing, and increasing integration of complex I/O devices, such as Ethernet, radio, graphics, and audio.

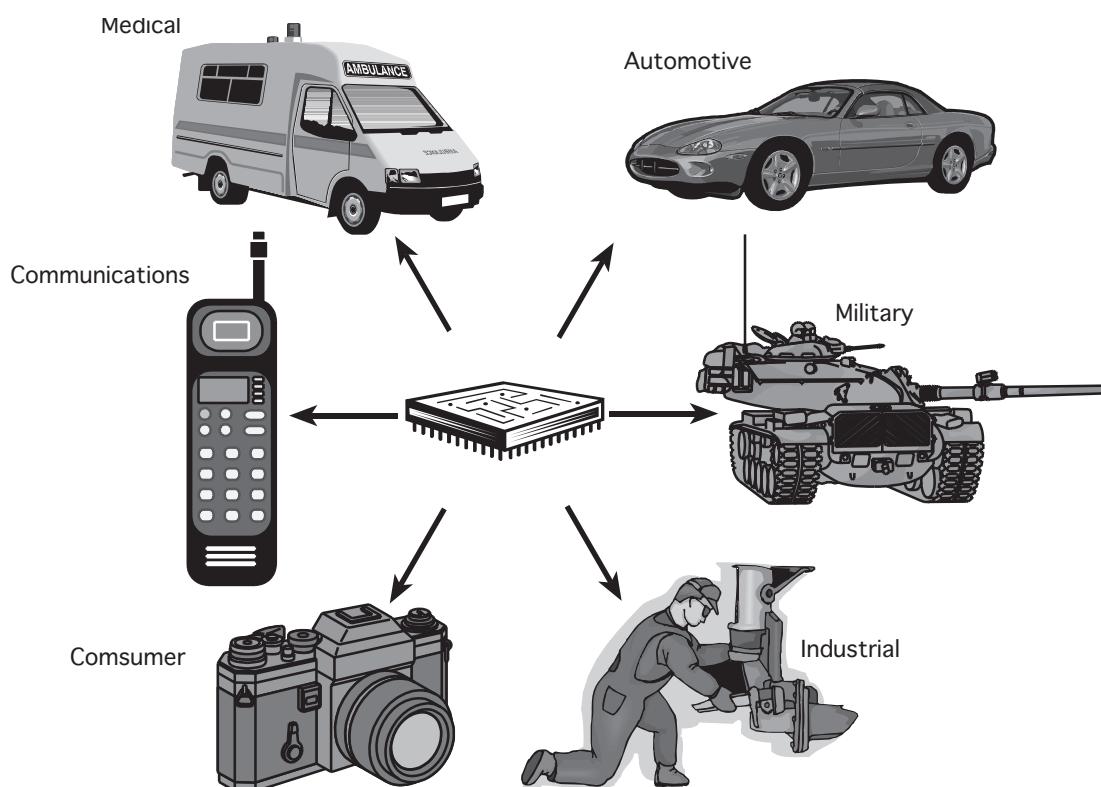
It is common for one device to have multiple microcontrollers, where the operational tasks are distributed and the microcontrollers are connected in a local area network (LAN). These high-end features are critical for consumer electronics, medical devices, automotive controllers, and military hardware, where performance and reliability are more important than cost. However, small size and low power continue as important features for all embedded systems.

**Checkpoint 1.6:** What is an embedded system?

The computer engineer has many design choices to make when building a real-time embedded system. Often, defining the problem, specifying the objectives, and identifying the constraints are harder than actual implementations. In this book, we will develop computer engineering design processes, introducing fundamental methodologies for problem specification, prototyping, testing, and performance evaluation.

In this book, we will refer to devices such as the Freescale MC9S12C32 simply as 9S12. The different versions of the microcomputers contain varying amounts of memory and input/output (I/O) devices. A typical automobile now contains an average of ten microcontrollers. In fact, upscale homes may contain as many as 150 microcontrollers, and the average consumer now interacts with microcontrollers up to 300 times a day. As shown in Figure 1.7, the general areas that employ embedded microcomputers encompass every field of engineering:

- Communications
- Automotive
- Military
- Medical
- Consumer
- Machine control



**Figure 1.7**

Example embedded computer systems.

Table 1.2 presents typical embedded microcomputer applications and the function performed by the embedded microcomputer. Each microcomputer accepts inputs, performs calculations, and generates outputs. We must also learn how to interface a wide range of inputs and outputs that can exist in either digital or analog form.

**Checkpoint 1.7:** There is a microcontroller embedded in an alarm clock. List three operations the software must perform.

In contrast, a *general-purpose computer system* typically has a keyboard, disk, and graphics display and can be programmed for a wide variety of purposes. Typical general-purpose applications include word processing, electronic mail, business accounting, scientific computing, and data base systems. General-purpose computers have the opposite of the four characteristics previously listed. First, they can perform a wide and dynamic range

**Table 1.2**  
Embedded system applications.

	<b>Function Performed by the Microcomputer</b>
<b>Consumer:</b>	
Washing machine	Controls the water and spin cycles
Exercise equipment	Measures speed, distance, calories, heart rate
Remote controls	Accepts key touches, sends infrared pulses
Clocks and watches	Maintains the time, alarm, and display
Games and toys	Entertains the user, joystick input, video output
Audio/video electronics	Interacts with the operator, enhances performance
<b>Communication:</b>	
Phone answering machines	Plays outgoing and saves incoming messages
Telephone system	Switches signals and retrieves information
Cellular phones, pagers	Interacts with key pad, microphone, and speaker
<b>Automotive:</b>	
Automatic braking	Optimizes stopping on slippery surfaces
Noise cancellation	Improves sound quality, removing noise
Theft deterrent devices	Allows keyless entry, controls alarm
Electronic ignition	Controls sparks and fuel injectors
Power windows and seats	Remembers preferred settings for each driver
Instrumentation	Collects and provides necessary information
<b>Military:</b>	
Smart weapons	Recognizes friendly targets
Missile guidance systems	Directs ordnance at the desired target
Global positioning systems	Determines where you are on the planet
Surveillance	Collects information about enemy activities
<b>Industrial:</b>	
Point-of-sale systems	Accepts inputs and manages money
Set-back thermostats	Adjusts day/night thresholds saving energy
Traffic control systems	Senses car positions, controls traffic lights
Robot systems	Inputs from sensors, controls the motors
Inventory systems	Inputs from bar code readers, prints labels
Automatic sprinklers	Controls the wetness of the soil
<b>Medical:</b>	
Infant apnea monitors	Detects breathing, alarms if stopped
Glucose monitors	Measures blood sugar levels in diabetics
Cardiac monitors	Measures heart function, alarms if problem
Drug delivery	Administers proper doses
Cancer treatments	Controls doses of radiation, drugs, or heat
Pacemakers	Sends pulses to the heart to make it beat
Prosthetic devices	Increases mobility for the handicapped

of functions. Because the general-purpose computer has a removable disk or network interface, new programs can easily be added to the system. The user of a general-purpose computer does have access to the software that controls the machine. In other words, the user decides which operating system to run and which applications to launch. Second, they are loosely constrained. For example, the Java machine used by a web browser will operate on a extremely wide range of computer platforms. Third, general-purpose machines do not run in real time. Yes, we would like the time to print a page on the printer to be fast, and we would like web page to load quickly, but there are no guaranteed response times for these types of activities. In fact, the real-time tasks that do exist (such as sound recording, burning CDs, and graphics) are actually performed by embedded systems built into the computer. Fourth, general-purpose computers employ billions, if not trillions, of memory cells.

The most common type of general-purpose computer is the personal computer (e.g., the Pentium-based IBM-PC compatible and Macintosh). Computers more powerful than the personal computer can be grouped in the workstation (\$10,000 to \$50,000 range) or the supercomputer categories (above \$50,000). See the web site [www.top500.org](http://www.top500.org) for a list of the fastest computers on the planet. These computers often employ multiple processors and have much more memory than the typical personal computer. The workstations and supercomputers are used for handling large amounts of information (business applications), running large simulations (weather forecasting), searching ([www.google.com](http://www.google.com)), or performing large calculations (scientific research). This book will not cover the general-purpose computer, although many of the basic principles of embedded computers do apply to all types of systems.

The I/O interfaces are a crucial part of an embedded system because they provide necessary functionality. Most personal computers have the same basic I/O devices (mouse, keyboard, display, CD, USB, etc.) In contrast, there is no common set of I/O that all embedded system have. The software together with the I/O ports and associated interface circuits give an embedded computer system its distinctive characteristics. A *device driver* is a set of software functions that facilitate the use of an I/O port. Another name for device driver is *application programmer interface* (API). In this book we will study a wide range of I/O ports supported by the Freescale microcomputers. Parallel ports provide for digital input and/or outputs. Serial ports include the synchronous *Serial Peripheral Interface* (SPI) and asynchronous *Serial Communications Interface* (SCI), which have a wide range of software selectable baud rates and are optimized to minimize CPU overhead. The SPI also enables synchronous communication between the microcontroller and peripheral devices such as

#### Sensors

Liquid Crystal Display (LCD) and light emitting diode (LED) displays

Analog to digital converters (ADC) and digital to analog converters (DAC)

Other microprocessors

*Analog to digital converters* convert analog voltages to digital numbers, and they are available on the 9S12 with 8-bit and 10-bit resolution. The timer features on the Freescale microcomputers include

Fixed periodic rate interrupts

Computer Operating Properly (COP) protection against software failures

Pulse accumulator for external event counting or gated time accumulation

Pulse Width Modulated outputs (PWM)

Event counter system for advanced timing operations

Input capture used for period and pulse width measurement

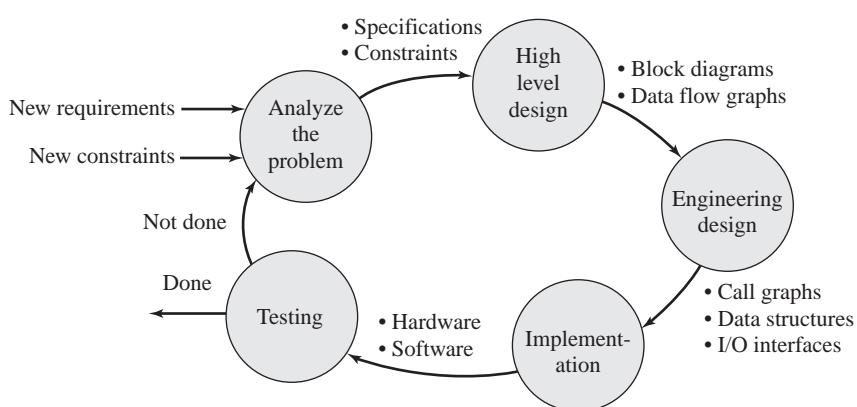
Output compare used for generating signals and frequency measurement

## 1.3 The Design Process

### 1.3.1 Top-Down Design

In this section, we introduce the design process. The process is called *top-down*, because we start with the high-level designs and work down to low-level implementations. The basic approach is introduced here, and the details of these concepts are presented throughout the remaining chapters of the book. As we learn software/hardware development tools and techniques, we can place them into the framework presented in this section. As illustrated in Figure 1.8, the development of a product follows an analysis-design-implementation-testing cycle. For complex systems with long life-spans, we traverse multiple times around the development cycle. For simple systems, a one-time pass may suffice.

**Figure 1.8**  
System development cycle.



During the **analysis phase**, we discover the requirements and constraints for our proposed system. We can hire consultants and interview potential customers in order to gather this critical information. A *requirement* is a general parameter that the system must satisfy. We begin by rewriting the system requirements, which are usually written in general form, into a list of detailed *specifications*. In general, specifications are detailed parameters describing how the system should work. For example, a requirement may state that the system should fit into a pocket, whereas a specification would give the exact size and weight of the device. For example, suppose we wish to build a motor controller. During the analysis phase, we would determine obvious specifications such as range, stability, accuracy, and response time. There may be less obvious requirements to satisfy, such as weight, size, battery life, product life, ease of operation, display readability, and reliability. Often, improving the performance of one parameter can be achieved only by decreasing the performance of another. This art of compromise defines the tradeoffs an engineer must make when designing a product. A *constraint* is a limitation, within which the system must operate. The system may be constrained as to such factors as cost, safety, compatibility with other products, use of specific electronic and mechanical parts as employed in other devices, interfaces with other instruments and test equipment, and development schedule. The following measures are often considered during the analysis phase of a project:

- Safety: The risk to humans or the environment
- Accuracy: The difference between desired and actual parameter performance
- Precision: The number of distinguishable measurements
- Resolution: The smallest change that can be reliably detected
- Response time: The time difference between triggering event and resulting action
- Bandwidth: The amount of information processed per time unit
- Maintainability: The flexibility with which the device can be modified
- Testability: The ease with which proper operation of the device can be verified

Compatibility: The conformity of the device to existing standards  
Mean time between failure: The reliability of the device  
Size and weight: The physical space required by the system  
Power: The amount of energy it takes to operate the system  
Nonrecurring engineering cost (NRE cost): The one-time cost to design and test the product  
Unit cost: The cost required to manufacture one additional product  
Time-to-prototype: The time required to design, build, and test an example system  
Time-to-market: The time required to deliver the product to the customer  
Human factors: The degree to which our customers enjoy or appreciate the product

**Checkpoint 1.8:** What's the difference between a requirement and a specification?

The following is one possible outline of a *software requirements document*. IEEE publishes a number of templates that can be used to define a project (IEEE STD 830-1998). A requirements document states what the system will do. It does not state how the system will do it. The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract. Write the document so that it is easy to read and understand by others. It should be unambiguous, complete, verifiable, and modifiable.

## 1. Overview

- 1.1. Objectives: Why are we doing this project? What is the purpose?
- 1.2. Process: How will the project be developed?
- 1.3. Roles and Responsibilities: Who will do what? Who are the clients?
- 1.4. Interactions with Existing Systems: How will it fit in?
- 1.5. Terminology: Define terms used in the document.
- 1.6. Security: How will intellectual property be managed?

## 2. Function Description

- 2.1. Functionality: What will the system do precisely?
- 2.2. Scope: List the phases and what will be delivered in each phase.
- 2.3. Prototypes: How will intermediate progress be demonstrated?
- 2.4. Performance: Define the measures and describe how they will be determined.
- 2.5. Usability: Describe the interfaces. Be quantitative if possible.
- 2.6. Safety: Explain any safety requirements and how they will be measured.

## 3. Deliverables

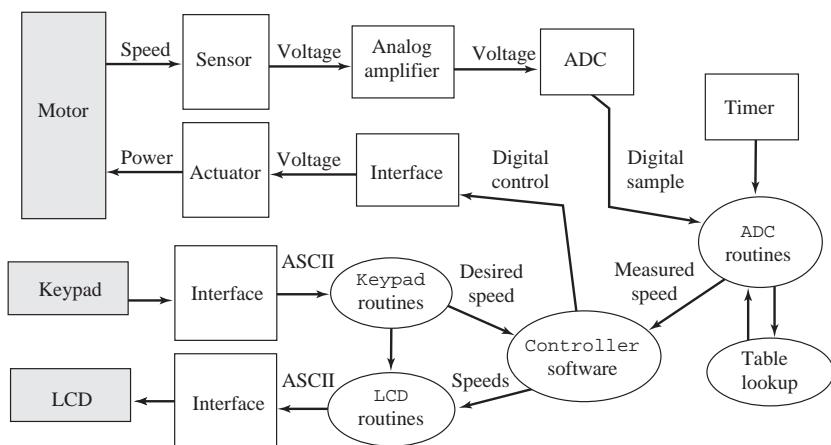
- 3.1. Reports: How will the system be described?
- 3.2. Audits: How will the clients evaluate progress?
- 3.3. Outcomes: What are the deliverables? How do we know when the system is done?

During the **high-level design** phase, we build a conceptual model of the hardware and software system. It is in this model that we employ as much abstraction as appropriate. The project is broken in modules or subcomponents. Modular design will be presented in Chapter 2. During this phase, we estimate the cost, schedule, and expected performance of the system. At this point we can decide whether the project has a high enough potential for profit. A *data flow graph* is a block diagram of the system, showing the flow of information. Arrows point from source to destination. The rectangles represent hardware components and the ovals are software modules. We use data flow graphs in the high-level design, because they describe the overall operation of the system while hiding the details of how it works. Issues such as safety (e.g., Isaac Asimov's first Law of Robotics: "A robot may not harm a human being, or, through inaction, allow a human being to come to harm") and testing (e.g., we need to verify that our system is operational) should be addressed during the high-level design.

An example data flow graph for a motor controller is shown in Figure 1.9. The requirement of the system is to deliver power to a motor so that the speed of the motor equals the desired value set by the operator using a keypad. In order to make the system easier to use and to assist in testing, a *liquid crystal display* (LCD) is added. The sensor converts motor speed into an electrical voltage. The amplifier converts this signal into the 0 to +5V voltage range required by the ADC. The ADC converts analog voltage into a digital sample. The ADC routines, using the ADC and timer hardware, collect samples and calculate voltages. Next, this software uses a table data structure to convert voltage into measured speed. The user will be able to select the desired speed using the Keypad interface. The desired and measured speed data are passed to the Controller software, which will adjust the power output in such a manner as to minimize the difference between the measured speed and the desired speed. Finally, the power commands are output to the *actuator* module. The actuator interface converts the digital control signals to power delivered to the motor. The measured speed and speed error will be sent to the LCD module. The solution to this problem will be presented in Chapter 13.

**Figure 1.9**

A data flow graph showing how signals pass through a motor controller.

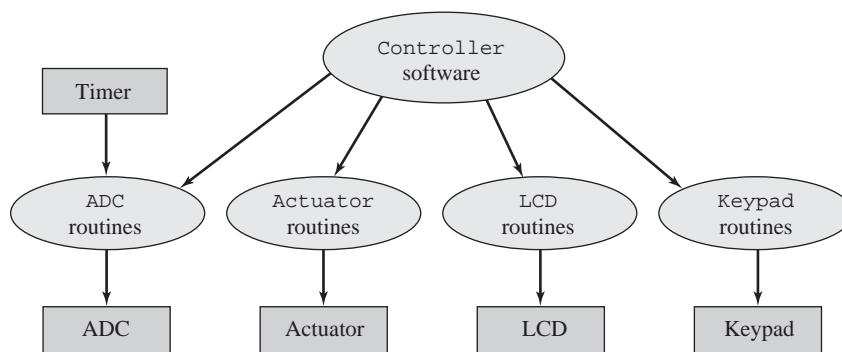


The next phase is **engineering design**. We begin by constructing a preliminary design. This system includes the overall top-down hierarchical structure, the basic I/O signals, shared data structures, and the overall software scheme. At this stage there should be a simple and direct correlation between the hardware/software systems and the conceptual model developed in the high-level design. Next, we finish the top-down hierarchical structure, and build mock-ups of the mechanical parts (connectors, chassis, cables, etc.) and user software interface. Sophisticated 3-D CAD systems can create realistic images of our system. Detailed hardware designs must include mechanical drawings. It is a good idea to have a *second source*, which is an alternative supplier that can sell us our parts if the first source can't deliver on time. *Call-graphs* are a graphical way to define how the software/hardware modules interconnect. A hierarchical system will have a tree-structured call graph. *Data structures* include both the organization of information and mechanisms to access the data. Again safety and testing should be addressed during this low-level design.

A call-graph for this motor controller is shown in Figure 1.10. Again, rectangles represent hardware components and ovals show software modules. An arrow points from the calling routine to the module it calls. The I/O ports are organized into groups and placed at the bottom of the graph. A high-level call-graph, like the one shown in Figure 1.10, shows only the high-level hardware/software modules. A detailed call-graph would include each software function and I/O port. Normally, hardware is passive and the software initiates hardware/software communication, but as we will learn in Chapter 4, it is possible for the hardware to interrupt the software and cause certain software modules to be run. In this system, the timer hardware will cause the ADC software to collect a sample at a regular rate.

**Figure 1.10**

A call flow graph for a motor controller.



The Controller software calls the Keypad routines to get the desired speed, calls the ADC software to get the motor speed at that point, determines what power to deliver to the motor, and updates the actuator by sending the power value to the Actuator interface. The Controller software calls the LCD routines to display the status of the system. As we will see in Chapters 11–15, acquiring data, calculating parameters, and outputting results at a regular rate is strategic when performing digital processing in embedded systems.

**Checkpoint 1.9:** What confusion could arise if two software modules were allowed to access the same I/O port? This situation would be evident on a call-graph if the two software modules had arrows pointing to the same I/O port.

**Observation:** If module A calls module B, and B returns data, then a data flow graph will show an arrow from B to A, but a call-graph will show an arrow from A to B.

The next phase is **implementation**. An advantage of a top-down design is that implementation of subcomponents can occur concurrently. During the initial iterations of the development cycle, it is quite efficient to implement the hardware/software using simulation. One major advantage of simulation is that it is usually quicker to implement an initial product on a simulator than to construct a physical device out of actual components. Rapid prototyping is important in the early stages of product development. This allows for more loops around the analysis-design-implementation-testing cycle, which in turn leads to a more sophisticated product.

Recent software and hardware technological developments have made significant impacts on the software development process for embedded microcomputers. The simplest approach is to use a cross-assembler or cross-compiler to convert source code into the machine code for the target system. The machine code can then be loaded into the target machine. Debugging embedded systems with this simple approach is very difficult for two reasons. First, the embedded system lacks the usual keyboard and display that assist us when we debug regular software. Second, the nature of embedded systems involves the complex and real-time interaction between the hardware and software. These real-time interactions make it impossible to test software with the usual single-stepping and print statements.

The next technological advancement that has greatly affected the manner in which embedded systems are developed is simulation. Because of the high cost and long times required to create hardware prototypes, many preliminary feasibility designs are now performed using hardware/software simulations. A simulator is a software application that models the behavior of the hardware/software system. If both the external hardware and software program are simulated together, even through the simulated time is slower than the clock on the wall, the real-time hardware/software interactions can be studied.

During the **testing** phase, we evaluate the performance of our system. First, we debug the system and validate basic functions. Next, we use careful measurements to optimize

performance, such as static efficiency (memory requirements), dynamic efficiency (execution speed), accuracy (difference between truth and measured), and stability (consistent operation). Debugging techniques are presented in Chapter 2.

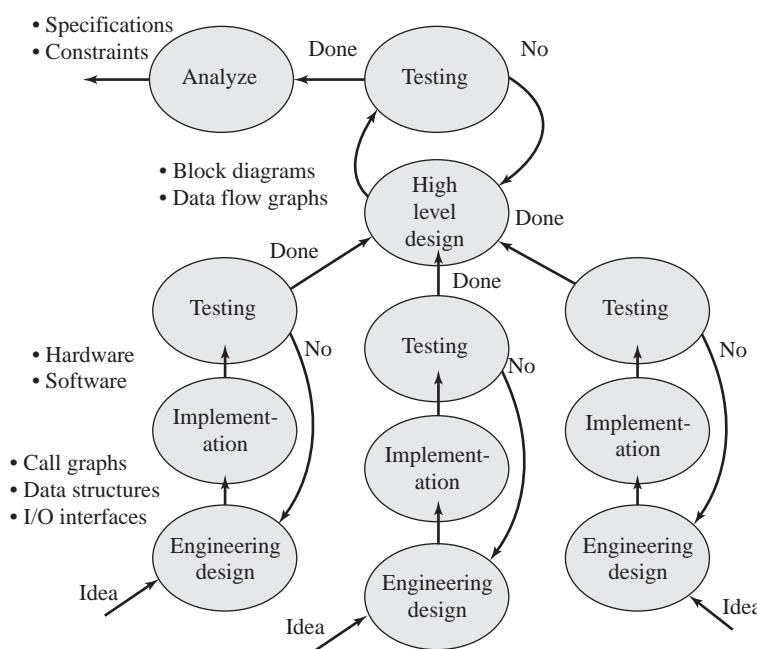
**Maintenance** is the process of correcting mistakes, adding new features, optimizing for execution speed or program size, porting to new computers or operating systems, and reconfiguring the system to solve a similar problem. No system is static. Customers may change or add requirements or constraints. To be profitable, we probably will wish to tailor each system to the individual needs of each customer. Maintenance is not really a separate phase, but rather involves additional loops around the development cycle.

### 1.3.2 Bottom-Up Design

Figure 1.8 describes top-down design as a cyclic process, beginning with a problem statement and ending up with a solution. With a *bottom-up* design we begin with solutions and build up to a problem statement. Many innovations begin with an idea, “what if . . .?” In a bottom-up design, one begins with designing, building, and testing low-level components. Figure 1.11 illustrates a two-level process, combining three subcomponents to create the overall product. This hierarchical process could have more levels and/or more components at each level. The low-level designs can occur in parallel. The design of each component is cyclic, iterating through the design-build-test cycle until the performance is acceptable. Bottom-up design may be inefficient because some subsystems may be designed, built, and tested but never used. As the design progresses the components are fit together to make the system more and more complex. Only after the system is completely built and tested does one define the overall system specifications. The bottom-up design process allows creative ideas to drive the products a company develops. It also allows one to quickly test the feasibility of an idea. If one fully understands a problem area and the scope of potential solutions, then a top-down design will arrive at an effective solution most quickly. On the other hand, if one doesn’t really understand the problem or the scope of its solutions, a bottom-up approach allows one to start off by learning about the problem.

**Observation:** A good engineer knows both bottom-up and top-down design methods, choosing the approach most appropriate for the situation at hand.

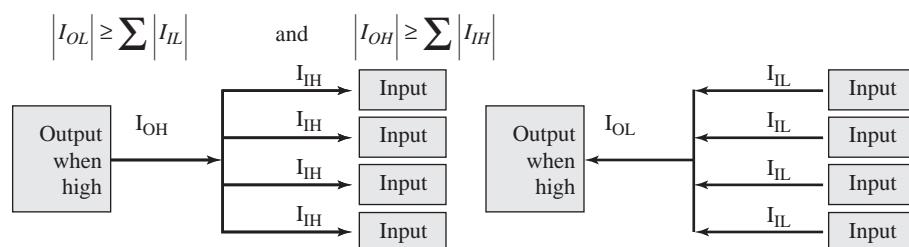
**Figure 1.11**  
System development process illustrating bottom-up design.



## 1.4 Digital Logic and Open Collector

Normal digital logic has two states: high and low. There are four currents of interest, as shown in Figure 1.12, when analyzing whether the inputs of the next stage are loading the output.  $I_{IH}$  and  $I_{IL}$  are the currents required of an input when high and low, respectively. Similarly,  $I_{OH}$  and  $I_{OL}$  are the maximum currents available at the output when high and low. In order for the output to properly drive all the inputs of the next stage, the maximum available output current must be larger than the sum of all the required input currents for both the high and low conditions.

**Figure 1.12**  
Sometimes one output  
must drive multiple  
inputs.

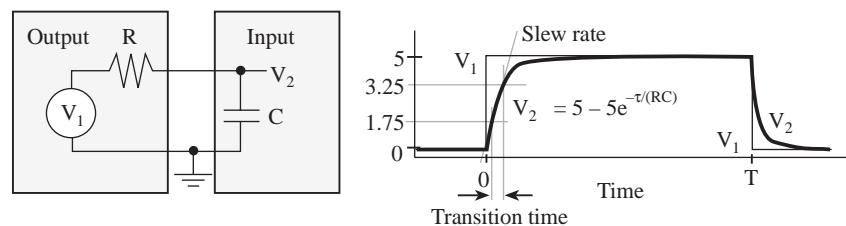


When we design circuits using devices all from a single logic family, we can define *fan out* as the maximum number of inputs one output can drive. For *transistor-transistor logic* (TTL) logic we can calculate *fan out* from the input and output currents:

$$\text{fan out} = \min((I_{OH}/I_{IH}), (I_{OL}/I_{IL}))$$

The fan out of high speed *complementary metal-oxide semiconductor* (CMOS) devices like the MC9S12C32 is determined by capacitive loading and not by the currents. Figure 1.13 shows a simple model of a CMOS interface. The ideal voltage of the output device is labeled  $V_1$ . For interfaces in close proximity, the resistance  $R$  results from the output impedance of the output device, and the capacitance  $C$  results from the input capacitance of the input device. However, if the interface requires a cable to connect the two devices, both the resistance and capacitance will be increased by the cable. The voltage labeled  $V_2$  is the effective voltage as seen by the input. The *slew rate* of a signal is the slope of the voltage versus time during the time when the logic level switches between low and high. A similar parameter is the *transition time*, which is the time it takes for an output to switch from one logic level to another. In Figure 1.13, the transition time is defined as the time it takes  $V_2$  to go from 1.75 to 3.25 V. There is a capacitive load for each CMOS input connected to a CMOS output. As this capacitance increases, the slew rate decreases, which will increase the transition time. The time constant of this simple circuit is  $\tau = R*C$ . Let  $T$  be the pulse width of the digital signal. If  $T$  is large compared

**Figure 1.13**  
Capacitance loading is  
an important factor  
when interfacing CMOS  
devices.



to  $\tau$ , then the CMOS interface functions properly. For circuits that mix devices from one family with another, we must look individually at the input and output currents, voltages, and capacitive loads. Table 1.3 shows typical current values for the various digital logic families. The MC9S12C32 allows full drive (a low R providing 10 mA) and reduced drive (a larger R providing a smaller 2 mA) modes.

**Table 1.3**

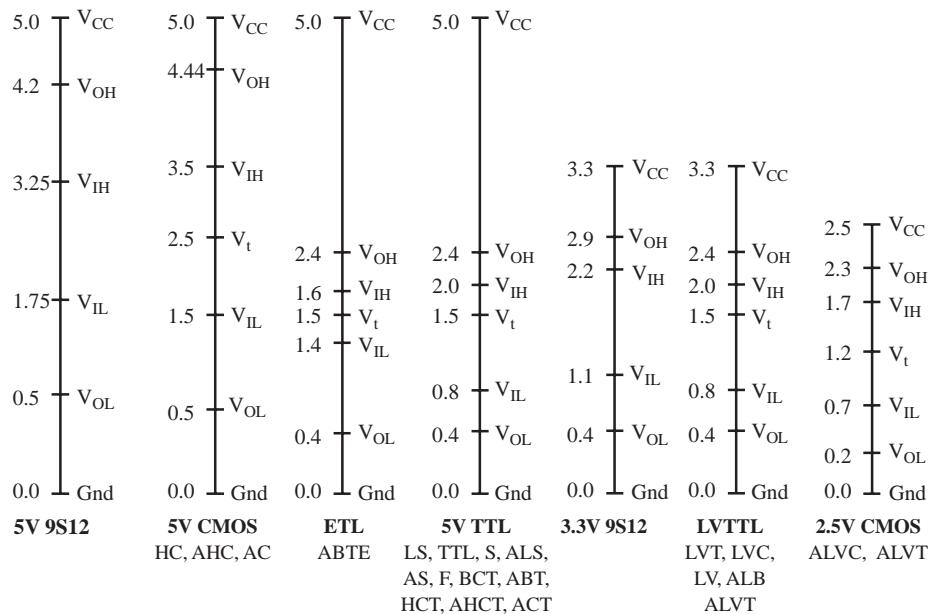
The input and output currents of various digital logic families and microcomputers.

Family	Example	$I_{OH}$	$I_{OL}$	$I_{IH}$	$I_{IL}$	Fan Out
Standard TTL	7404	0.4 mA	16 mA	40 $\mu$ A	1.6 mA	10
Schottky TTL	74S04	1 mA	20 mA	50 $\mu$ A	2 mA	10
Low-power Schottky TTL	74LS04	0.4 mA	4 mA	20 $\mu$ A	0.4 mA	10
High-speed CMOS	74HC04	4 mA	4 mA	1 $\mu$ A	1 $\mu$ A	
Freescale microcomputer	MC68HC11E	0.8 mA	1.6 mA	1 $\mu$ A	1 $\mu$ A	
Freescale microcomputer	MC9S12C32	10 mA	10 mA	1 $\mu$ A	1 $\mu$ A	
Intel microcomputer	87C51 P0	7 mA	3.2 mA	10 $\mu$ A	10 $\mu$ A	
	87C51 P1,P2,P3	60 $\mu$ A	1.6 mA		50 $\mu$ A	

**Observation:** For TTL devices the logic low currents are much larger than the logic high currents.

Figure 1.14 compares the input and output voltages for many of the digital logic families.  $V_{IL}$  is the voltage below which an input is considered a logic low. Similarly,  $V_{IH}$  is the voltage above which an input is considered a logic high.  $V_{OH}$  is the output voltage when the signal is high. In particular, if the output is a logic high, and the current is less than  $I_{OH}$ , then the voltage will be greater than  $V_{OH}$ . Similarly,  $V_{OL}$  is the output voltage when the signal is low. In particular, if the output is a logic low and the current is less than  $I_{OL}$ , then the voltage will be less than  $V_{OL}$ . The maximum output current specification on the 9S12 is 25 mA, which is the current above which it will

**Figure 1.14**  
Voltage thresholds for various digital logic families.



cause damage. Normally, we design the system so the output currents are less than  $I_{OH}$  and  $I_{OL}$ .  $V_t$  is the typical threshold voltage, which is the voltage at which the input usually switches between logic low and high. Formally however, an input is considered as indeterminate for voltages between  $V_{IL}$  and  $V_{IH}$ . The five parameters that affect our choice of logic families are

- Power supply voltage (e.g., +5 V, 3.3 V etc.)
- Power supply current (e.g., will the system need to run on batteries?)
- Speed (e.g., clock frequency and propagation delays)
- Output drive,  $I_{OL}$ ,  $I_{OH}$  (e.g., does it need to drive motors or lights?)
- Noise immunity (e.g., electromagnetic field interference)
- Temperature (e.g., 0 to 70 °C)

**Common error:** If the voltage applied to a high-speed CMOS input pin exists between  $V_{IL}$  and  $V_{IH}$  for extended periods of time, permanent damage may occur.

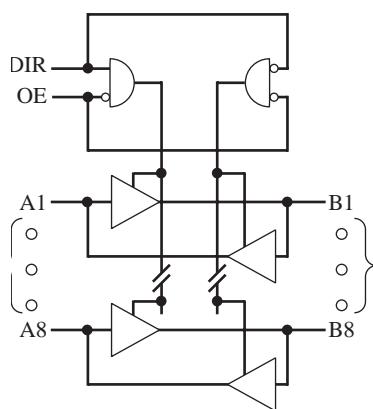
**Checkpoint 1.10:** The 9S12 is an HC device. How will the 9S12 interpret an input pin as the input voltage changes from 0, 1, 2, 3, 4, to 5V? That is, for each voltage, will it be considered as a logic low, as a logic high, or as indeterminate?

**Checkpoint 1.11:** Considering both voltage and current, can the output of a 74HC04 drive the input of a 74LS04?

**Checkpoint 1.12:** Considering both voltage and current, can the output of a 74LS04 drive the input of a 74HC04?

A very important concept used in computer technology is *tristate logic*, which has three output states: high, low, and off. Other names for the off state are HiZ, floating, and tristate. As shown in Figure 1.15, the 74HC245 chip has eight bidirectional tristate drivers. The triangle shape, drawn with a signal on the top or the bottom of the triangle, is used to specify tristate control output in logic diagrams. The 74HC245 chip is active when the output enable, **OE**, input is low, and the direction is controlled by the **DIR** input. For example, if **OE**=0 and **DIR**=0, then the **B1-B8** are inputs and **A1-A8** are outputs, and each output **A<sub>n</sub>** equals the corresponding **B<sub>n</sub>** input. Conversely, if **OE**=0 and **DIR**=1, then the **A1-A8** are inputs and **B1-B8** are outputs, and each output **B<sub>n</sub>** equals the corresponding **A<sub>n</sub>** input. When **OE** is high, all the outputs will be off (floating). Devices like the 74HC245 are used in microcomputer systems because of the large output current and bidirectional tristate outputs. Tables 1.4 and 1.5 illustrate the wide range of technologies available for digital logic design. Not all logic families have the same choice of logic functions.  $t_{pd}$  is the propagation delay from input to output.

**Figure 1.15**  
Block diagram of a 74HC245 tristate driver.



<b>Family Technology</b>	<b>V<sub>IL</sub></b>	<b>V<sub>IH</sub></b>	<b>V<sub>OL</sub></b>	<b>V<sub>OH</sub></b>	<b>I<sub>OL</sub></b>	<b>I<sub>OH</sub></b>	<b>I<sub>CC</sub></b>	<b>t<sub>pd</sub></b>
LVT—Low-Voltage BiCMOS	LVTTL	LVTTL	64	−32	190	3.5		
ALVC—Advanced Low-Voltage CMOS	LVTTL	LVTTL	24	−24	40	3.0		
LVC—Low-Voltage CMOS	LVTTL	LVTTL	24	−24	10	4.0		
ALB—Advanced Low-Voltage BiCMOS	LVTTL	LVTTL	25	−25	800	2.0		
AC—Advanced CMOS	CMOS	CMOS	12	−12	20	8.5		
AHC—Advanced high-Speed CMOS	CMOS	CMOS	4	−4	20	11.9		
LV—Low-Voltage CMOS	LVTTL	LVTTL	8	−8	20	14		
	Units				mA	mA	μA	ns

**Table 1.4**

Comparison of the output drive, power supply current, and speed of various 3.3 V logic '245 gates.

<b>Family Technology</b>	<b>V<sub>IL</sub></b>	<b>V<sub>IH</sub></b>	<b>V<sub>OL</sub></b>	<b>V<sub>OH</sub></b>	<b>I<sub>OL</sub></b>	<b>I<sub>OH</sub></b>	<b>I<sub>CC</sub></b>	<b>t<sub>pd</sub></b>
AHC—Advanced High-Speed CMOS	CMOS	CMOS	8	−8	0.04	7.5		
AHCT—Advanced High-Speed CMOS	TTL	CMOS	8	−8	0.04	7.7		
AC—Advanced CMOS	CMOS	CMOS	24	−24	0.04	6.5		
ACT—Advanced CMOS	TTL	CMOS	24	−24	0.04	8.0		
HC—High-Speed CMOS Logic	CMOS	CMOS	6	−6	0.08	21		
HCT—High-Speed CMOS Logic	TTL	CMOS	6	−6	0.08	30		
ABT—Advanced BiCMOS	TTL	TTL	64	−32	0.25	3.5		
74F—Fast Logic	TTL	TTL	64	−15	120	6.0		
BCT—BiCMOS	TTL	TTL	64	−15	90	6.6		
AS—Advanced Schottky Logic	TTL	TTL	64	−15	143	7.5		
ALS—Advanced Low-Power Schottky	TTL	TTL	24	−15	58	10		
LS—Low Power Schottky logic	TTL	TTL	24	−15	95	12		
S—Schottky Logic	TTL	TTL	64	−15	180	9		
TTL—Transistor-Transistor Logic	TTL	TTL	16	−0.4	22	22		
	Units				mA	mA	mA	ns

**Table 1.5**

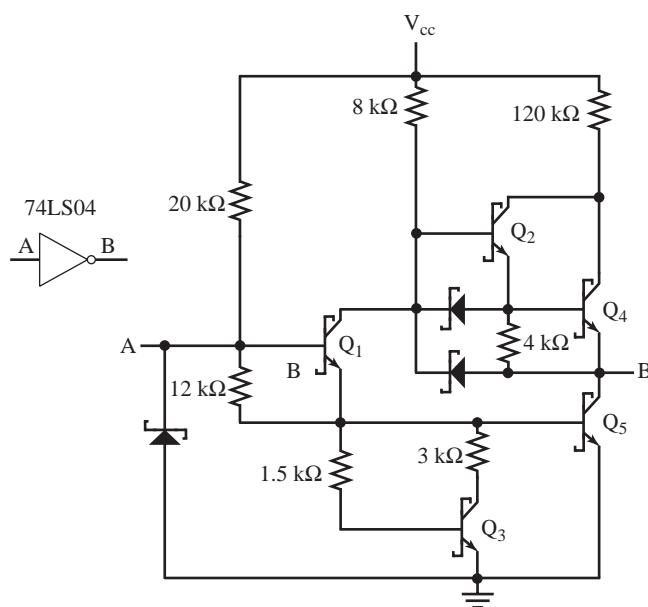
Comparison of the output drive, power supply current, and speed of various 5 V logic '245 gates.

The 74LS04 is a low-power Schottky NOT gate, as shown in Figure 1.16. It is called Schottky logic because the devices are made from Schottky transistors. The output is *high* when the transistor Q4 is active, driving the output to +5V. The output is *low* when the transistor Q5 is active, driving the output to 0.

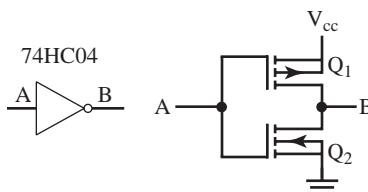
The 74HC04 is a high-speed CMOS NOT gate, as shown in Figure 1.17. The output is *high* when the transistor Q1 is active, driving the output to +5V. The output is *low* when the transistor Q2 is active, driving the output to 0. Since the 9S12 is made with high-speed CMOS logic, its outputs behave like the Q1/Q2 “push/pull” transistor pair. The 9S12 output ports are not inverting. That is, when you write a “1” to an output port, then the output voltage goes high. Similarly, when you write a “0” to an output port, then the output voltage goes low. Analyses of the circuit in Figure 1.17 reveal some of the basic properties of high-speed CMOS logic. First, because of the complementary nature of the P-channel (the one on the top) and the N-channel (the one on the bottom) transistors, when the input is constant (continuously high or continuously low), the supply current,  $I_{cc}$ , is very low. Second, the gate will require supply current only when the output switches from low to high or from high to low. This observation leads to the design rule that the power required to run a high-speed CMOS system is linearly related to the frequency of its clock, because the frequency of the clock determines the number of transitions per second. Along the same lines,

**Figure 1.16**

Transistor implementation of a low-power Schottky NOT gate.

**Figure 1.17**

Transistor implementation of a high-speed CMOS NOT gate.



we see that if the voltage on input A exists between  $V_{IL}$  and  $V_{IH}$  for extended periods of time, then both Q1 and Q2 are partially active, causing a short from  $V_{cc}$  to ground. This condition can cause permanent damage to the transistors. Third, since the input A is connected to the gate of the two MOS transistors, the input currents will be very small ( $1 \mu\text{A}$ ). In other words, the input impedance (input voltage divided by input current) of the gate is very high. Normally, a high input impedance is a good thing, except when the input is not connected. If the input is not connected, then it takes very little input currents to cause the logic level to switch.

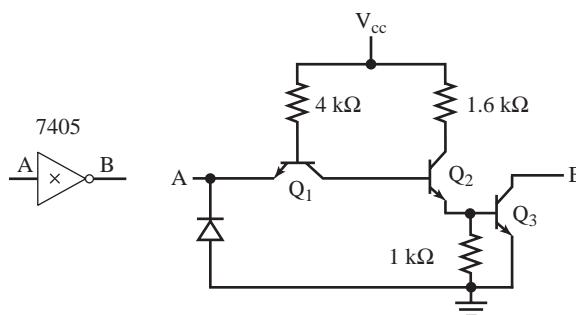
**Common error:** If unused input pins on a CMOS microcontroller are left unconnected, then the input signal may oscillate at high frequencies depending on the EM fields in the environment, wasting power unnecessarily.

**Maintenance tip:** It is a good design practice to connect unused CMOS inputs to ground or connect them to +5 V.

*Open collector* logic has outputs with two states: low and off. The 7405 is a TTL open collector NOT gate, as shown in Figure 1.18. When drawing logic diagrams, we add the “x” on the output to specify open collector logic. It is called open collector because the collector pin of Q3 is not connected, or left open. The output is off when there is no active transistor driving the output. In other words, when the input is low, the output floats. This “not driven” condition is called the open collector state. The output is low when the transistor Q3 is active, driving the output to 0.

**Figure 1.18**

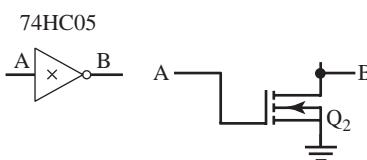
Transistor implementation of a regular TTL open collector NOT gate.



The 74HC05 is a high-speed CMOS open collector NOT gate, as shown in Figure 1.19. The output is *off* when there is no active transistor driving the output. In other words, when the input is low, the output floats. The output is low when the transistor Q2 is active driving the output to 0. Technically, the 74HC05 implements *open drain* rather than open collector, because it is the drain pin of Q2 that is left open. In this book, we will use the terms open collector and open drain interchangeably to refer to digital logic with two output states (low and off). The data sheets of the 9S12 refer to open collector logic as *wire or mode* (WOM).

**Figure 1.19**

Transistor implementation of a high-speed CMOS open collector NOT gate.



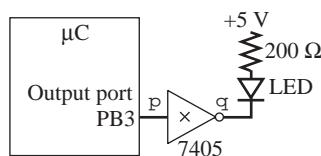
Because of the multiple uses of open collector, many microcomputers can implement open collector logic. All the ports of the Intel 8051 are inherently open collector. The 9S12 PORTS also supports open collector outputs by setting the SWOM bit.

**Observation:** The 7405 and 74HC05 are inverting open collector examples, but most microcomputer output ports are not inverting. That is, when you write a "1" to an open-collector output port, then the output floats, and when you write a "0" to an open-collector output port, then the output voltage goes low.

In general, we can use an **open collector NOT** gate to control the current to a device, such as a relay, a *light emitting diode* (LED), a solenoid, a small motor, or a small light. We used the open collector NOT gate in the LED interface shown in Figure 1.20 to control the current to our diode. When input to the 7405 is high ( $p = 1$ , which means +5 V), the output is low ( $q=0$ , which means 0 V). In this state, a 10 mA current is applied to the diode, and it lights up. But, when the input is low ( $p = 0$ , which means +0 V), the output floats ( $q = \text{HiZ}$ , which is neither high or low). This floating output state causes the LED current to be zero, and the diode is dark. The resistor choice will be explained in Chapter 8.

**Figure 1.20**

Open collector used to interface a light emitting diode.



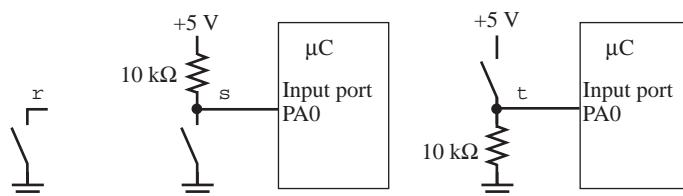
When needed for digital logic, we can convert an open collector output to a digital signal using a pull-up resistor from the output to +5V. In this way, when the open collector output floats, the signal will be a digital high. How do we select the value of the pullup resistor? In general the smaller the resistor, the larger the  $I_{OH}$  it will be able to supply when the output is high. On the other hand, a larger resistor does not waste as much  $I_{OL}$  current when the output is low. One way to calculate the value of this pull-up resistor is to first determine the required output high voltage,  $V_{out}$ , and output high current,  $I_{out}$ . To supply a current of at least  $I_{out}$  at a voltage above  $V_{out}$ , the resistor must be less than:

$$R \leq (+5 - V_{out})/I_{out}$$

As an example, we will calculate the resistor value for the situation where the circuit needs to drive five regular TTL loads. We see from Figure 1.14 that  $V_{out}$  must be above  $V_{IH}$  (2V) in order for the TTL inputs to sense a high logic level. We can add a safety factor and set  $V_{out}$  at 3 V. In order for the high output to drive all five TTL inputs,  $I_{out}$  must be more than five  $I_{IH}$ . From Table 1.3, we see that  $I_{IH}$  is 40  $\mu$ A, so  $I_{out}$  should be larger than  $5 \cdot 40 \mu$ A or 0.2 mA. For this situation the resistor must be less than 10 k $\Omega$ .

Another example of open collector logic occurs when interfacing switches to the microcontroller. The circuit in the left of Figure 1.21 shows a mechanical switch with one terminal connected to ground. In this circuit, when the switch is pressed, the voltage  $r$  is zero. When the switch is not processed, the signal  $r$  floats. The circuit in the middle of Figure 1.21 shows the mechanical switch with a 10 k pull-up resistor attached the other side. When the switch is pressed, the voltage at  $s$  still goes to zero, because the resistance of the switch (less than 0.1  $\Omega$ ) is much less than the pull-up resistor. But now, when the switch is not pressed, the pull-up resistor creates a +5 V at  $s$ . This circuit is shown connected to an input pin of the microcontroller. The software, by reading the input port, can determine whether or not the switch is processed. If the switch is pressed, the software will read zero, and if the switch is not pressed, the software will read one. The circuit on the right of Figure 1.21 also interfaces a mechanical switch to the microcontroller, but it implements positive logic using a pull-down resistor. The signal  $t$  will be high if the switch is pressed and low if it is released.

**Figure 1.21**  
Switch interface.



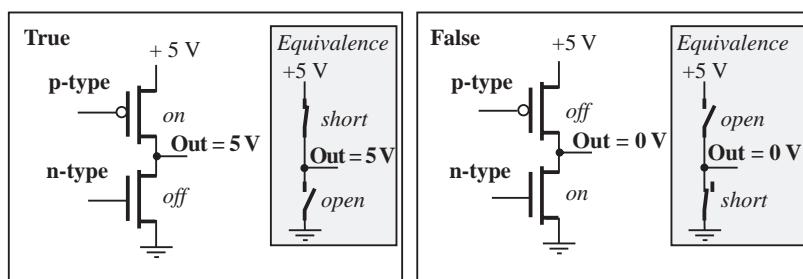
**Observation:** Some of the ports on the 9S12 implement pull-up or pull-down resistors, so the interfaces shown in Figure 1.21 can be made without the resistor.

## 1.5 Digital Representation of Numbers

**1.5.1 Fundamentals** Information is stored on the computer in binary form. A binary *bit* can exist in one of two possible states. In *positive logic*, the presence of a voltage is called the 1, true, asserted, or high state. The absence of a voltage is called the 0, false, not asserted, or low state. Figure 1.22 shows the output of a typical complementary metal-oxide semiconductor (CMOS) circuit. The left side shows the condition with a true bit, and the right side shows a false bit. The output of each digital circuit consists of a p-type transistor “on top of” an n-type transistor. In digital circuits, each transistor is essentially on or off. If the transistor is *on*, it is equivalent to a short circuit between its two output pins. Conversely, if the transistor is *off*, it is equivalent to an open circuit between its outputs pins.

**Figure 1.22**

A binary bit is true if a voltage is present and false if the voltage is 0.



On a 9S12 powered with 5 V supply, a voltage between 3.25 and 5 V is considered high, and a voltage between 0 and 1.75 V is considered low. Separating the two regions by 1.5 V allows digital logic to operate reliably at very high speeds. The design of transistor-level digital circuits is beyond the scope of this book. However, it is important to know that digital data exist as binary bits and encoded as high and low voltages.

Numbers are stored on the computer in binary form. In other words, information is encoded as a sequence of 1's and 0's. On most computers, the memory is organized into 8-bit bytes. This means each 8-bit byte stored in memory will have a separate address. *Precision* is the number of distinct or different values. We express precision in alternatives, decimal digits, bytes, or binary bits. *Alternatives* are defined as the total number of possibilities. For example, an 8-bit number scheme can represent 256 different numbers. An 8-bit *digital to analog converter* (DAC) can generate 256 different analog outputs. An 8-bit *analog to digital converter* (ADC) can measure 256 different analog inputs. We use the expression  $4^{1/2}$  decimal digits to mean 20,000 alternatives and the expression  $4^{3/4}$  decimal digits to mean 40,000 alternatives. The  $\frac{1}{2}$  decimal digit means twice the number of alternatives or one additional binary bit. The  $\frac{3}{4}$  decimal digit means four times as many alternatives or two additional binary bits. For example, a voltmeter with a range of 0.00 to 9.99 V has a three decimal digit precision. Let the operation  $[[x]]$  be the greatest integer of x. E.g.,  $[[2.1]]$  is rounded up to 3. Tables 1.6 and 1.7 illustrate various representations of precision.

**Table 1.6**  
Relationship between bits, bytes, and alternatives as units of precision.

Binary Bits	Bytes	Alternatives
8	1	256
10		1024
12		4096
16	2	65536
20		1,048,576
24	3	16,777,216
30		1,073,741,824
32	4	4,294,967,296
n	$[[n/8]]$	$2^n$

**Table 1.7**  
Definition of decimal digits as a unit of precision.

Decimal Digits	Alternatives
3	1000
$3^{1/2}$	2000
$3^{3/4}$	4000
4	10000
$4^{1/2}$	20000
$4^{3/4}$	40000
5	100000
n	$10^n$

**Observation:** A good rule of thumb to remember is  $2^{10 \cdot n} \approx 10^{3 \cdot n}$ .

**Checkpoint 1.13:** How many binary bits correspond to  $2^{1/2}$  decimal digits?

**Checkpoint 1.14:** About how many decimal digits can be presented in a 64-bit 8-byte number? You can answer this without a calculator, just using the “rule of thumb.”

The *hexadecimal* number system uses base 16 as opposed to our regular decimal number system, which uses base 10. Hexadecimal is a convenient mechanism for humans to represent binary information, because it is extremely simple for us to convert back and forth between binary and hexadecimal. Hexadecimal number system is often abbreviated as “hex.” A *nibble* is defined as 4 binary bits, which will be one hexadecimal digit. In mathematics, a subscript of 2 means binary, but in assembly language we will use the prefix % to signify binary numbers. The hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In assembly language, we use the prefix \$ to signify hexadecimal, and in C, we use the prefix 0x. To convert from binary to hexadecimal, you simply separate the binary number into groups of four binary bits (starting on the right), then convert each group of four bits into one hexadecimal digit. For example, if you wished to convert %10100111, first you would group it into nibbles 1010 0111, then you would convert each group 1010=A 0111=7, yielding the result of \$A7. To convert hexadecimal to binary, you simply substitute the 4-bit binary for each hexadecimal digit. For example, if you wished to convert \$B5D1, you substitute B=1011 5=0101 D=1101 1=0001, yielding the result of %1011010111010001.

**Checkpoint 1.15:** Convert the binary number %111011101011 to hexadecimal.

**Checkpoint 1.16:** Convert the hex number \$3800 to binary.

**Checkpoint 1.17:** How many binary bits does it take to represent \$12345?

A great deal of confusion exists over the abbreviations we use for large numbers. In 1998, the International Electrotechnical Commission (IEC) defined a new set of abbreviations for the powers of 2, as shown in Table 1.8. These new terms are endorsed by the Institute of Electrical and Electronics Engineers (IEEE) and International Committee for Weights and Measures (CIPM) in situations where the use of a binary prefix is appropriate. The confusion arises over the fact that the mainstream computer industry (such as Microsoft, Apple, and Dell) continues to use the old terminology. According to the companies that market to consumers, a 1 GHz is 1,000,000,000 Hz, but 1 Gbyte of memory is 1,073,741,824 bytes. The correct terminology is to use the SI-decimal abbreviations to represent powers of 10 and the IEC-binary abbreviations to represent powers of 2. The scientific meaning of 2 kilovolts is 2000 volts, but 2 kibibytes is the proper way to specify 2048 bytes. The term **kibibyte** is a contraction of a kilo binary byte and is a unit of information or computer storage abbreviated KiB.

$$1 \text{ KiB} = 2^{10} \text{ bytes} = 1024 \text{ bytes}$$

$$1 \text{ MiB} = 2^{20} \text{ bytes} = 1,048,576 \text{ bytes}$$

$$1 \text{ GiB} = 2^{30} \text{ bytes} = 1,073,741,824 \text{ bytes}$$

**Table 1.8**  
Common abbreviations  
for large numbers.

Value	SI	Decimal	Value	IEC	Binary
1000 <sup>1</sup>	k	kilo-	1024 <sup>1</sup>	Ki	kibi-
1000 <sup>2</sup>	M	mega-	1024 <sup>2</sup>	Mi	mebi-
1000 <sup>3</sup>	G	giga-	1024 <sup>3</sup>	Gi	gibi-
1000 <sup>4</sup>	T	tera-	1024 <sup>4</sup>	Ti	tebi-
1000 <sup>5</sup>	P	peta-	1024 <sup>5</sup>	Pi	pebi-
1000 <sup>6</sup>	E	exa-	1024 <sup>6</sup>	Ei	exbi-
1000 <sup>7</sup>	Z	zetta-	1024 <sup>7</sup>	Zi	zebi-
1000 <sup>8</sup>	Y	yotta-	1024 <sup>8</sup>	Yi	yobi-

These abbreviations also can be used to specify the number of binary bits. The term **kibibit** is a contraction of kilo binary bit and is a unit of information or computer storage abbreviated Kibit.

$$1 \text{ Kibit} = 2^{10} \text{ bits} = 1024 \text{ bits}$$

$$1 \text{ Mibit} = 2^{20} \text{ bits} = 1,048,576 \text{ bits}$$

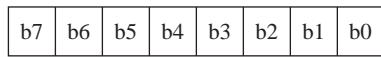
$$1 \text{ Gibit} = 2^{30} \text{ bits} = 1,073,741,824 \text{ bits}$$

A **mebibyte** (1 MiB is 1,048,576 bytes) is approximately equal to a megabyte (1 MB is 1,000,000 bytes), but mistaking the two has nonetheless led to confusion and even legal disputes. In the engineering community, it is appropriate to use terms that have a clear and unambiguous meaning.

## 1.5.2 8-Bit Numbers

A byte contains 8 bits as shown in Figure 1.23, where each bit  $b_7, \dots, b_0$  is binary and has the value 1 or 0. We specify  $b_7$  as the *most significant bit* or MSB, and  $b_0$  as the least significant bit or LSB.

**Figure 1.23**  
8-bit binary format.



If a byte is used to represent an unsigned number, then the value of the number is

$$N = 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0$$

Notice that the significance of bit n is  $2^n$ . There are 256 different unsigned 8-bit numbers. The smallest unsigned 8-bit number is 0, and the largest is 255. For example, %00001010 is 8+2 or 10.

**Checkpoint 1.18:** Convert the binary number %01101010 to unsigned decimal.

**Checkpoint 1.19:** Convert the hex number \$32 to unsigned decimal.

The *basis* of a number system is a subset from which linear combinations of the basis elements can be used to construct the entire set. The basis represents the “places” in a “place-value” system. For positive integers, the basis is the infinite set {1, 10, 100, ...}, and the “values” can range from 0 to 9. Each positive integer has a unique set of values such that the dot-product of the **value-vector** times the **basis-vector** yields that number. For example, 2345 is (..., 2, 3, 4, 5) • (..., 1000, 100, 10, 1), which is  $2 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 5$ . For the unsigned 8-bit number system, the basis is

$$\{1, 2, 4, 8, 16, 32, 64, 128\}$$

The values of a binary number system can only be 0 or 1. Even so, each 8-bit unsigned integer has a unique set of values such that the dot-product of the values times the basis yields that number. For example, 69 is (0, 1, 0, 0, 0, 1, 0, 1) • (128, 64, 32, 16, 8, 4, 2, 1), which equals  $0 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$ .

**Checkpoint 1.20:** Give the representations of decimal 35 in 8-bit binary and hexadecimal.

**Checkpoint 1.21:** Give the representations of decimal 200 in 8-bit binary and hexadecimal.

One of the first schemes to represent signed numbers was called *one’s complement*. It was called one’s complement because to negate a number, you complement (logical not) each bit. For example, if 25 equals 00011001 in binary, then -25 is 11100110. An 8-bit one’s complement number can vary from -127 to +127. The most significant bit is a sign bit, which is

1 if and only if the number is negative. The difficulty with this format is that there are two zeros +0 is 00000000, and -0 is 11111111. Another problem is that ones complement numbers do not have basis elements. These limitations led to the use of two's complement.

The *two's complement* number system is the most common approach used to define signed numbers. It was called two's complement because to negate a number, you complement each bit (like one's complement), then add 1. For example, if 25 equals 00011001 in binary, then -25 is 11100111. If a byte is used to represent a signed two's complement number, then the value of the number is

$$N = -128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0$$

**Observation:** One usually means two's complement when one refers to signed integers.

There are 256 different signed 8-bit numbers. The smallest signed 8-bit number is -128, and the largest is 127. For example, %10000010 equals -128+2 or -126.

**Checkpoint 1.22:** Are the signed and unsigned decimal representations of the 8-bit hex number \$35 the same or different?

For the signed 8-bit number system the basis is

$$\{1, 2, 4, 8, 16, 32, 64, -128\}$$

**Observation:** The most significant bit in a two's complement signed number will specify the sign.

Notice that the same binary pattern of %11111111 could represent either 255 or -1. It is very important for the software developer to keep track of the number format. The computer can not determine whether the 8-bit number is signed or unsigned. You, as the programmer, will determine whether the number is signed or unsigned by the specific assembly instructions you select to operate on the number. Some operations like addition, subtraction, and shift left (multiply by 2) use the same hardware (instructions) for both unsigned and signed operations. On the other hand, multiply, divide, and shift right (divide by 2) require separate hardware (instruction) for unsigned and signed operations. For example, the 9S12 multiply instruction, `mul`, operates only on unsigned values. So if you use the `mul` instruction, you are implementing unsigned arithmetic. The 9S12 has both unsigned, `emul`, and signed, `emuls`, multiply instructions. So if you use the `emuls` instruction, you are implementing signed arithmetic.

**Observation:** To take the negative of a two's complement signed number we first complement (flip) all the bits, then add 1.

**Checkpoint 1.23:** Give the representations of -35 in 8-bit binary and hexadecimal.

**Checkpoint 1.24:** Why can't you represent the number 200 using 8-bit signed binary?

**Common error:** An error will occur if you use signed operations on unsigned numbers, or use unsigned operations on signed numbers.

**Maintenance tip:** To improve the clarity of your software, always specify the format of your data (signed versus unsigned) when defining or accessing the data.

### 1.5.3 Character Information

We can use bytes to represent characters with the **American Standard Code for Information Interchange** (ASCII) code. Standard ASCII is actually only 7 bits, but is stored using 8-bit bytes with the most significant byte equal to 0. Some computer systems use the 8th bit of the ASCII code to define additional characters such as graphics and letters in other alphabets. The 7-bit ASCII code definitions are given in the Table 1.9. For example, the letter "V" is in the \$50 row and the 6 column. Putting the two together yields hexadecimal \$56.

**Table 1.9**  
Standard 7-bit ASCII.

		BITS 4 to 6							
		0	1	2	3	4	5	6	7
B	0	NUL	DLE	SP	0	@	P	`	p
I	1	SOH	XON	!	1	A	Q	a	q
T	2	STX	DC2	"	2	B	R	b	r
S	3	ETX	XOFF	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
O	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
T	8	BS	CAN	(	8	H	X	h	x
O	9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z	
3	B	VT	ESC	+	K	[	k	{	
C	FF	FS	,	<	L	\	l		
D	CR	GS	-	=	M	]	m	}	
E	SO	RS	.	>	N	^	n	~	
F	S1	US	/	?	O	_	o		DEL

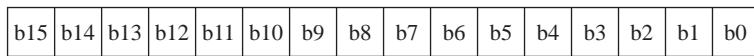
**Checkpoint 1.25:** How is the character O represented in ASCII?

One way to encode a character string is to use null-termination. In this way, the characters of the string are stored one right after the other, and the end of the string is signified by the NUL character (0). For example, the string “Valvano” is encoded as the following eight bytes: \$56,\$61,\$6C,\$76,\$61,\$6E,\$6F,\$00.

**Checkpoint 1.26:** How is “Hello World” encoded as a null-terminated ASCII string?

**1.5.4  
16-Bit Numbers** A word or double byte contains 16 bits, where each bit  $b_{15}, \dots, b_0$  is binary and has the value 1 or 0, as shown in Figure 1.24.

**Figure 1.24**  
16-bit binary format.



If a word is used to represent an unsigned number, then the value of the number is

$$\begin{aligned} N = & 32768 \cdot b_{15} + 16384 \cdot b_{14} + 8192 \cdot b_{13} + 4096 \cdot b_{12} \\ & + 2048 \cdot b_{11} + 1024 \cdot b_{10} + 512 \cdot b_9 + 256 \cdot b_8 \\ & + 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \end{aligned}$$

There are 65536 different unsigned 16-bit numbers. The smallest unsigned 16-bit number is 0, and the largest is 65535. For example, %0010000110000100 or \$2184 is  $8192+256+128+4$  or 8580. For the unsigned 16-bit number system the basis is

$$\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768\}$$

There are also 65536 different signed 16-bit numbers. The smallest two’s complement signed 16-bit number is -32768 and the largest is 32767. For example, %110100000000100 or \$D004 is  $-32768+16384+4096+4$  or -12284. If a word is used to represent a signed two’s complement number, then the value of the number is

$$\begin{aligned} N = & -32768 \cdot b_{15} + 16384 \cdot b_{14} + 8192 \cdot b_{13} + 4096 \cdot b_{12} \\ & + 2048 \cdot b_{11} + 1024 \cdot b_{10} + 512 \cdot b_9 + 256 \cdot b_8 \\ & + 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \end{aligned}$$

For the signed 16-bit number system the basis is

$$\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, -32768\}$$

**Common error:** An error will occur if you use 16-bit operations on 8-bit numbers, or use 8-bit operations on 16-bit numbers.

**Maintenance tip:** To improve the clarity of your software, always specify the precision of your data when defining or accessing the data.

When we store 16-bit data into memory, it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Freescale microcomputers implement the *big endian* approach, which stores the most significant part first. Intel microcomputers implement the *little endian* approach, which stores the least significant part first. The PowerPC is *bi endian*, because it can be configured to efficiently handle both big and little endian. Figure 1.25 shows two ways to store the 16-bit number 1000 (\$03E8) at locations \$50–\$51.

**Figure 1.25**

Example of big and little endian formats of a 16-bit number.

Address	Contents	Address	Contents
\$0050	\$03	\$0050	\$E8
\$0051	\$E8	\$0051	\$03

Big Endian                      Little Endian

We also can use either the big endian or the little endian approach when storing 32-bit numbers into memory that is byte (8-bit) addressable. Figure 1.26 shows the big and little endian formats that could be used to store the 32-bit number \$12345678 at locations \$50–\$53.

**Figure 1.26**

Example of big and little endian formats of a 32-bit number.

Address	Contents	Address	Contents
\$0050	\$12	\$0050	\$78
\$0051	\$34	\$0051	\$56
\$0052	\$56	\$0052	\$34
\$0053	\$78	\$0053	\$12

Big Endian                      Little Endian

In the foregoing two examples, we normally would not pick out individual bytes (e.g., the \$12) but rather capture the entire multiple byte data as one nondivisible piece of information. On the other hand, if each byte in a multiple byte data structure is individually addressable, then both the big and little endian schemes store the data in first-to-last sequence. For example, if we wish to store the four ASCII characters ‘9\$12’ as a string, which is \$3953313200 at locations \$50–\$54, then the ASCII ‘9’ = \$39 comes first in both big and little endian schemes.

The terms “big and little endian” comes from Jonathan Swift’s satire *Gulliver’s Travels*. In Swift’s book, a Big Endian refers to people who crack their egg on the big end. The Lilliputians were Little Endians, because they insisted that the only proper way is to break an egg on the little end. The Lilliputians considered the Big Endians as inferiors. The Big and Little Endians fought a long and senseless war over which end is best to crack an egg.

## 1.5.5 Fixed-Point Numbers

We will use fixed-point numbers when we wish to express values in our software that have noninteger values. A **fixed-point number** contains two parts. The first part is a **variable integer**, called **I**. This integer may be signed or unsigned. An unsigned fixed-point number

is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer. The **precision** of a number is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. On the 9S12, we typically use 8 bits or 16 bits. Extended precision can be implemented, but the execution speed will be slower because the calculations will have to be performed using software algorithms rather than hardware instructions. This integer part is saved in memory and is manipulated by software. These manipulations include but are not limited to add, subtract, multiply, divide, convert to BCD and convert from BCD. The second part of a fixed-point number is a **fixed constant**, called  $I$ . This value is fixed, and cannot be changed during execution of the program. The fixed constant is not stored in memory. Usually we specify the value of this fixed constant using software comments to explain our fixed-point algorithm. The value of the fixed-point number is defined as the product of the two parts:

$$\text{fixed-point number } I \bullet$$

The **resolution** of a number is the smallest difference that can be represented. In the case of fixed-point numbers, the resolution is equal to the fixed constant ( $I$ ). Sometimes we express the resolution of the number as its units. For example, a decimal fixed-point number with a resolution of 0.001V is really the same thing as an integer with units of mV. When interacting with a human operator, it is usually convenient to use **decimal fixed-point**. With decimal fixed-point the fixed constant is a power of 10.

$$\text{decimal fixed-point number} = I \bullet 10^m \text{ for some constant integer } m$$

Again, the integer  $m$  is fixed and is not stored in memory. Decimal fixed-point is easy to display, whereas **binary fixed-point** is easier to use when performing mathematical calculations. With binary fixed-point the fixed constant is a power of 2.

$$\text{binary fixed-point number} = I \bullet 2^n \text{ for some constant integer } n$$

**Observation:** If the range of numbers is known and small, then the numbers can be represented in a fixed-point format.

**Checkpoint 1.27:** Give an approximation of  $\pi$ , using the decimal fixed-point ( $= 0.001$ ) format.

**Checkpoint 1.28:** Give an approximation of  $\pi$ , using the binary fixed-point ( $= 2^{-8}$ ) format.

An 8-bit ADC has a range of 0 to +5 V, and its resolution is about 5 V/256 or 0.02 V. It would be appropriate to store voltages as 16-bit unsigned fixed-point numbers with a resolution of 0.01V. For the 9S12C32, which has a 10-bit ADC and a range of 0 to +5 V, its resolution is about 5 V/1024 or 0.005 V. It would be appropriate to store voltages on the 9S12C32 as 16-bit unsigned fixed-point numbers with a resolution of 0.001 V. The resolution is chosen so that no information is lost.

It is very important to carefully consider the order of operations when performing multiple integer calculations. For example, assume we wished to calculate  $M=(53*N)/100$ , where  $M$  and  $N$  are integers. There are two mistakes that can happen. The first error is **overflow**, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. In this example, if  $N$  is an 8-bit unsigned number, then  $53*N$  can overflow the 0 to 255 range. One solution of the overflow problem is **promotion**. **Promotion** is the action of increasing the inputs to a higher precision, performing the calculation at the higher precision, checking for overflow, then demoting the result back to the lower precision. In this example, the 53,  $N$ , and 100 are all converted to 16-bit unsigned numbers.  $(53*N)/100$  is calculated in 16-bit precision. The result can be verified to be in the 0 to 255 range, then converted back to 8-bit precision. The other error is called **drop-out**. Drop-out occurs during a right shift or a divide, and the consequence is that an intermediate

result loses its ability to represent all of the values. To avoid drop-out, it is very important to divide last when performing multiple integer calculations. If we divided first, e.g.,  $M=53*(N/100)$ , then the values of M would be only 0, 53, or 106. We could have calculated  $M=(53*N+50)/100$  to implement rounding to the closest integer. The value 50 is selected because it is about one half of the divisor.

When adding or subtracting two fixed-point numbers with the same  $\bullet$ , we simply add or subtract their integer parts. First, let  $x,y,z$  be three fixed-point numbers with the same  $\bullet$ . Let  $x=I\bullet$ ,  $y=J\bullet$ , and  $z=K\bullet$ . To perform  $z=x+y$ , we simply calculate  $K=I+J$ . Similarly, to perform  $z=x-y$ , we simply calculate  $K=I-J$ . When adding or subtracting fixed-point numbers with different fixed parts, we must first convert two the inputs to the format of the result before adding or subtracting. This is where binary fixed-point is more convenient, because the conversion process involves shifting rather than multiplication/division.

For multiplication, we have  $z=x\bullet y$ . Again, we substitute the definitions of each fixed-point parameter and solve for the integer part of the result

$$K=I\bullet J\bullet$$

For division, we have  $z=x/y$ . Again, we substitute the definitions of each fixed-point parameter and solve for the integer part of the result

$$K=I/J/$$

Again, it is very important to carefully consider the order of operations when performing multiple integer calculations. We must worry about overflow and drop-out.

We can use these fixed-point algorithms to perform complex operations using the integer functions of our 9S12. For example, consider the following digital filter calculation.

$$y=x-0.0532672\bullet x_1+x_2+0.0506038\bullet y_1-0.9025\bullet y_2$$

In this case, the variables  $y$ ,  $y_1$ ,  $y_2$ ,  $x$ ,  $x_1$ , and  $x_2$  are all integers, but the constants will be expressed in binary fixed-point format. The value  $-0.0532672$  will be approximated by  $-14\bullet 2^{-8}$ . The value  $0.0506038$  will be approximated by  $13\bullet 2^{-8}$ . Lastly, the value  $-0.9025$  will be approximated by  $-231\bullet 2^{-8}$ . The fixed-point implementation of this digital filter is

$$y=(256\bullet x - 14\bullet x_1 + 256\bullet x_2 + 13\bullet y_1 - 231\bullet y_2) \gg 8$$

**Common error:** Lazy or incompetent programmers use floating-point in many situations where fixed-point would be preferable.

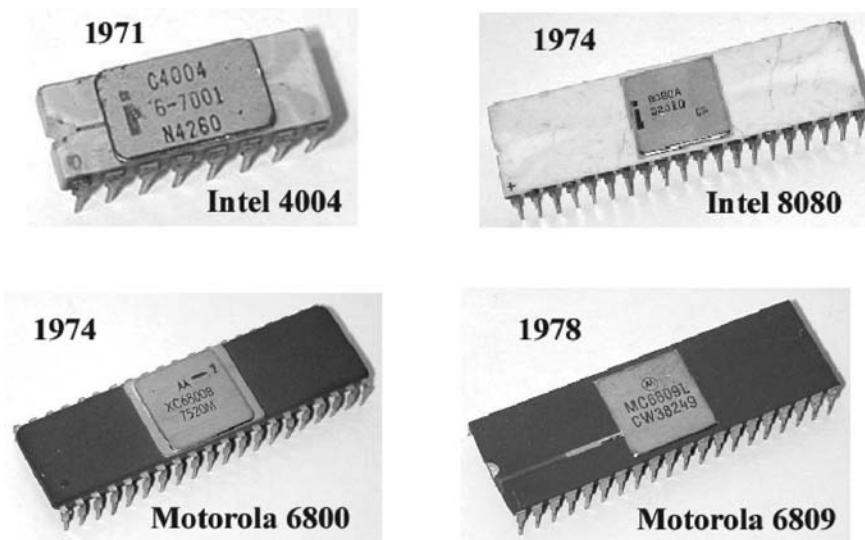
**Observation:** As the fixed constant is made smaller, the accuracy of the fixed-point representation is improved, but the variable integer part also increases. Unfortunately, larger integers require more bits for storage and calculations.

**Checkpoint 1.29:** Using a fixed constant of  $10^{-3}$ , rewrite the digital filter  $y=x-0.0532672\bullet x_1+x_2+0.0506038\bullet y_1-0.9025\bullet y_2$  in decimal fixed-point format.

## 1.6 Common Architecture of the 9S12

In 1968, two unhappy engineers named Bob Noyce and Gordon Moore left the Fairchild Semiconductor Company and created their own company, which they called Integrated Electronics (Intel). Working for Intel in 1971, Federico Faggin, Ted Hoff, and Stan Mazor invented the first single-chip microprocessor, the Intel 4004 (Figure 1.27). It was a four-bit processor designed to solve a very specific application for a Japanese company called Busicon. Busicon backed out of the purchase, so Intel decided to market it as a “general purpose” microprocessing system. The product was a success, which lead to two more powerful microprocessors: the Intel 8008 in 1974, and the Intel 8080 also in 1974. Both the Intel 8008 and the Intel 8080 were 8-bit microprocessors using N-channel metal-oxide semiconductor (NMOS) technology. Seeing the long-term potential for this technology,

**Figure 1.27**  
The first microprocessors  
(<http://www.cpu-world.com>).



[www.cpu-world.com](http://www.cpu-world.com)

Motorola released its MC6800 in 1974, which was also an 8-bit processor with about the same capabilities as the 8080. Although similar in computing power, the 8080 and 6800 had very different architectures. The 8080 used isolated I/O and handled addresses in a fundamentally different way from data. Isolated I/O defines special hardware signals and special instructions for input/output. On the 8080, certain registers had capabilities designed for addressing, whereas other registers had capabilities for data manipulation. In contrast, the 6800 used memory-mapped I/O and handled addresses and data in a similar way. As defined previously, input/output on a system with memory-mapped I/O is performed in a manner similar to accessing memory.

During the 1980s and 1990s, Motorola (von Neumann architecture) and Intel (Harvard architecture) traveled down similar paths. The microprocessor families from both companies developed bigger and faster products: Intel 8085, 8088, 80x86, . . . and the Motorola 6809, 68000, 680x0. . . . During the early 1980's another technology emerged—the microcontroller. In sharp contrast to the microprocessor family, which optimized computational speed and memory size at the expense of power and physical size, the microcontroller devices minimized power consumption and physical size, striving for only modest increases in computational speed and memory size. Out of the Intel architecture came the 8051 family ([www.semiconductors.philips.com](http://www.semiconductors.philips.com)), and out of the Motorola architecture came the 6805, 6811, and 6812 microcontroller family ([www.freescale.com](http://www.freescale.com)). Many of the same fundamental differences that existed between the original 8-bit Intel 8080 and Motorola 6800 have persisted during forty years of microprocessor and microcontroller developments. In 1999, Motorola shipped its 2 billionth MC68HC05 microcontroller. In 2004, Motorola spun off its microcontroller products as Freescale Semiconductor, and remained No. 2 in market share in microcontrollers overall and No. 1 in market share in microcontrollers for automotive applications (Gartner Dataquest). Microchip is the No. 1 supplier of 8-bit microcontrollers.

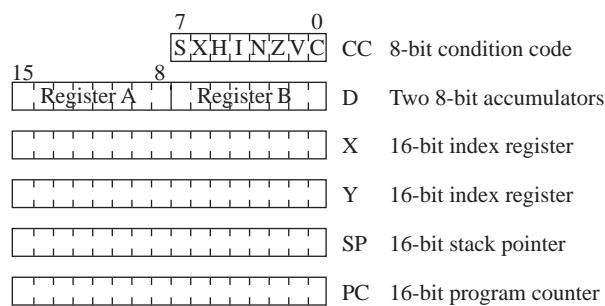
In this section, common features of the 9S12 are presented. In subsequent sections, information specific to each processor is presented.

## 1.6.1 Registers

The 9S12 registers are depicted in Figure 1.28. Registers A and B concatenated together form a 16-bit accumulator, Register D, with Register A containing the most significant byte. Typically Registers A and B contain data (numbers) whereas Registers X and Y contain addresses (pointers.) On the 9S12, the SP (stack pointer) points to the top element of the stack. Register PC (program counter) points to the current instruction.

**Figure 1.28**

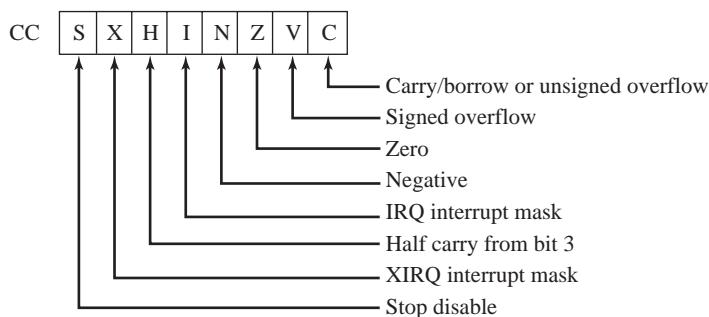
The 9S12 has six registers.



The condition code bits are shown in Figure 1.29. The N, Z, V, and C bits signify the status of the previous ALU operation. Many instructions set these bits to signify the result of the operation. When S=1, the stop instruction is disabled. When X=0, XIRQ interrupts are allowed. Once X is set to zero, the software cannot set it back to 1. The H bit is used for binary coded decimal (BCD) addition. When I=0, IRQ interrupts are enabled. Interrupts are discussed in Chapter 4.

**Figure 1.29**

The 9S12 condition code bits.



**Checkpoint 1.30:** To prevent (disable) interrupts, what value should the I bit be?

**Checkpoint 1.31:** List all the registers that can hold 16-bit addresses.

## 1.6.2 Terminology

This chapter focuses on the 9S12 architecture, but now we introduce a few simple instructions in order to understand how the microcomputer works. When describing the action of an assembly instruction we will use the following notation.

```
w is a signed 8-bit -128 to +127 or unsigned 8-bit 0 to 255
n is a signed 8-bit -128 to +127
u is a unsigned 8-bit 0 to 255
W is a signed 16-bit -32787 to +32767 or unsigned 16-bit 0 to 65535
N is a signed 16-bit -32787 to +32767
U is a unsigned 16-bit 0 to 65535
=[addr] specifies an 8-bit read from addr
={addr} specifies a 16-bit read from addr using "big endian"
=<addr> specifies a 32-bit read from addr using "big endian"
[addr]= specifies an 8-bit write to addr
{addr}= specifies a 16-bit write to addr using "big endian"
<addr>= specifies a 32-bit write to addr using "big endian"
```

Assembly language instructions have four fields. The *label field* is optional and starts in the first column; it is used to identify the position in memory of the current instruction. You must choose a unique name for each label. The *opcode field* specifies the microcomputer

command to execute. The *operand field* specifies where to find the data to execute the instruction. We will see that opcodes have 0, 1, 2, or 3 operands. The *comment field* is also optional and is ignored by the computer, but it allows you to describe the software making it easier to understand. Good programmers add comments to explain the software. The `ldaa` instruction reads 8 bits of data from memory and places them in register A. The `staa` instruction stores the 8-bit value from register A into memory. The `ldx` instruction reads 16 bits of data from memory and places them in register X. The `stx` instruction stores the 16-bit value from register X into memory. The first two assembly instructions copy the contents of Port A (location 0 on the 9S12) to memory location \$3800. The next two assembly instructions move a 16-bit value from locations \$3802–\$3803 into locations \$3804–\$3805.

label	opcode	operand	comment
here	<code>ldaa</code>	\$0000	;RegA=[ \$0000 ]
	<code>staa</code>	\$3800	; [ \$3800 ]=RegA
	<code>ldx</code>	\$3802	;RegX={ 3802 }
	<code>stx</code>	\$3804	; { 3804 }=RegX

As we will learn in further along in the book, it is much better to add comments to explain how—or even better, why—we do the action. But for now we are learning what the instruction is doing, so in this chapter, comments will describe what the instruction does. The assembly language instructions (like the forgoing example) are translated into machine instructions. The `ldaa $0000` instruction is translated into 2 bytes of machine code:

Object code	instruction	comment
\$96 \$00	<code>ldaa \$0000</code>	;RegA=[ \$0000 ]

### 1.6.3 Addressing Modes

A fundamental issue in program development is the differentiation between data and address. It is in assembly language programming in general and addressing modes in specific that this differentiation becomes clear. When we put the number 1000 into register X, whether this is data or address depends on how the 1000 is used. Most instructions access memory to fetch parameters or save results. The addressing mode is the format the instruction uses to specify the memory location to read or write data. All instructions begin by fetching the machine instruction (opcode and operand) pointed to by the PC. Some instructions operate completely within the processor and require no memory data fetches. These instructions have no operand and are classified as *inherent*. For example, the `clra` instruction places a zero into register A. If the data is found in the instruction itself, the instruction uses *immediate* addressing mode. If the instruction uses the absolute address to specify the memory data location, the instruction uses either *direct* or *extended* addressing mode. Notice in Table 1.10 that the `ldaa` instruction can be used with the immediate, direct, and extended addressing modes. In particular, the `ldaa` instruction means “load into register A” and the addressing mode specifies the source of the data to be used. Many computers, including the 9S12, use *PC-relative* addressing mode to encode branch instructions. PC-relative addressing makes the object code smaller and relocatable. Normally, the computer executes one instruction after another as they are listed in memory, except the branch instructions, which cause the program to jump to another place. For example, the `bra $F042` instruction is an unconditional branch, causing the program to jump to location

**Table 1.10**  
Simple addressing modes.

9S12 code	opcode	operand	comment
\$87	<code>clra</code>		;Reg A = 0 (inherent)
\$86 24	<code>ldaa</code>	#36	;Reg A = \$24 (immediate)
\$96 24	<code>ldaa</code>	36	;Reg A = [ \$0024 ] (direct)
\$B6 08 01	<code>ldaa</code>	\$0801	;Reg A = [ \$0801 ] (extended)
\$20 \$40	<code>bra</code>	\$F042	;Jump to \$F042

\$F042. The addressing mode is called PC-relative because the machine code contains the address difference between where the program is now and the address to which the program will jump. There are many more addressing modes, but for now, these five addressing modes, as illustrated in Table 1.10, are enough to get us started.

**Checkpoint 1.32:** What is the addressing mode used for?

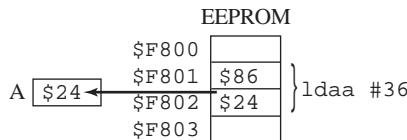
In this section, these five addressing modes are introduced. These simple addressing modes are sufficient to understand most of the software presented in this book. The more complicated addressing modes are presented in Chapter 2.

**1. Inherent** addressing mode has no operand field. Sometimes there is no data for the instruction at all. For example, the `stop` instruction halts execution. Sometimes the data for the instruction is implied. For example, the `clrB` instruction sets register B to zero. In this case, the data value of zero is implied. On the other hand, sometimes the data must be fetched from memory, but the address of the data is implied. For example, the `pula` instruction will pop an 8-bit data from the stack and store it in register A. In particular, the data value pointed to by the SP is read from memory and stored into register A.

**2. Immediate** addressing mode uses a fixed data constant. The data itself is included in the machine code. For example, the `ldaa #36` instruction will store a data value of 36 into register A (Figure 1.30). Notice that the “36” itself is encoded in the machine code for the `ldaa #36` instruction. In assembly code, this mode is signified by the # sign.

**Figure 1.30**

Example of the immediate addressing mode.



**Observation:** With immediate mode addressing, the information is stored in the machine code.

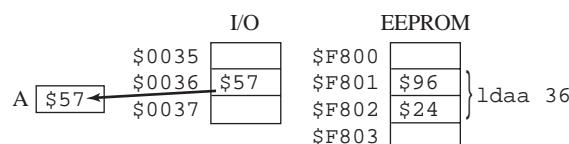
**Common error:** It is illegal to use the immediate addressing mode with instructions that store data into memory (e.g., `staa`).

**Checkpoint 1.33:** What is the difference between `ldaa #36` and `ldaa #$24`?

**3. Direct-page** addressing mode uses an 8-bit address to access from addresses 0 to \$00FF. In many computer systems outside the Freescale family, this addressing mode is called *zero-page*. On the 9S12, they reference the I/O ports. In assembly language, the < operator forces direct addressing. Figure 1.31 illustrates the execution of the `ldaa 36` instruction.

**Figure 1.31**

Example of the direct-page addressing mode.

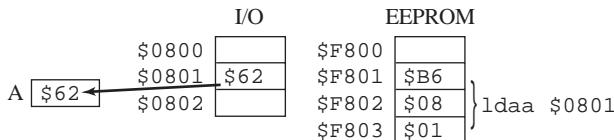


**Observation:** With direct and extended mode addressing, a fixed pointer to the information is stored in the machine code. The data itself may change dynamically, but its location is fixed.

**Checkpoint 1.34:** What is the difference between `ldaa #36` and `ldaa 36`?

**4. Extended addressing** mode uses a 16-bit address to access all memory and I/O devices. In many computer systems outside the Freescale family this addressing mode is called *direct*, because it can directly access all of memory. In this book, we will adhere to the Freescale terminology (direct means 8-bit addressing and extended means 16-bit addressing). The `>` operator forces extended addressing. In general, when the address happens to fall in the 0 to \$0OFF range, the assembler will automatically use direct addressing, otherwise it uses extended addressing. Figure 1.32 illustrates the execution of the `ldaa $0801` instruction.

**Figure 1.32**  
Example of the extended addressing mode.



**Common error:** It is wrong to assume the `<` and `>` operators affect the amount of data that is transferred. The `<` and `>` operators will affect the addressing mode; that is how the address is represented.

**Observation:** Accesses to Registers A, B, and CC transfer 8 bits, whereas accesses to Registers D, X, Y, SP, and PC transfer 16 bits regardless of the addressing mode.

**Checkpoint 1.35:** What is the difference between `ldx #$0801` and `ldx $0801`?

**Checkpoint 1.36:** What is the difference between the instruction `ldaa $12` and the instruction `ldax $12`?

**5. PC-relative** addressing mode is used for the branch and branch to subroutine instructions. Stored in the machine code is not the absolute address of where to branch, but the 8-bit signed offset relative distance from the current PC value. When the branch address is being calculated, the PC already points to the next instruction. Calculating relative offsets gives first-time programmers a lot of trouble, but lucky for us the assembler calculates it for us. It is explained here in order to better understand how the computer works, rather than being necessary for us to do while programming. The address of the next instruction is (location of instruction)+(number of bytes in the machine code). In the following example, assume the branch instruction is located at address \$F880. The destination address is before the current instruction, which is called a backward jump.

`bra $F840`

The operand field for PC relative addressing is an 8-bit value called **rr**, which is calculated using the equation: (destination address)–(location of instruction)–(size of the instruction). Since the `bra` op code is one byte (\$20) and the operand is one byte, this instruction requires two bytes and the **rr** field is

$$\$F840 - \$F880 - 2 = -\$42 = \$BE$$

and the object code for this instruction will be \$20BE. Again assume the branch instruction is located at address \$F880. This time the destination address is after the current instruction, which is called a forward jump.

`bra $F8C8`

The **rr** field is

$$\$F8C8 - \$F880 - 2 = \$46$$

and the object code for this instruction will be \$2046.

**Common error:** Since not every instruction supports every addressing mode, it would be a mistake to use an addressing mode not available for that instruction.

**Observation:** Some of the conditional branch instructions on the 9S12 require a different number of cycles to execute depending on whether or not the branch is taken. The cycle time when accessing external memory on a 9S12 depends on the speed of the external memory. It also depends on whether the address is an even number or an odd number. These facts complicate the task of predetermining how long a 9S12 program will take to execute.

**Observation:** Relative addressing within a program block is essential for implementing relocatable code.

### 1.6.4 Numbering Scheme Used by Freescale

The numbering scheme used by Freescale allows you to quickly determine the type of ROM used as the main program memory in the device, as shown in Table 1.11. In most cases, the value at the end of the part number specifies the size of the program memory (e.g., MC68HC711E20 is 20K and MC9S12C32 is 32K), but sometimes for the 6811 it does not (e.g., MC68HC11D3 is 4K and MC68HC711E9 is 12K). The letter (or letters) placed after the 11 or 12 and before the final number specifies the series. For example, the MC68HC711E9 belongs to the “E” series. Please note that most Freescale microcontrollers have two or three memory modules, and the numbering scheme refers only to the main program memory. For example, according to the numbering scheme the MC9S12DP512 has 512K bytes flash EEPROM, but it also has 4K bytes of regular EEPROM and 12K bytes of RAM.

**Table 1.11**  
Freescale uses a numbering scheme to specify the type of main memory.

Number	Main Memory	Examples
None	ROM	MC68HC11E9 MC68HC12D60
7	EPROM	MC68HC711E9 MC68HC711D3
8	EEPROM	MC68HC811E2 MC68HC812A4
9	Flash EEPROM	MC68HC912B32 MC9S12C32

**Checkpoint 1.37:** The MC9S12A64 has how much and what type of main memory?

## 1.7 9S12 Architecture Details

The microcontrollers in the 9S12 family differ by the amount of memory and by the types of I/O modules. All 9S12 microcontrollers have a 16-bit central processing unit (HCS12 or HCS12X), SIM (a system integration module), RAM (volatile random access memory), Flash EEPROM (nonvolatile electrically erasable programmable read-only memory), and a PLL (phase-locked loop). The 9S12 microcontrollers are configured with zero, one, or more of the following modules: asynchronous serial communications interface (SCI), serial peripheral interface (SPI), inter-integrated circuit ( $I^2C$ ), key wakeup, 16-bit timer, a pulse-width modulation (PWM), 10-bit or 12-bit analog-to-digital converter (ADC), 8-bit digital-to-analog converter (DAC), liquid-crystal display driver (LCD), controller area network (CAN 2.0), universal serial bus (USB 2.0) interface, Ethernet (MAC FEC 10/100) interface, memory expansion logic, and XGate. The PLL allows the software to increase or decrease the execution speed. XGate is an I/O coprocessor and is appropriate for systems requiring high-speed I/O. Typically, the SPI and  $I^2C$  modules allow the 9S12 to communicate with other dedicated peripheral devices, whereas typically, the SCI, CAN, USB, and Ethernet allow the 9S12 to communicate with other computers. The ADC module (together with sensors and analog amplifiers) can be used to collect information. The PWM module can be used to control delivered

power to motors and lights. The PWM also can be used as an analog output. We can use key wake-up modules to interface digital inputs to the 9S12. Currently, the 9S12 family consists of a large number of devices with a wide range of memory and I/O configurations, only some of which are shown in Table 1.12. This book will focus on the MC9S12C32 and MC9S12DP512, because they are low-cost full-featured devices with minimal memory, making them ideally suitable for educational use. The fundamental concepts learned in this book can be adapted easily to other members of the Freescale microcontroller family.

Series	ROM	RAM	I/O pins	Features
MC9S12A	32 to 512	2 to 12	59 to 91	I <sup>2</sup> C, SCI, SPI, ADC
MC9S12B	64 to 128	2 to 4	59 to 91	Automotive/Industrial with CAN
MC9S12C	32 to 128	2 to 4	31 to 60	CAN, SCI, SPI, ADC
MC9S12D	32 to 512	4 to 12	59 to 91	Automotive/Industrial with CAN
MC9S12E	32 to 128	2 to 16	58 to 90	General purpose, 3 V with D/A
MC9S12GC	16 to 128	1 to 4	31 to 60	Low-cost, Low-pin count
MC9S12H	128 to 256	6 to 12	83 to 117	LCD/H-Bridge drivers with CAN
MC9S12HZ	64 to 256	4 to 12	58 to 85	I <sup>2</sup> C, SCI, SPI, ADC, LCD
MC9S12NE	64	8	80	Ethernet, I <sup>2</sup> C
MC9S12UF	32	3 to 5	75 to 77	30 MHz, USB, SCI
MC9S12XA	128 to 256	11 to 32	59 to 119	XGate, 40 MHz, I <sup>2</sup> C, ADC, SCI
MC9S12XD	64 to 512	4 to 32	59 to 119	XGate, 40 MHz, I <sup>2</sup> C, CAN, ADC
MC9S12XE	128 to 1000	12 to 64	59 to 152	XGate, 40 MHz, SCI, SPI, I <sup>2</sup> C, ADC
MC9S12XS	64 to 256	4 to 12	44 to 91	XGate, 40 MHz, CAN, SCI, SPI, ADC

**Table 1.12**

The 9S12 family includes many series; memory is defined in kibibytes.

### 1.7.1 9S12C32 Architecture

The 9S12C32, with its port structure shown in Figure 1.33, is one of the smaller and lower-cost members of the 9S12 family. It has 2 kibibytes of RAM and 32 kibibytes of EEPROM. Modules include an asynchronous serial communications interface (SCI) module; a serial peripheral interface (SPI) module; an 8-channel, 16-bit timer module; a pulse-width modulation (PWM) module; an 8-channel, 10-bit analog-to-digital converter (ATD); a 1 Mbps controller area network (CAN 2.0) module; memory expansion logic; and a phase-locked loop (PLL). The PLL allows the software to increase or decrease the execution speed. The PWM module can utilize either Port P or Port T as selected by the MODRR register. Ports J and P also support keypad wake-up interrupts.

The address space of the input/output devices and the RAM can be mapped on any 2-KiB boundary by software. In this book, we will use the address map shown in Table 1.13 for the single-chip MC9S12C32.

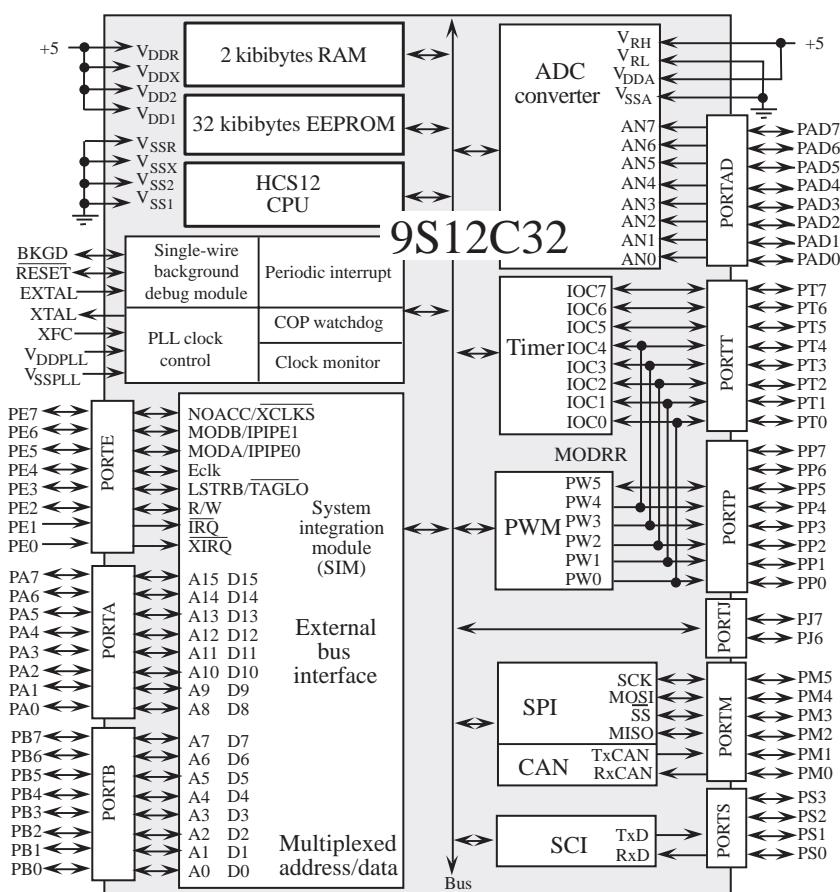
The 9S12C32 is available in three quad flat package (QFP) sizes. The larger chip packages have more pins, as shown in Table 1.14.

One of the smallest systems based on the 9S12 family is the 24-pin Nanocore module from TechArts, as shown in Figure 1.34. This system includes a voltage regulator, a run/load switch, a BDM header, RS232 drivers for the SCI port, the eight pins of Ports T, and the eight pins of Port AD.

Program 1.1 and Table 1.15 define some of the parallel ports for the 9S12C32. A full list of I/O ports can be found in the reference manual at <http://users.ece.utexas.edu/~valvano/Datasheets>. A *pseudo-operation code* is a command to the assembler and usually does not create machine executable code. Other names for pseudo-operation code are *pseudo-op* and *assembly directive*. The *equ* is a pseudo-op that creates a mapping from a

**Figure 1.33**

Block diagram of a Freescale 9S12C32.



Address	Size	Device	Device	Contents
\$0000 to \$03FF	1K	I/O	Input/output devices	
\$3800 to \$3FFF	2K	RAM	Random access memory	Variables and stack
\$4000 to \$7FFF	16K	EEPROM	Electrically erasable PROM	Programs and fixed constants
\$C000 to \$FFFF	16K	EEPROM	Electrically erasable PROM	Programs and fixed constants

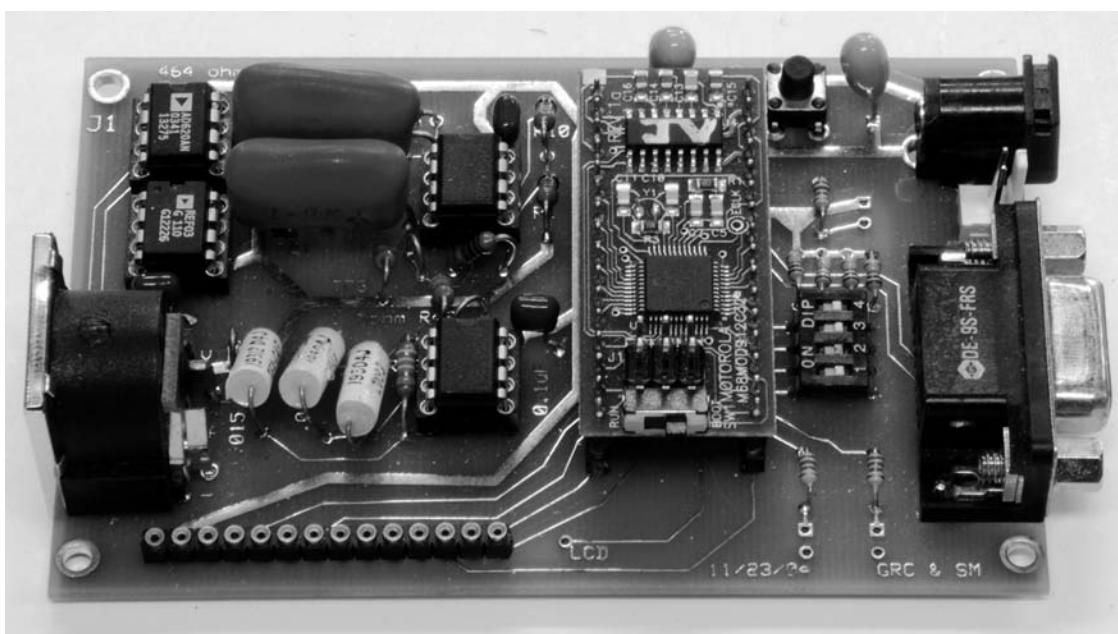
**Table 1.13**

The MC9S12C32 has 32K of EEPROM and 2K bytes of RAM.

Port	48-pin	52-pin	80-pin	Shared Functions
Port A	PA0	PA2-PA0	PA7-PA0	Address/data bus
Port B	PB4	PB4	PB7-PB0	Address/data bus
Port E	PE7, PE4, PE1, PE0	PE7, PE4, PE1, PE0	PE7-PE0	System integration module
Port J	—	—	PJ7, PJ6	Key wake-up
Port M	PM5-PM0	PM5-PM0	PM5-PM0	SPI, CAN
Port P	PP5	PP5-PP3	PP7-PP0	Key wake-up, PWM
Port S	PS1-PS0	PS1-PS0	PS3-PS0	SCI
Port T	PT7-PT0	PT7-PT0	PT7-PT0	Timer, PWM
Port AD	PAD7-PAD0	PAD7-PAD0	PAD7-PAD0	Analog-to-digital converter

**Table 1.14**

The 9S12C32 has nine external I/O ports.



Courtesy of Jonathan Valkano

**Figure 1.34**

The TechArts NC12C32 Nanocore used to build an EKG monitor (see Figure 12.49).

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
\$0241	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTIT
\$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
\$0250	0	0	PM5	PM4	PM3	PM2	PM1	PM0	PTM
\$0251	0	0	PM5	PM4	PM3	PM2	PM1	PM0	PTIM
\$0252	0	0	DDRM5	DDRM4	DDRM3	DDRM2	DDRM1	DDRM0	DDRM
\$008D	Bit 7	6	5	4	3	2	1	Bit 0	ATDDIEN
\$0270	PAD7	PAD6	PAD5	PAD4	PAD3	PAD2	PAD1	PAD0	PTAD
\$0271	PAD7	PAD6	PAD5	PAD4	PAD3	PAD2	PAD1	PAD0	PTIAD
\$0272	DDRAD7	DDRAD6	DDRAD5	DDRAD4	DDRAD3	DDRAD2	DDRAD1	DDRAD0	DDRAD

**Table 1.15**

Some 9S12C32 Parallel ports.

symbol and a value. Given the code in Program 1.1 on page 40, the instructions `staa $0240` and `staa PTT`, produce the exact same machine code; therefore, they perform the exact same function when executed. We prefer `staa PTT`, because it is easier to understand. We clear (0) a bit in the direction register to make that pin an input, and set it (1) to make it an output. Notice that Port M is only six bits wide. A pin on Port AD (PTAD) can be used as a digital input if the corresponding bit in the ATDDIEN is set to 1 and the bit in the DDRAD is cleared to 0. A pin on Port AD (PTAD) can be used as a digital output if the corresponding bit in the DDRAD is set to 1. The pins on Port AD can be used as analog input if the ADC is enabled and the corresponding bit in the ATDDIEN is cleared to 0.

## 1.7.2 9S12DP512 Architecture

Figure 1.35 shows the port structure of the 9S12DP512. Although the 9S12DP512 has 512 KiB EEPROM, only 48 kibibytes of it is directly addressable using standard 16-bit addressing modes. The remaining 464 KiB must be accessed using the paged memory process.

<i>; port name definitions</i>	<i>#define _P(n) *(unsigned char volatile *) (n)</i>
ATDDIEN equ \$008D ; Input Enable	<i>#define ATDDIEN _P(0x008D)</i>
DDRAD equ \$0272 ; Direction	<i>#define DDRAD _P(0x0272)</i>
DDRM equ \$0252 ; Direction	<i>#define DDRM _P(0x0252)</i>
DDRT equ \$0242 ; Direction	<i>#define DDRT _P(0x0242)</i>
PTAD equ \$0270 ; I/O	<i>#define PTAD _P(0x0270)</i>
PTIAD equ \$0271 ; Input	<i>#define PTIAD _P(0x0271)</i>
PTM equ \$0250 ; I/O	<i>#define PTM _P(0x0250)</i>
PTIM equ \$0251 ; Input	<i>#define PTIM _P(0x0251)</i>
PTT equ \$0240 ; I/O	<i>#define PTT _P(0x0240)</i>
PTIT equ \$0241 ; Input	<i>#define PTIT _P(0x0241)</i>

### Program 1.1

Definitions of some of the 9S12C32 I/O ports.

**Figure 1.35**

Block diagram of a Freescale 9S12DP512.

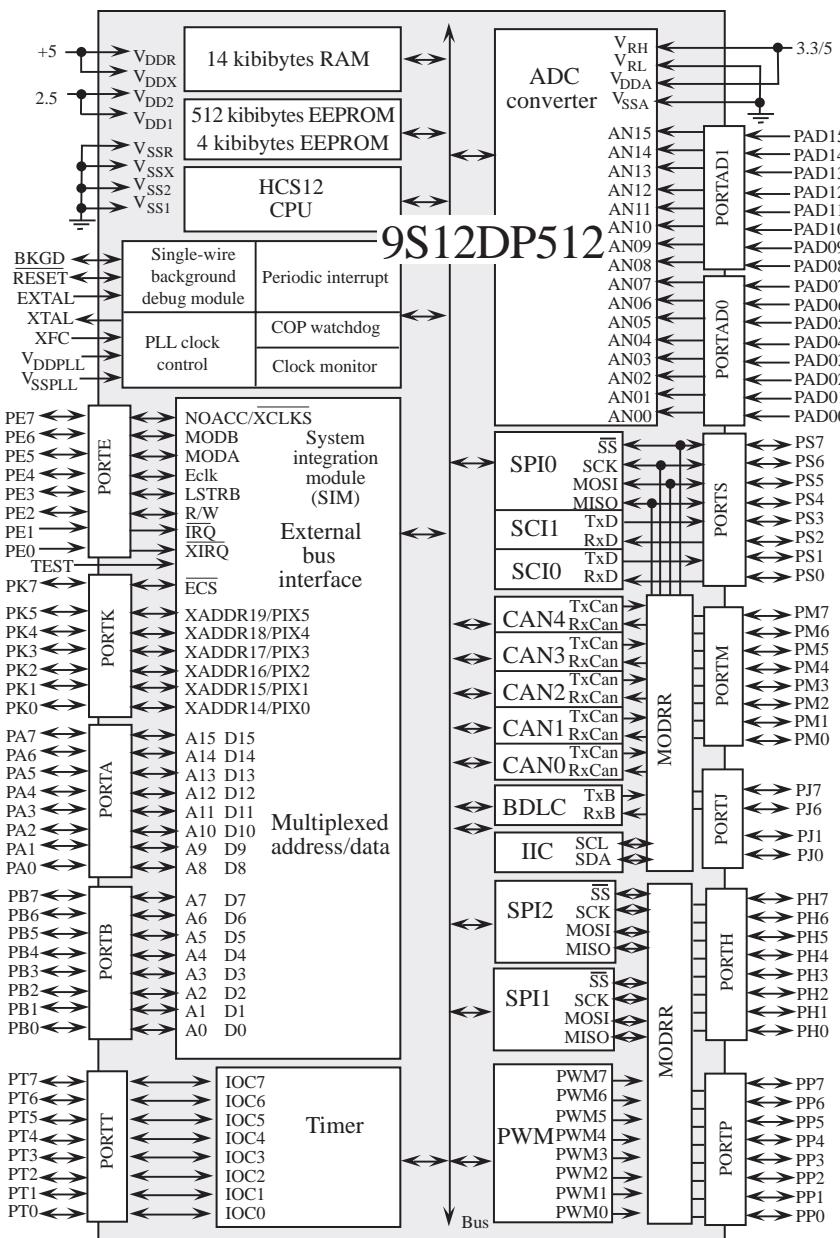


Table 1.16 shows the I/O pins that exist on the 9S12DP512 chip. Other versions of the D-family, such as DG DJ DT, are at a lower cost than the DP512, because they have less memory, fewer CAN, fewer SPI, and no J1850 ports. All the DP512 examples in this book will apply to these other members of the D-family. All pins except PE5 and PE6 are available on the TechArts Adapt module shown in Figure 1.36.

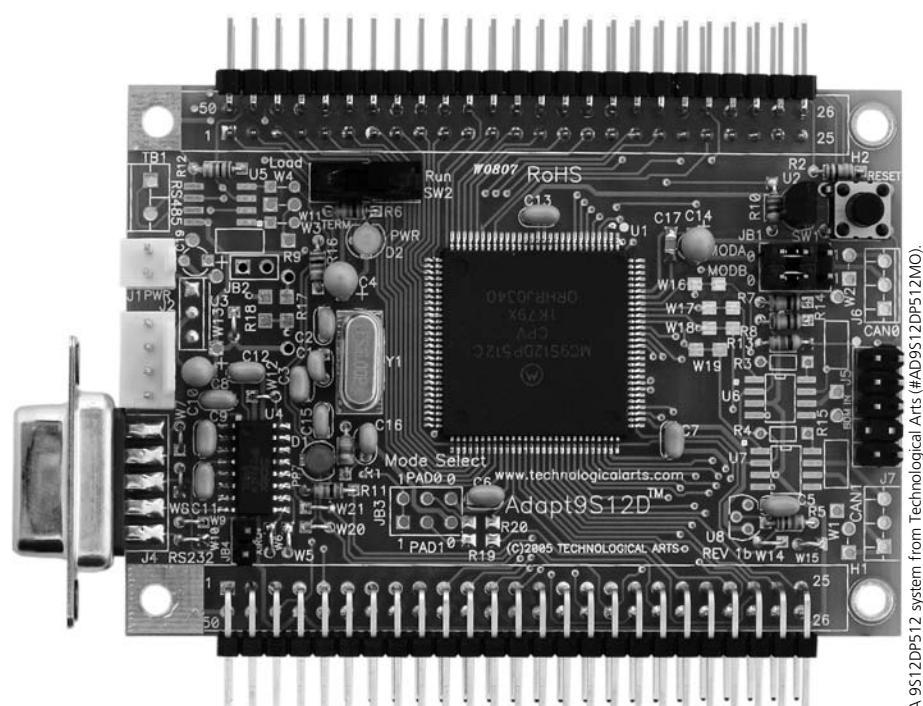
**Table 1.16**

The 9S12DP512 is a 112 pin module with 91 I/O pins

Chip	Special I/O (Priority Order)	Standard I/O
PAD15-PAD8	ADC	Digital input
PAD7-PAD0	ADC	Digital input
PA7-PA0		Digital I/O
PB7-PB0		Digital I/O
PE7-PE0	IRQ and XIRQ	Digital I/O
PH7-PH4	SPI2	Digital I/O and key wake-up
PH3-PH0	SPI1	Digital I/O and key wake-up
PJ7-PJ6	CAN4 I <sup>2</sup> C or CAN0	Digital I/O and key wake-up
PJ1-PJ0		Digital I/O and key wake-up
PK7, PK5-PK0		Digital I/O
PM7-PM6	CAN3 or CAN4	Digital I/O
PM5-PM4	CAN2 CAN0 CAN4 or SPI0	Digital I/O
PM3-PM2	CAN1 CAN0 or SPI0	Digital I/O
PM1-PM0	CAN0 or BDLC	Digital I/O
PP7-PP4	PWM or SPI2	Digital I/O and key wake-up
PP3-PP0	PWM or SPI1	Digital I/O and key wake-up
PS7-PS4	SPI0	Digital I/O
PS3-PS2	SCI1	Digital I/O
PS1-PS0	SCI0	Digital I/O
PT7-PT0	Input capture or output compare	Digital I/O

**Figure 1.36**

A 9S12DP512 system from Technological Arts (#AD9S12DP512M0).



A 9S12DP512 system from Technological Arts (#AD9S12DP512M0).

The 9S12DP512 has 91 I/O pins, some of which are listed in Program 1.2. Many of the pins can be configured to implement complex I/O functions, but in this section, the pins are used as simple digital inputs or outputs. We clear (0) a bit in the direction register to make that pin an input, and set it (1) to make it an output. Program 1.2 defines four of these parallel ports. A full list of I/O ports can be found in the reference manual at <http://users.ece.utexas.edu/~valvano/Datasheets/>. Table 1.17 shows all 91 digital I/O pins that can be used on the 9S12DP512. Pins PE1, PE0, and Ports PORTAD1, PORTAD0 are input only. The module routine register (MODRR) will be explained later, when it is needed. In C, we add the `volatile`, so the compiler will not optimize the code involving I/O ports. More precisely, it tells the compiler the value may change beyond the control of the software itself. In particular, each time the value is needed, it will be reread from the port.

<code>; port name definitions</code>	
<code>DDRH equ \$026A ; Direction</code>	<code>#define _P(n) *(unsigned char volatile *)(n)</code>
<code>DDRJ equ \$0262 ; Direction</code>	<code>#define DDRH _P(0x0262)</code>
<code>DDRM equ \$0252 ; Direction</code>	<code>#define DDRJ _P(0x026A)</code>
<code>DDRP equ \$025A ; Direction</code>	<code>#define DDRM _P(0x0252)</code>
<code>DDRT equ \$0242 ; Direction</code>	<code>#define DDRP _P(0x025A)</code>
<code>PTH equ \$0260 ; I/O</code>	<code>#define DDRT _P(0x0242)</code>
<code>PTIH equ \$0261 ; Input</code>	<code>#define PTH _P(0x0260)</code>
<code>PTJ equ \$0268 ; I/O</code>	<code>#define PTIH _P(0x0261)</code>
<code>PTIJ equ \$0269 ; Input</code>	<code>#define PTJ _P(0x0268)</code>
<code>PTM equ \$0250 ; I/O</code>	<code>#define PTIJ _P(0x0269)</code>
<code>PTIM equ \$0251 ; Input</code>	<code>#define PTM _P(0x0250)</code>
<code>PTP equ \$0258 ; I/O</code>	<code>#define PTIM _P(0x0251)</code>
<code>PTIP equ \$0259 ; Input</code>	<code>#define PTP _P(0x0258)</code>
<code>PTT equ \$0240 ; I/O</code>	<code>#define PTIP _P(0x0259)</code>
<code>PTIT equ \$0241 ; Input</code>	<code>#define PTT _P(0x0240)</code>
	<code>#define PTIT _P(0x0241)</code>

### Program 1.2

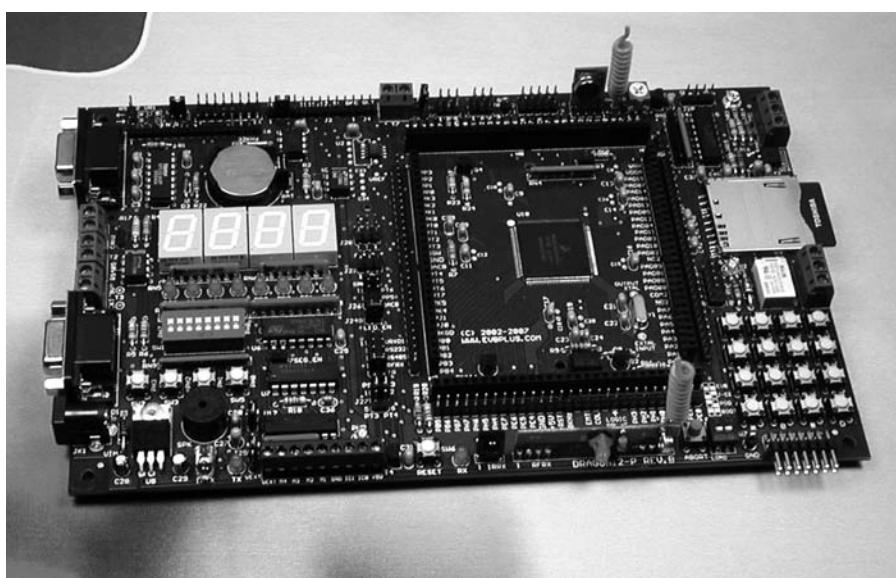
Definitions of some of the 9S12DP512 I/O ports.

Figure 1.37 shows the Dragon12 board from Wytec (<http://www.evbplus.com/index.html>). This board provides an integrated approach to teaching embedded systems where most of the I/O devices are included on the board itself. This board includes the 9S12DG512 and the following additional components: 16 by 2 LCD display module with LED backlight module, 4-digit, multiplexed 7-segment display module, 4 by 4 matrix keypad, eight LEDs connected, an 8-position DIP switch, four pushbutton switches, IR transceiver with built-in 38 kHz oscillator, RS485 communication port with terminal block, a speaker driven by timer (or PWM or DAC for alarm), voice and music applications, a potentiometer trimmer pot for analog input, dual SCIs with DB9 connectors, dual 10-bit DAC for testing SPI interface and generating analog waveforms, I<sup>2</sup>C-based Real Time Clock DS1307 with backup battery, dual H-Bridge with motor feedback or incremental encoder interface, four robot servo outputs, an opto-coupler output, a DPDT form C relay, a temperature sensor, a light sensor, a logic probe with LED indicator, a solderless breadboard, and a DB9 RS232 cable for connecting to a PC serial port. The Dragon 12 is an excellent platform to teach embedded systems. This board is very cost effective, much cheaper than purchasing the components individually. If embedded systems is taught in more than one class, this board supports both a simple introduction to embedded systems as well as a more sophisticated embedded systems lab including CAN, I<sup>2</sup>C, motors, and servos.

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	PORTA
\$0001	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	PORTB
\$0002	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	DDRA
\$0003	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	DDRB
\$0008	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0	PORTE
\$0009	DDRE7	DDRE6	DDRE5	DDRE4	DDRE3	DDRE2	0	0	DDRE
\$0032	PK7	0	PK5	PK4	PK3	PK2	PK1	PK0	PORTK
\$0033	DDRK7	0	DDRK5	DDRK4	DDRK3	DDRK2	DDRK1	DDRK0	DDRK
\$008F	PAD07	PAD06	PAD05	PAD04	PAD03	PAD02	PAD01	PAD00	PORTAD0
\$012F	PAD15	PAD14	PAD13	PAD12	PAD11	PAD10	PAD09	PAD08	PORTAD1
\$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
\$0241	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTIT
\$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
\$0247	0	MODRR6	MODRR5	MODRR4	MODRR3	MODRR2	MODRR1	MODRR0	MODRR
\$0248	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTS
\$0249	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTIS
\$024A	DDRS7	DDRS6	DDRS5	DDRS4	DDRS3	DDRS2	DDRS1	DDRS0	DDRS
\$0250	PM7	PM6	PM5	PM4	PM3	PM2	PM1	PM0	PTM
\$0251	PM7	PM6	PM5	PM4	PM3	PM2	PM1	PM0	PTIM
\$0252	DDRM7	DDRM6	DDRM5	DDRM4	DDRM3	DDRM2	DDRM1	DDRM0	DDRM
\$0258	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0	PTP
\$0259	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0	PTIP
\$025A	DDRP7	DDRP6	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0	DDRP
\$0260	PH7	PH6	PH5	PH4	PH3	PH2	PH1	PH0	PTH
\$0261	PH7	PH6	PH5	PH4	PH3	PH2	PH1	PH0	PTIH
\$0262	DDRH7	DDRH6	DDRH5	DDRH4	DDRH3	DDRH2	DDRH1	DDRH0	DDRH
\$0268	PJ7	PJ6	0	0	0	0	PJ1	PJ0	PTJ
\$0269	PJ7	PJ6	0	0	0	0	PJ1	PJ0	PTIJ
\$026A	DDRJ7	DDRJ6	0	0	0	0	DDRJ1	DDRJ0	DDRJ

**Table 1.17**

9S12DP512 parallel ports.

**Figure 1.37**  
Dragon12-plus board  
from Wytec.

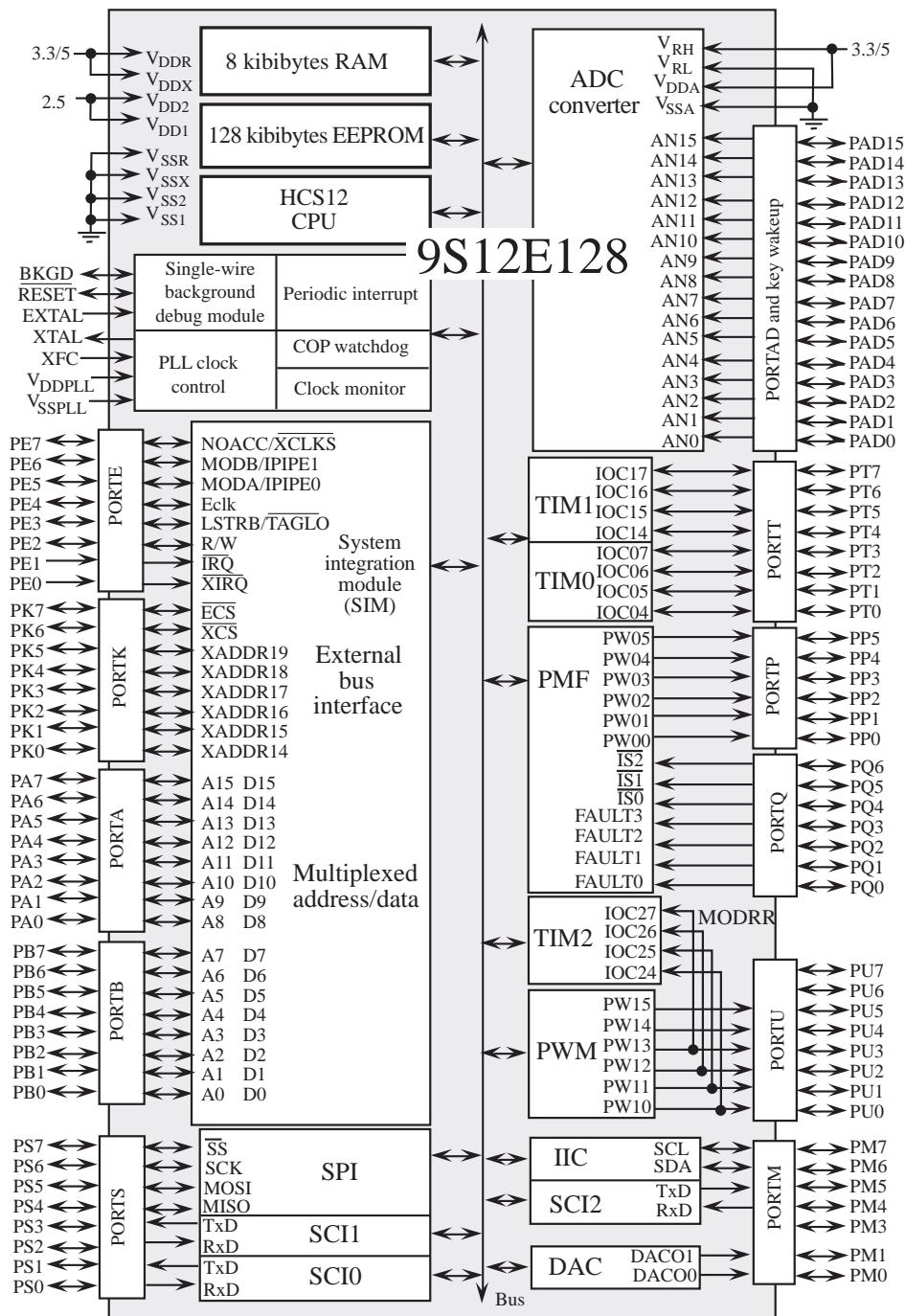
Dragon12-plus board from Wytec

### 1.7.3 9S12E128 Architecture

Figure 1.38 shows the port structure of the 9S12E128, which has 8 KiB of RAM and 128 KiB of flash EEPROM. This member of the 9S12 family has 12 input capture/output compare timer pins, 12 pulse-width modulated output pins, 16 ADC inputs, two DAC outputs, one SPI module three SCI modules, and one I<sup>2</sup>C interface. There are two sizes of the 9S12E128 chip: one with 80 pins and the other with 112 pins. The 112-pin chip has 92 I/O pins, some of which are listed in Table 1.18. We clear (0) a bit in the direction register to make that pin an input, and set it (1) to make it an output.

**Figure 1.38**

Block diagram of a Freescale 9S12E128.



Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	PORTA
\$0001	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	PORTB
\$0002	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	DDRA
\$0003	DDR87	DDR86	DDR85	DDR84	DDR83	DDR82	DDR81	DDR80	DDR8
\$0008	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0	PORTE
\$0009	DDRE7	DDRE6	DDRE5	DDRE4	DDRE3	DDRE2	0	0	DDRE
\$0032	PK7	PK6	PK5	PK4	PK3	PK2	PK1	PK0	PORTK
\$0033	DDR77	DDR76	DDR75	DDR74	DDR73	DDR72	DDR71	DDR70	DDR7
\$008C	Bit 15	14	13	12	11	10	9	Bit 8	ATDDIEN0
\$008D	Bit 7	6	5	4	3	2	1	Bit 0	ATDDIEN1
\$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
\$0241	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTIT
\$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
\$0248	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTS
\$0249	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTIS
\$024A	DDRS7	DDRS6	DDRS5	DDRS4	DDRS3	DDRS2	DDRS1	DDRS0	DDRS
\$0250	PM7	PM6	PM5	PM4	PM3	0	PM1	PM0	PTM
\$0251	PM7	PM6	PM5	PM4	PM3	0	PM1	PM0	PTIM
\$0252	DDRM7	DDRM6	DDRM5	DDRM4	DDRM3	0	DDRM1	DDRM0	DDRM
\$0258	0	0	PP5	PP4	PP3	PP2	PP1	PP0	PTP
\$0259	0	0	PP5	PP4	PP3	PP2	PP1	PP0	PTIP
\$025A	0	0	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0	DDRP
\$0260	0	PQ6	PQ5	PQ4	PQ3	PQ2	PQ1	PQ0	PTQ
\$0261	0	PQ6	PQ5	PQ4	PQ3	PQ2	PQ1	PQ0	PTIQ
\$0262	0	DDRQ6	DDRQ5	DDRQ4	DDRQ3	DDRQ2	DDRQ1	DDRQ0	DDRQ
\$0268	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	PTU
\$0269	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	PTIU
\$026A	DDRU7	DDRU6	DDRU5	DDRU4	DDRU3	DDRU2	DDRU1	DDRU0	DDRU
\$0270	PAD15	PAD14	PAD13	PAD12	PAD11	PAD10	PAD9	PAD8	PTADH
\$0271	PAD07	PAD06	PAD05	PAD04	PAD03	PAD02	PAD01	PAD00	PTADL
\$0274	DDRAD15	DDRAD14	DDRAD13	DDRAD12	DDRAD11	DDRAD10	DDRAD09	DDRAD08	DDRADH
\$0275	DDRAD07	DDRAD06	DDRAD05	DDRAD04	DDRAD03	DDRAD02	DDRAD01	DDRAD00	DDRADL

**Table 1.18**

9S12E128 Parallel ports.

A pin on Port AD (PTAD) can be used as a digital input if the corresponding bit in the ATDDIEN is set to 1 and the bit in the DDRAD is cleared to 0. A pin on Port AD (PTAD) can be used as a digital output if the corresponding bit in the DDRAD is set to 1. The pins on Port AD can be used as analog input if the ADC is enabled and the corresponding bit in the ATDDIEN is cleared to 0.

### 1.7.4 Operating Modes

The 9S12 can operate in one of eight modes, where the mode is selected by the values of the three signals BKGD, MODA, and MODB that exist when the device starts up after a reset. In this book, we will only study the three modes shown in Table 1.19. In *single chip mode*, the 9S12 contains the four major building blocks required to make a complete computer system: processor, I/O, RAM, and EEPROM. For most of the book, we will be using single chip mode, where all ports are available for input/output. Because the 9S12 family is available with flash EEPROM ranging from 32 to 512 KiB, most embedded projects can be developed directly in single chip mode. On the other hand, the 9S12 family RAM sizes only range from 2 to 32 KiB. In Chapter 9, we will use the two expanded modes to interface external devices to Ports A, B, and E. For embedded

BKGD	MODB	MODA	Mode	Port A	Port B
1	0	0	Normal Single Chip	In/Out	In/Out
1	0	1	Normal Expanded Narrow	A15-8/D15-8/D7-0	A7-A0
1	1	1	Normal Expanded Wide	A15-8/D15-8	A7-0/D7-0

**Table 1.19**

The 9S12 has eight operating modes, but we will use normal single chip mode.

systems that require a large amount of read/write storage, we will use expanded modes to interface external RAM to the system. *Expanded narrow mode* creates a 16-bit address bus and 8-bit data bus, while *expanded wide mode* implements both the address bus and data bus as 16 bits.

We use flash EEPROM during development because it takes only minutes to perform an edit/assemble/download cycle. For delivered projects, we simply program our code into the flash EEPROM and embed the device into our final product. In single chip mode, the 9S12 implements a complete microcomputer, where all of its I/O ports are available. This mode is used for the final product with the application software programmed into the EEPROM.

## 1.8 Phase-Lock-Loop (PLL)

Normally, the execution speed of a microcontroller is determined by an external crystal. Some MC9S12C32 boards have an 8 MHz crystal creating a 4 MHz E clock. The MC9S12DP512 shown in Figure 1.36 has a 16 MHz crystal creating a 8 MHz E clock. The 9S12 has a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Program 1.3 will increase the E clock from 8 MHz to 24 MHz. The OSCCLK is the frequency of the crystal. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the E clock will require less power to operate and generate less heat. Speeding up the E clock obviously allows for more calculations per second.

```
; 9S12DP512 with a 16 MHz crystal
PLL_Init
    movb #$02,SYNR
    movb #$011,REFDV
; PLLCLK = 2*OSCCLK*(SYNR+1)/(REFDV+1)
    movb #$00,CLKSEL
    movb #$D1,PLLCTL      ; turn on PLL
    brclr CRGFLG,#$08,*  ; stabilized?
    bset CLKSEL,#$80      ; Switch to PLL
    rts
```

<sup>1</sup> Use 00 for the 9S12C32 or 9S12E128 with an 8 MHz crystal.

<sup>2</sup> Use 00 for the 9S12C32 or 9S12E128 with an 8 MHz crystal.

```
// 9S12DP512 with a 16 MHz crystal
void PLL_Init(void){
    SYNR = 0x02;
    REFDV = 0x012;
// PLLCLK = 2*OSCCLK*(SYNR+1)/(REFDV+1)
    CLKSEL = 0x00;
    PLLCTL = 0xD1; // turn on PLL
    while((CRGFLG&0x08) == 0){}
    CLKSEL |= 0x80; // Switch to PLL clock
}
```

### Program 1.3

These programs active the phase lock loop, setting the E clock to 24 MHz.

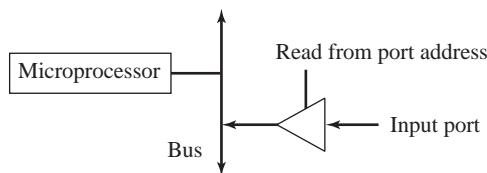
## 1.9 Parallel I/O Ports

### 1.9.1 Basic Concepts of Input and Output Ports

A *parallel I/O port* is a simple mechanism that allows the software to interact with external devices. It is called parallel because multiple signals can be accessed all at once. An input port, which allows the software to read external digital signals, is usually read only. That means a read cycle access from the port address returns the values existing on the inputs at that time. In particular, the tristate driver (triangle-shaped circuit in Figure 1.39) will drive the input signals onto the data bus during a read cycle from the port address. A write cycle access to an input port usually produces no effect. A simple fixed input port, such as pin PAD0 on the 9S12DP512 or pin PE0 on the MC9S12C32, behaves similarly to the circuit shown in Figure 1.39. The digital values existing on the input pins are copied into the microcomputer when the software executes a read from the port address.

**Figure 1.39**

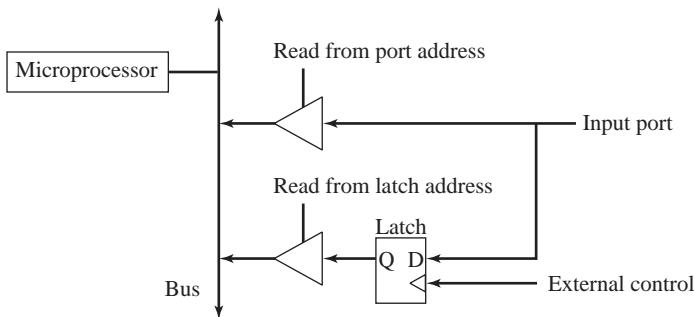
A read-only input port allows the software to read external digital signals.



A latched input port behaves similarly to the circuit shown in Figure 1.40. The digital values existing on the input pins are copied into an internal latch on an appropriate edge of the external control signal. At a later time, the data is transferred to the microcomputer when the software executes a read from the latch address. Notice that this latched input port also supports the regular input function. In other words, the software has the option of reading the port address to get information directly from the input port pins or from the latch address to get information that existed at the time of the previous active edge of the external control signal. None of the 9S12 pins can be latched.

**Figure 1.40**

A latched input port allows the software to read external digital signals that are captured via the external control signal.



**Checkpoint 1.38:** What happens if the software writes to an input port?

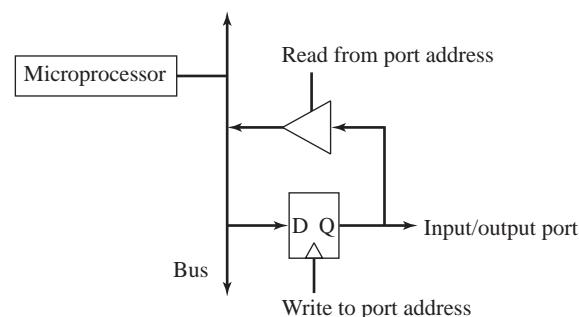
Often the latched input port allows the software to select the strategic edge to affect the latch function. In other words, the software specifies whether the rise or fall of an external control signal latches the input data. It is important to remember that when the software reads from the latch address, it obtains the values of the input signals occurring at the time of the active edge of the external control signal. In this way, the external device can provide the data at the input and issue an edge on the control signal latching the data into the computer;

the software can process the data at a later time without requiring the external device to maintain the valid data at the input. One of the limitations of this particular configuration is that it does not include a way for the software to signal back (acknowledge) to the external hardware that the previous data has been read by the software. In Chapter 3, we will develop more sophisticated handshaking protocols that provide for two-way synchronization.

Although an input device usually involves just the software reading the port, an output port can participate in both the read and write cycles very much like a regular memory. Figure 1.41 describes a “readable output port.” For an 8-bit output port, there will be 8 D flip flops to hold the values on the output pins. A write cycle to the port address will affect the values on the output pins. In particular, the microcomputer places information on the data bus and that information is clocked into the D flip flops. Since it is a readable output, a read cycle access from the port address returns the current values existing on the port pins. A “write-only output port” does not allow software to read the current values. There are no output-only ports on the 9S12.

**Figure 1.41**

A readable output port allows the software to generate external digital signals.

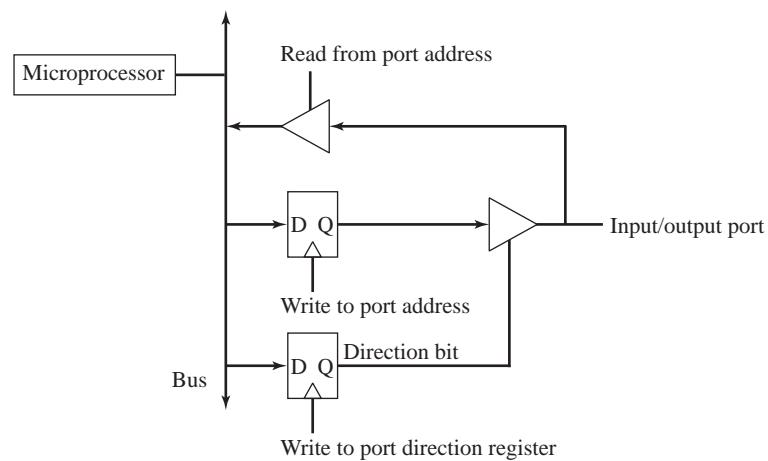


**Checkpoint 1.39:** What happens if the software reads from an output port as shown in Figure 1.41?

To make the microcontroller more marketable, most ports can be software-specified to be either inputs or outputs. Freescale uses the concept of a direction register to determine whether a pin is an input (direction register bit is 0) or an output (direction register bit is 1), as shown in Figure 1.42. We define a *ritual* as a program executed during startup that initializes hardware and software. If the ritual software makes direction bit zero, the port pin behaves like a simple input, and if it makes the direction bit one, the pin becomes a readable output. Most of the 9S12 ports operate as shown in Figure 1.42.

**Figure 1.42**

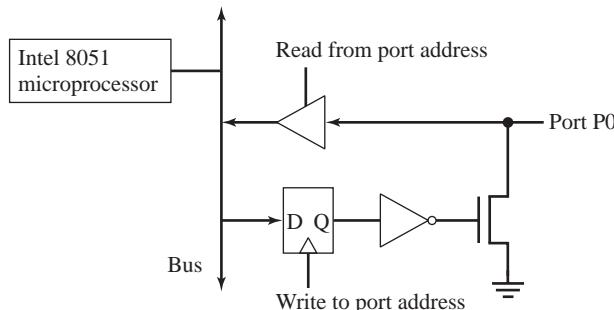
A bidirectional port can be configured as a read-only input port or a readable output port.



Intel uses open collector outputs to determine whether a pin is an input or an output, as shown in Figure 1.43. Port P0 on the Intel 8051 is open collector. If the software writes a “1” to the port, it behaves like a simple input port. If Port P0 is to be used as an output, you will need to add external pull-up resistors.

**Figure 1.43**

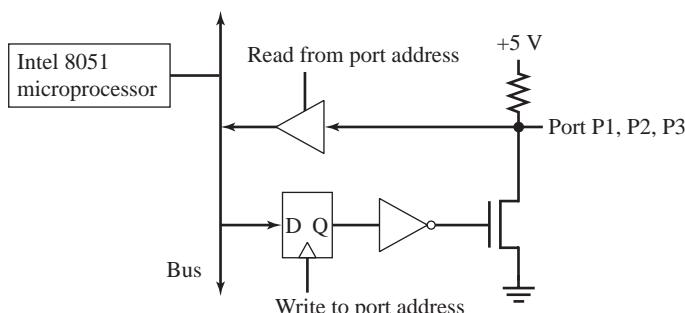
A bidirectional port is sometimes implemented with an open collector readable output port.



On the Intel 8051, Ports P1, P2, and P3 are open collector with internal pull-up resistors, as shown in Figure 1.44. If the software writes a “1” to the port, it behaves like a simple input port with the pullup to +5 V. If the software writes a “0” to the port, the output will be pulled low, and if it writes a “1”, the output will be pulled up by the internal resistor.

**Figure 1.44**

Some of the bidirectional ports on the Intel 8051 have internal pull-up resistors.



**Common error:** An output port that is created with open collector outputs with pull-up may not have enough  $I_{OH}$  to drive its external circuits.

**Common error:** Many program errors can be traced to confusion between I/O ports and regular memory. For example, you should not write to an input port, and sometimes you cannot read from an output port.

**Observation:** If a port pin is configured as a readable output but external loading causes the pin voltage to be different from the value written by the software, then a read from the port will return the voltage level at the pin and not the value last written by the software. This fact can be used by the software to detect excess loading by the external circuits.

## 1.9.2 Introduction to I/O Programming and the Direction Register

On most embedded microcomputers, the I/O ports are memory mapped. This means the software accesses an input/output port simply by reading from or writing to the appropriate address. To make our software more readable, we include symbolic definitions for the I/O ports, as previously shown in Programs 1.1 and 1.2. We use the direction register (e.g., DDRT) to specify which pins are input and which are output. Typically, we write to the data

register (e.g., PTT) *once* during the initialization phase. We use the data register to perform input/output on the port. Conversely, we read and write the data register *multiple times* to perform input and output, respectively, during the running phase. To initialize an I/O port, we set its direction register. The direction register specifies bit for bit whether the corresponding pins are input (0) or output (1). To make pins 7–4 input and pins 3–0 output, you should set the direction register to \$0F, as shown in Program 1.4.

#### Program 1.4

Software that initializes pins 7–4 to input and pins 3–0 to output on the 9S12.

<code>ldaa #\$0F ;PTT 7-4 input staa DDRT ;PTT 3-0 output</code>	<code>DDRT = 0x0F;</code>
--	---------------------------

If a pin is programmed as an input, then a write to that port has no effect on that pin. If a pin is programmed as an output, a write to that port will set or clear that pin. Notice that most 9S12 I/O ports have two I/O addresses, e.g., PTT and PTIT. On the 9S12, we have two options when reading pins that are configured as outputs. If a pin on the 9S12 is configured as an output, a read from the regular port address (e.g., PTT) will return the value that was previously written. In addition to the regular port addresses, the 9S12 has separate input addresses (e.g., PTIT), which are read-only. If a pin on the 9S12 is configured as an output, a read from the input address returns the value that exists on the pin at that time. We could compare PTT to PTIT to determine if any output pins are damaged or to detect excess loading. When a pin is an input, reading PTT or PTIT returns the same value, which is of course the input on that pin.

**Checkpoint 1.40:** Does the entire port need to be defined as input or output, or can some pins be input while others are output?

**Observation:** We can create open collector outputs (zero, hiZ) by setting the data port to zero at the beginning of our software, then setting the direction register to the complement of the desired output whenever we want to change the output.

**Example 1.1:** Design an embedded system that flashes LEDs in a 0101, 0110, 1010, 1001 binary repeating pattern.

**Solution:** Some problems are so unusual that they require the engineer to invent completely original solutions. Most of the time, however, the engineer can solve even complex problems by building the system from components that already exist. Creativity will still be required in selecting the proper components, making small changes in their behavior (tweaking), arranging them in an effective and efficient manner, then verifying the system satisfies both the requirements and constraints. When young engineers begin their first job, they are sometimes surprised to see that education does not stop with college graduation, but rather is a life-long activity. In fact, it is the educational goal of all engineers to continue to learn both processes (rules about how to solve problems) and products (hardware software components). As the engineer becomes more experienced, he or she has a larger toolbox from which processes and components can be selected.

The hardest step for most new engineers is the first one, where to begin? We begin by analyzing the problem to create a set of specifications and constraints. This system will need four LEDs, and the computer must be able to activate and deactivate them. Since the problem didn't specify power source, speed, color, or brightness, we could either put off these decisions until the engineering design stage in order to simplify the design or minimize cost, or we could go back to the customer and ask for additional requirements. In this case, the customer didn't care about power, speed, color or brightness, but did think minimizing cost was a good idea. Due to the nature of this book, we will constrain all our designs

to include the 9S12. Because we have +5 V microcomputer systems, we will specify the system to run on +5 V power. We have in our stockroom lost-cost red LEDs with 2.2 V, 10 mA specification, so we will use them. Table 1.20 summarizes the specifications and constraints. We will use standard 5% resistors to minimize cost.

**Table 1.20**  
Specifications and constraints of the LED output system.

Specifications	Constraints
Repeating pattern of 5, 6, 10, 9	9S12 based
Four 2.2 V, 10 mA red LEDs	Minimize cost
+5 V power supply	Standard 5% resistors

It is often difficult to distinguish whether a parameter is a specification or a constraint. In actuality, when designing a system it often doesn't matter into which category a parameter falls, because the system must satisfy all specifications and constraints. Nevertheless, when documenting the device it is better to categorize parameters properly. Specifications generally define in a quantitative manner the overall system objectives as given to us by our customers. Constraints, on the other hand, generally define the boundary space within which we must search for a solution to the problem. If we must use a particular component, it is often considered a constraint. In this case we will be using the 9S12. Constraints also are often defined as an inequality, such as the cost must be less than \$50, or the battery must last for at least one week. Specifications on the other hand are often defined as a quantitative number, and the system satisfies the requirement if the system operates within a specified *tolerance* of that parameter. Tolerance can be defined as a percentage error or as a range with minimum and maximum values. For example, our LED output system is acceptable if it has four LEDs, but unacceptable if it has three or five of them. Similarly, it will be OK as long as the LED current is between 8 and 12 mA. If the current drops below 8 mA, we won't be able to see the LED, and if it goes above 12 mA, it might damage the LED.

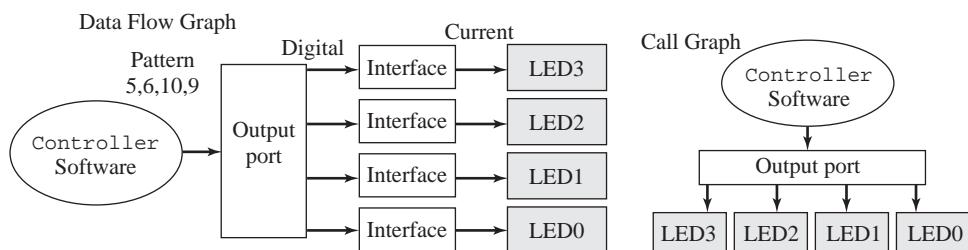
**Observation:** Defining realistic tolerances on our specifications will have a profound effect on system cost.

**Checkpoint 1.41:** What are the effects of specifying a tighter tolerance (e.g., 1% when the problem asked for 5%)?

**Checkpoint 1.42:** What are the effects of specifying a looser tolerance (e.g., 10% when the problem asked for 5%)?

The next step is the high-level design. The data flow graph in Figure 1.45 shows information as it flows from the controller software to the four LEDs. The data flow graph will be important during the subsequent design phases because the hardware blocks can be considered as a preliminary hardware block diagram of the system. The call graph, also shown in Figure 1.45, illustrates this is master/slave configuration in which the controller software will manipulate the four LEDs.

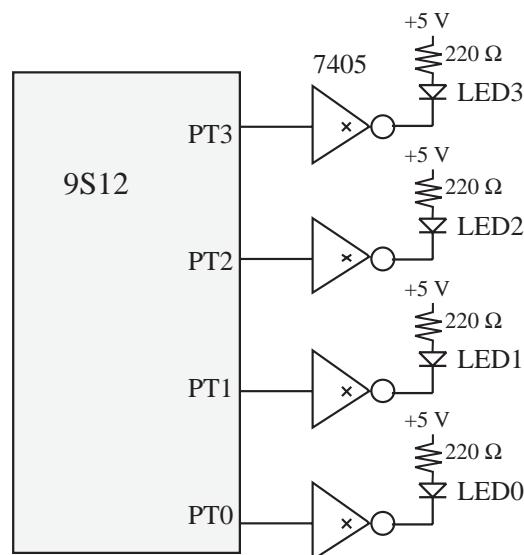
**Figure 1.45**  
Data flow graph and call graph of the LED output system.



The hardware design of this system simply involves using four copies of the LED interface presented earlier in Figure 1.20. We can use the 7405 because its  $I_{OL}$  (16mA) is enough to drive the 10 mA LED. Notice the similarity of the data flow graph in Figure 1.45 with the hardware circuit in Figure 1.46. The data flow graph and call graph in this example look similar because this system just performs output. We will need to tweak the circuit, adjusting the value of the resistor to produce the desired 2.2 V, 10 mA specification. If the  $V_{OL}$  of the 7405 is 0.4 V, and the voltage across the LED is 2.2 V, then the voltage across the resistor should be  $(5-2.2-0.4)$  V or 2.4 V. We calculate the resistor value using Ohm's Law— $R = \frac{V}{I}$  is  $2.4\text{ V}/10\text{ mA}$  or  $240\text{ }\Omega$ . Using standard resistor values with a 5% tolerance will make the product both cheaper and easier to build. One good way to tell whether a resistor value is standard is to go online and see which resistor values are in stock (e.g., [www.digikey.com](http://www.digikey.com), [www.jameco.com](http://www.jameco.com), [www.mouser.com](http://www.mouser.com)). In particular, 220  $\Omega$  and 270  $\Omega$  are two standard resistor values near 240  $\Omega$ . If we were to use 220  $\Omega$ , then the LED current would be  $(5-2.2-0.4)\text{ V}/220\text{ }\Omega$  or 10.9 mA. Similarly, if we were to use 270  $\Omega$ , then the LED current would be  $(5-2.2-0.4)\text{ V}/270\text{ }\Omega$  or 8.9 mA. Both would have been acceptable, but we will use the 220  $\Omega$  resistor because it is closer. It would have been more expensive to design this system with 240  $\Omega$  resistors.

**Figure 1.46**

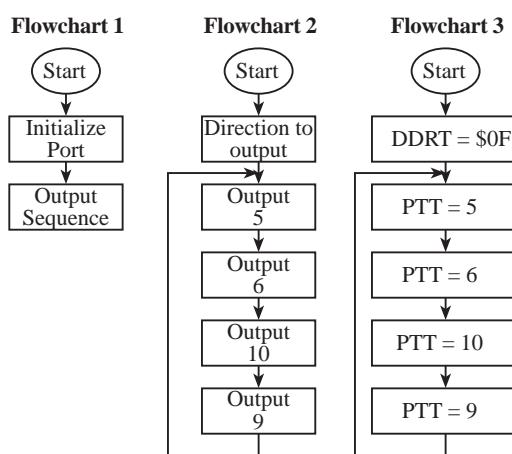
Hardware circuit for the LED output system.



The software design of this system also involves using examples presented earlier with some minor tweaking. The only data required in this problem is the 5,6,10,9 sequence. In Chapter 3 we will consider solutions to this type of problem using data structures, but in this first example, we will take a simple approach, not using a data structure. Figure 1.47 illustrates a software design process using *flowcharts*. We start with a general approach on the left. Flowchart 1 shows that the software will initialize the output port and perform the output sequence. As we design the software system, we fill in the details. This design process is called *successive refinement*. It is also classified as top-down, because we begin with high-level issues and end with the low-level. In Flowchart 2, we set the direction register, then output the 5,6,10,9. It is at this stage that we figured out how to create the repeating sequence. Flowchart 3 fills in the remaining details. To output the pattern 0101 to the LEDs, we will output a 5 to PTT on the 9S12. *Pseudo-code* is similar to high-level languages but lacks a rigid syntax. This means we can utilize whatever syntax we like. Flowcharts are good when the software involves complex algorithms with many decisions points causing control paths. On the other hand, pseudo-code may be better when the software is more sequential and involves complex mathematical calculations.

**Figure 1.47**

Software design for the LED output system using flowcharts.



Many software developers use pseudo-code rather than flowcharts, because the pseudo-code itself can be embedded into the software as comments. Program 1.5 shows the assembly code and C implementations. Notice the similarity in structure between Flowchart 3 and this code. Information following the semicolon is a comment, which allows programmers to document their software, but is ignored when converting to machine code. `org` is a pseudo-op instructing the assembler to place the subsequent software at the specified address. The first `org` specifies that the machine code for this system will be loaded into flash EEPROM. Making the bottom four bits of the port outputs was previously presented as Program 1.4. In order to set the LEDs to the pattern 0101, we first bring the value of 5 into Register A, then we output the 5 by storing register A to the port. We create the repeating pattern by using an unconditional branch at the end, so the 5,6,10,9 output pattern occurs over and over. The `reset vector` on the 9S12 is a 16-bit value stored at \$FFFE and \$FFFF. This 16-bit value is loaded into the program counter when the system starts up after a reset. In particular, this value specifies where our software will start execution. The last two lines of this program define the reset vector so our program will start at Main.

### Program 1.5

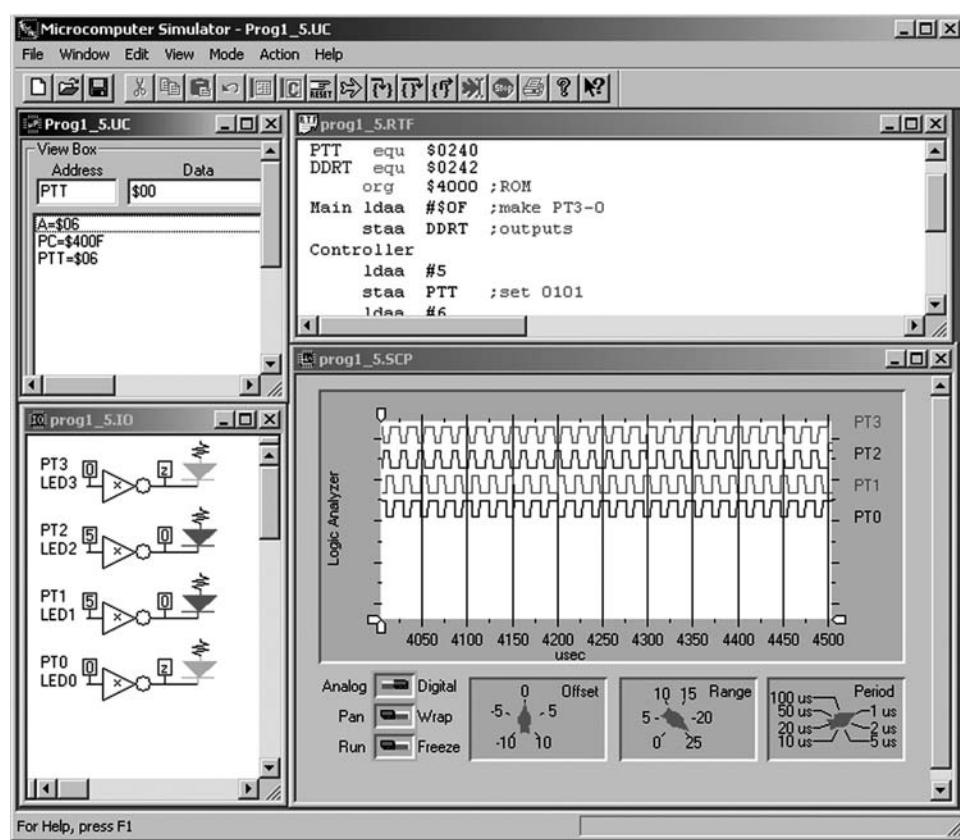
Assembly and C software for the LED output system.

<pre> org \$4000 ;ROM Main ldaa #\$0F ;make PT3-0         staa DDRT ;outputs Controller         ldaa #5         staa PTT ;set 0101         ldaa #6         staa PTT ;set 0110         ldaa #10         staa PTT ;set 1010         ldaa #9         staa PTT ;set 1001         bra Controller         org \$FFFE         fdb Main ;Reset vector       </pre>	<pre> void main(void){// make PT3-0   DDRT = 0x0F; // outputs   while(1){     PTT = 5; // 0101     PTT = 6; // 0110     PTT = 10; // 1010     PTT = 9; // 1001   } }       </pre>
--	---

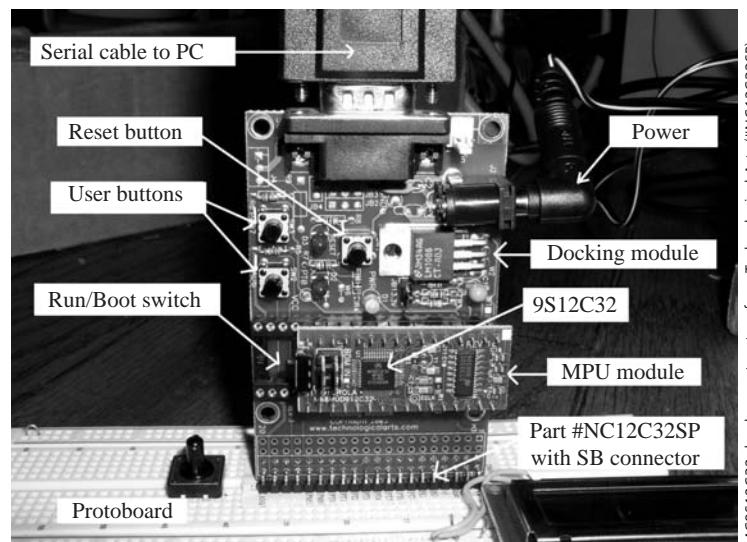
In order to test the system, we need to build a prototype. One option is simulation. A screen shot of this implementation can be seen as Figure 1.48. A second option is to use a development system like the one shown in Figure 1.49. In this approach, you build the external circuits on a protoboard and use the debugger to download and test the software.

**Figure 1.48**

TExaS simulation of the LED output system.

**Figure 1.49**

MC9S12C32 development system from Technological Arts (#NC12C32SP).



MC9S12C32 development system from Technological Arts (#NC12C32SP).

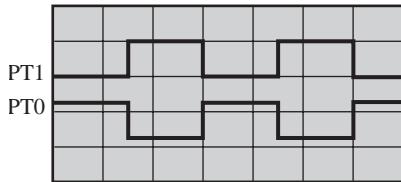
A third approach, is typically used after a successful evaluation with one of the previous methods. In this approach, we design a printed circuit board (PCB) including both the external circuits and the microcontroller itself. The 9S12, using the background debug module, has facilities to download software onto the microcontroller.

During the testing phase of the project, we observe that all four of the LEDs are continuously on. We use the software debugger to single-step our program, which correctly outputs the 0101, 0110, 1010, 1001 binary repeating pattern. During this single stepping, the

LEDs do come on and off in the proper pattern. Using a voltmeter on the circuit, we observe the 0.4V signal on the output of the 7405 whenever the software wishes to turn the LED on. We run the system at full speed again and observe two 7405 outputs on the oscilloscope, collecting data presented as Figure 1.50. Simulation, shown in Figure 1.48, also seems to be completely functional. All the electronic tests show the system is running properly, but our eyes still see all four LEDs continuously on.

**Figure 1.50**

Oscilloscope waveforms collected during the testing of the LED output system. The voltage sensitivity is 5V/division, and the time base is 1 $\mu$ s/division.



**Checkpoint 1.43:** What is the error in this design, and how do we fix it?

*Portability* is a measure of how easy it is to convert software that runs on one machine to run on another machine.

**Observation:** In general, C code is more portable than assembly language.

## 1.10 Choosing a Microcontroller

I chose to focus this book on the 9S12, because year after year Freescale continues to be a major supplier of microcontrollers. Sometimes, the computer engineer is faced with the task of selecting the microcontroller for the project. When faced with this decision, some engineers consider only those devices for which they have hardware and software experience. Fortunately, this blind approach often yields an effective and efficient product, because many microcontrollers overlap in their cost and performance. In other words, if a familiar microcontroller can implement the desired functions for the project, then it is often efficient to bypass that more perfect piece of hardware in favor of a faster development time. On the other hand, sometimes we wish to evaluate all potential candidates. It may be cost-effective to hire or train the engineering personnel so that they are proficient in a wide spectrum of potential microcontroller devices. There are many factors to consider when selecting a microcontroller, including the following:

- Labor costs include training, development, and testing
- Material costs include parts and supplies
- Manufacturing costs depend on the number and complexity of the components
- Maintenance costs involve revisions to fix bugs and perform upgrades
- ROM size must be big enough to hold instructions and fixed data for the software
- RAM size must be big enough to hold locals, parameters, and global variables
- EEPROM must hold nonvolatile fixed constants that are field-configurable
- Processor must be capable of performing all calculations in real time
- I/O bandwidth determines the input/output rate
- 8-, 16-, or 32-bit data size should match most of the data to be processed
- Numerical operations may be required, such as multiply, divide, signed, floating point
- Special functions may be required, such as multiply/accumulate, fuzzy logic, complex numbers
- Parallel ports are needed for the input/output digital signals
- Serial ports are used to interface with other computers or I/O devices
- Timer functions are used to generate signals, measure frequency, measure period

- Pulse-width modulation is used for the output signals in many control applications
- ADC is used to convert analog inputs to digital numbers
- Package size and environmental issues affect many embedded systems
- Second source availability increases the chance parts can be purchased
- Availability of high-level language cross-compilers, simulators, emulators is important
- Power requirements, are important because many systems will be battery operated

When considering speed, it is best to compare time to execute a benchmark program similar to your specific application, rather than just comparing bus frequency. One of the difficulties is that the microcomputer selection depends on the speed and size of the software, but the software cannot be written without the computer. Given this uncertainty, it is best to select a family of devices with a range of execution speeds and memory configurations. In this way a prototype system with large amounts of memory and peripherals can be purchased for software and hardware development, and once the design is in its final stages, the specific version of the computer can be selected now that the memory and speed requirements for the project are known. In conclusion, while this book focuses on the 9S12 microcontroller, it is expected that once the study of this book is completed, the reader will be equipped with the knowledge needed to select the proper microcontroller and complete the software design.

## 1.11 Exercises

For these questions, substitute your specific 9S12 for *YourComputer* as appropriate.

- 1.1** Is RAM volatile or nonvolatile?
- 1.2** Assuming *YourComputer* is running in expanded mode, what are the names of its bus signals?
- 1.3** Consider *YourComputer* with memory-mapped I/O. Assume there is no direct memory access (DMA) on your microcomputer. For this problem, we specify four classes of devices: *processor, RAM, ROM, and I/O*.
  - a)** List the devices that can drive the address bus during a CPU read cycle.
  - b)** List the devices that can drive the address bus during a CPU write cycle.
  - c)** List the devices that can drive the data bus during a CPU read cycle.
  - d)** List the devices that can drive the data bus during a CPU write cycle.
  - e)** List the devices that can drive the R/W line during a CPU read cycle.
  - f)** List the devices that can drive the R/W line during a CPU write cycle.
  - g)** List the devices that can receive the information from data bus during a CPU read cycle.
  - h)** List the devices that can receive the information from data bus during a CPU write cycle.
- 1.4** How many 74LS low-power Schottky inputs can an output of *YourComputer* drive?
- 1.5** Consider the current it takes to power a +5 V 74xx245 ( $I_{CC}$ ) as shown in Table 1.5. What is the qualitative difference between the CMOS devices and the non-CMOS devices? What is the explanation for the difference?
- 1.6** How many alternatives does a 14-bit ADC have?
- 1.7** If a system uses a 12-bit ADC, about how many decimal digits will it have?
- 1.8** If a system requires  $3\frac{1}{2}$  decimal digits of precision, what is the smallest number of bits the ADC needs to have?
- 1.9** How many alternatives does a 16-bit ADC have?
- 1.10** If a system uses an 11-bit ADC, about how many decimal digits will it have?
- 1.11** If a system requires  $2\frac{3}{4}$  decimal digits of precision, what is the smallest number of bits the ADC needs to have?
- 1.12** If a system requires  $4\frac{3}{4}$  decimal digits of precision, what is the smallest number of bits the ADC needs to have?

- 1.13** Convert the following decimal numbers to 8-bit unsigned binary: 25, 63, 125, and 200.
- 1.14** Convert the following decimal numbers to 8-bit signed binary: 25, 63, -125, and -2.
- 1.15** Convert the following hex numbers to unsigned decimal: \$25, \$63, \$A3, and \$FE.
- 1.16** Convert the 16-bit binary number %0010000001101010 to unsigned decimal.
- 1.17** Convert the 16-bit hex number \$1234 to unsigned decimal.
- 1.18** Convert the unsigned decimal number 1234 to 16-bit hexadecimal.
- 1.19** Convert the unsigned decimal number 10000 to 16-bit binary.
- 1.20** Convert the 16-bit hex number \$1234 to signed decimal.
- 1.21** Convert the 16-bit hex number \$ABCD to signed decimal.
- 1.22** Convert the signed decimal number 1234 to 16-bit hexadecimal.
- 1.23** Convert the signed decimal number -10000 to 16-bit binary.
- 1.24** List the registers in *YourComputer*.
- 1.25** How much RAM and ROM are in *YourComputer*? What are the specific address ranges of these memory components?
- 1.26** What are the bits in the CCR of *YourComputer*?
- 1.27** What are the bidirectional ports on *YourComputer*?
- 1.28** What are the unidirectional ports on *YourComputer* (if any)?
- 1.29** What is a direction register?
- 1.30** Why does the microcomputer have direction registers?
- D1.31** Design the circuit that interfaces a 3 V, 5 mA LED to *YourComputer*.
- D1.32** Design the circuit that interfaces a 2.5 V, 1 mA LED to *YourComputer*.
- D1.33** Redesign the switch interface shown in the middle of Figure 1.21 assuming the signal *s* is to be connected to a 74LS04 instead of the input port of the microcontroller. In particular, change the 10 k $\Omega$  resistor value to the appropriate value for low-power Schottky logic.
- D1.34** Write software that initializes 9S12 Port T so pins 7,5,3,1 are output and the rest are input.
- D1.35** Write software that initializes 9S12 Port T so pins 5,4 are output and the rest are input.
- D1.36** You can make the 6811 Port C generate open collector outputs by setting appropriate bit in the direction register, DDRC, to 1 and setting the CWOM bit in the PIOC register to 1. This open collector mode can be found in some of the ports of the Intel 8051 microcontroller family, but it is missing in the 9S12.

Freescale claims you can easily upgrade 6811 systems to the more powerful 9S12. You have just graduated from the Prestigious University, and the first task at your new high-paying job at *We Are Nerds, INC*, is to upgrade an existing *WAN INC* 6811 system to the 9S12. It seems easy, so you begin by reading all about the 9S12. The 9S12 has more instructions and addressing modes, but luckily all existing 6811 assembly instructions and addressing modes are still available on the 9S12. The 9S12 has more parallel ports, more serial ports, more input captures, and more output compares.

So far so good, but all of a sudden BAM it hits you—looking at you square in the face is a big problem. The existing system uses open collector output mode with the 6811 bit CWOM set, but the 9S12 has no open collector modes on any of its parallel port outputs. So now you, the young engineering genius from PU, must solve your first engineering problem.

Lucky for you, the engineers that worked on the problem previously were also from PU. They implemented the low-level access to PORTC as a device driver, and organized the software with a layered approach. This layered system will allow you to come in and replace the low (hardware access) level, without having to modify the upper levels. In particular, here are the existing low-level routines in both assembly and C.

```

;Initialize Parallel Port
;Input: Reg B specifies which port bits will be input(0) or
output(1)
;Outputs: none
Pinit ldaa PIOC
    oraa #$20 ; set CWOM so outputs are open collector
    staa PIOC
    stab DDRC ; 1 means open collector output, 0 means input
    ldaa #$FF
    stab PORTC ; any output pins are initialized to HiZ
    rts
;Set output (only output pins are effected)
;Input: Reg B specifies new output values
;Outputs: none
Pout    stab PORTC ; modifies output pins only
        rts
; Get input
; Input: none
; Outputs: Reg B returned with current values of both inputs
and outputs
Pin     ldab PORTC
        rts
void Pinit(unsigned char direction){
    PIOC |= 0x20; // set CWOM so outputs are open collector
    DDRC = direction; // 1 means open collector output, 0 means input
    Pout(0xFF); } // any output pins are initialized to HiZ
void Pout(unsigned char data){
    PORTC = data; } // modifies output pins only
unsigned char Pin(void){
    return PORTC; }

```

**Your specific task:** Rewrite the three device driver routines to run on the 9S12, either in assembly or in C. You may use any 9S12 assembly language instructions and addressing modes. Refer to the 9S12 parallel port using the symbol PTT and to the direction register using the symbol DORT. A global variable will be required.

**Test your answer with this example:** Assume the four input signals (C7–C4) are set by external hardware to C7=0,C6=1,C5=0, C4=0. Assume the four output signals (C3–C0) have +5 V pull-up resistors.

```

void main(void){ unsigned char info;
    Pinit(0x0F); // C7-C4 inputs(HiZ), C3-C0 outputs are HiZ
    info=Pin(); // info set to 0x4F (because of pullups)
    Pout(0x00); // C7-C4 still inputs(HiZ), but C3-C0 are low
    info=Pin(); // info set to 0x40
    Pout(0x05); // C7-C4,C2,C0 are HiZ, but C3,C1 are low
    info=Pin(); } // info set to 0x45

```

## 1.12 Lab Assignments

The labs in this book involve the following steps:

**Part a)** During the analysis phase of the project, determine additional specifications and constraints. In particular, discover which microcontroller you are to use, whether you are to develop in assembly language or in C, and whether the project is to be simulated then built, just built, or just simulated. For example, inputs can be created with switches, and outputs can be generated with LEDs. The SCI can be interfaced to a PC, and a communication program such as HyperTerminal can be used to interact with the system.

**Part b)** Design, build, and test the hardware interfaces. Use a computer-aided-drawing (CAD) program to draw the hardware circuits. Label all pins, chips, and resistor values. In this chapter, there will be one switch for each input and one LED for each output. Connect the switch interfaces to the 9S12 input pins, and connect the LED interfaces to the 9S12 output pins. Pressing the switch will signify a high input logic value. You should activate the LED to signify a high output logic value.

**Part c)** Design, implement, and test the software that initializes the I/O ports and performs the specified function. Often a main program is used to demonstrate the system.

**Lab 1.1.** The overall objective is to create a `not` gate. The system has one digital input and one digital output, such that the output is the logical complement of the input. Implement the design such that the complement function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `coma lsra` and `lsla` instructions.

**Lab 1.2.** The overall objective is to create a 3-input `and` gate. The system has three digital inputs and one digital output, such that the output is the logical `and` of the three inputs. Implement the design such that the `and` function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `anda lsra` and `lsla` instructions.

**Lab 1.3.** The overall objective is to create a 3-input `or` gate. The system has three digital inputs and one digital output, such that the output is the logical `or` of the three inputs. Implement the design such that the `or` function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `oraa lsra` and `lsla` instructions.

**Lab 1.4.** The overall objective is to create a 2-input `exclusive or` gate. The system has two digital inputs and one digital output, such that the output is the logical `exclusive or` of the two inputs. Implement the design such that the `exclusive or` function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `eora lsra` and `lsla` instructions.

**Lab 1.5.** The overall objective is to create a 3-input `voting` logic. The system has three digital inputs and one digital output, such that the output is high if and only if two or more inputs are high. This means the output will be low if two or more inputs are low. Implement the design such that the `voting` function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `anda oraa lsra` and `lsla` instructions.

# **2 Design of Software Systems**

---

## **Chapter 2 objectives are to:**

- ❖ Create an overview of assembly language programming
  - ❖ Develop techniques for writing modular or structured software
  - ❖ Introduce layered software organization
  - ❖ Present a software model for device drivers
  - ❖ Define the concept of threads
  - ❖ Describe effective techniques for software debugging
- 

**T**he ultimate success of an embedded system project depends on both its software and its hardware. Computer scientists pride themselves in their ability to develop quality software. Similarly, electrical engineers are well-trained in the processes to design both digital and analog electronics. Manufacturers, in an attempt to get designers to use their products, provide application notes for their hardware devices. The main objective of this book is to combine effective design processes with practical software techniques to develop quality embedded systems. As the size and especially the complexity of the software increase, the software development changes from simple “coding” to “software engineering,” and the required skills also vary along this spectrum. These software skills include modular design, layered architecture, abstraction, and verification. Real-time embedded systems are usually on the small end of the size scale, but nevertheless these systems can be quite complex. Therefore the above-mentioned skills are essential for developing embedded systems. This chapter on software development is placed early in the book because writing good software is an art that must be developed and cannot be added on at the end of a project. Good software combined with average hardware will always outperform average software on good hardware. In this chapter we will outline various techniques for developing quality software, then apply these techniques throughout the remainder of the book.

---

### **2.1 Quality Programming**

Software development is similar to other engineering tasks. We can choose to follow well-defined procedures during the development and evaluation phases, or we can meander in a haphazard way and produce code that is hard to test and harder to change. The ultimate goal of the system is to satisfy the stated objectives such as accuracy, stability, and I/O relationships. Nevertheless, it is appropriate to separately evaluate the individual

components of the system. Therefore in this section we will evaluate the quality of our software. There are two categories of performance criteria with which we evaluate the “goodness” of our software. Quantitative criteria include dynamic efficiency (speed of execution), static efficiency (ROM and RAM program size), and accuracy of the results. Qualitative criteria center around ease of software maintenance. Another qualitative way to evaluate software is ease of understanding. If your software is easy to understand, then it will be:

- Easy to debug (fix mistakes)
- Easy to verify (prove correctness)
- Easy to maintain (add features)

**Common error:** Programmers who sacrifice clarity in favor of execution speed often develop software that runs fast but doesn't work and can't be changed.

***Golden Rule of Software Development***

*Write software for others as you wish they would write for you.*

### 2.1.1 Quantitative Performance Measurements

To evaluate our software quality, we need performance measures. The simplest approaches to this issue are quantitative measurements. *Dynamic efficiency* is a measure of how fast the program executes. It is measured in seconds or CPU cycles. *Static efficiency* is the number of memory bytes required. Since most embedded computer systems have both RAM and ROM, we specify memory requirement in global variables, stack space, fixed constants, and program object code. The global variables plus maximum stack size must be less than the available RAM. Similarly, the fixed constants plus program size must be less than the ROM or EEPROM size. We can also judge our software system according to whether or not it satisfies given constraints, like software development costs, memory available, and timetable.

### 2.1.2 Qualitative Performance Measurements

Qualitative performance measurements include those parameters to which we cannot assign a direct numerical value. Often in life the most important questions are the easiest to ask but the hardest to answer. Such is the case with software quality. So we ask the following qualitative questions: Can we prove our software works? Is our software easy to understand? Is our software easy to change? Since there is no single approach to writing the best software, we can only hope to present some techniques that you may wish to integrate into your own software style. In fact, we will devote most of this chapter to the important issue of developing quality software. In particular, we will study self-documented code, abstraction, modularity, and layered software. These parameters indeed play a profound effect on the bottom-line financial success of our projects. Although quite real, because there often is not an immediate and direct relationship between a software's quality and profit, we may be tempted to dismiss its importance.

To get a benchmark on how good a programmer you are, we challenge you to two tests. In the first test, find a major piece of software that you have written over 12 months ago, then see if you can still understand it enough to make minor changes in its behavior. The second test is to exchange with a peer a major piece of software that you have both recently written (but not written together), then in the same manner see if you can make minor changes to each other's software.

**Observation:** You can tell that you are a good programmer if (1) you can understand your own code 12 months later and (2) others can make changes to your code.

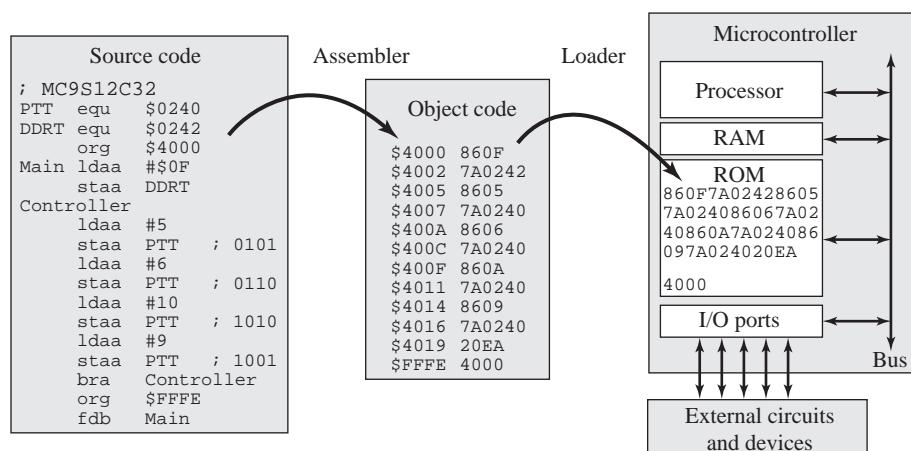
## 2.2 Assembly Language Programming

### 2.2.1 Introduction

In this section, we will present assembly language syntax for the **TExaS** assembler. There are minor syntactical differences between the assembler in **CodeWarrior** and the one in **TExaS**. However, most example programs in this book will be syntactically correct for both assemblers. Figure 2.1 outlines the assembly language development process. To develop assembly language software, we first use an **editor** to create our **source code**. Source code contains specific commands in human-readable form. Next, we use an **assembler** to translate our source code into **object code**. Object code contains the specific commands in machine-readable form. When developing software for a real microcontroller, a **loader** is used to place the object code into the microcontroller's memory. We test our system with the aid of a debugger. If the loader burns the object code into EEPROM, the program will exist in nonvolatile storage. Consequently, the power-on-reset will start our software after power is supplied to the microcontroller. The last two lines of the source code shown in Figure 2.1 define the reset vector, which is the starting location after a reset. Most microcontrollers have built-in features that assist in programming its EEPROM. Some older hardware development systems use RAM to hold the program. In these systems, the loader downloads the object code into RAM. Since RAM is volatile, the programs must be loaded each time power is removed.

**Figure 2.1**

Assembly language development process.



Assemblers are programs that process assembly language source program statements and translate them into executable machine language object files. Cross assemblers allow source programs written and edited on one computer (the host) generating executable code for another computer (the target). The executable code can either be simulated (using an application such as **TExaS** or **CodeWarrior**), or downloaded onto a real computer for execution.

The symbolic language used to code source programs to be processed by the assembler is called assembly language. The language is a collection of mnemonic symbols representing operations (i.e., machine instruction mnemonics or directives to the assembler), symbolic names, operators, and special symbols. The assembly language provides mnemonic operation codes for all machine instructions in the instruction set. The instructions are defined and explained in the Programming Reference Manual for the 9S12, which can be found in the **pdf** directory of the CD. These documents are also available for downloading from the **freescale.com** web site. A brief overview of each instruction and example usage can be found by executing **Help→OpCodes** with the

**TExaS** application. The assembly language also contains mnemonic directives that specify auxiliary actions to be performed by the assembler. These directives or pseudo-ops are not always translated into machine language. For more information execute **Help→PseudoOp**.

Most assemblers have two passes, meaning it will scan through the source code twice. During the first pass, the source program is analyzed in order to develop the symbol table. A **symbol table** is a mapping between symbolic names (e.g., PTT) and their numeric values (e.g., \$0240). During the second pass, the object file is created (assembled) using the symbol table that was developed in pass one. It is during the second pass that a **listing file** is also produced (see Program 2.1). The symbol table is recreated in the second pass. A phasing error occurs if any symbol table values calculated during the two passes are different.

### Program 2.1

Assembly language listing of Program 1.5.

```
Copyright 2011-2012 Test EXecute And Simulate
; MC9S12C32
$0240          PTT equ   $0240
$0242          DDRT equ  $0242
$4000          org    $4000 ;ROM
$4000 860F      [ 1]( 0){P }Main ldaa #$0F ;make PT3-0
$4002 7A0242      [ 3]( 1){wOP }} staa DDRT ;outputs
$4005          Controller
$4005 8605      [ 1]( 4){P }} ldaa #5
$4007 7A0240      [ 3]( 5){wOP }} staa PTT ;set 0101
$400A 8606      [ 1]( 8){P }} ldaa #6
$400C 7A0240      [ 3]( 9){wOP }} staa PTT ;set 0110
$400F 860A      [ 1]( 12){P }} ldaa #10
$4011 7A0240      [ 3]( 13){wOP }} staa PTT ;set 1010
$4014 8609      [ 1]( 16){P }} ldaa #9
$4016 7A0240      [ 3]( 17){wOP }} staa PTT ;set 1001
$4019 20EA      [ 3]( 20){PPP }} bra Controller
$FFFF          org    $FFFE
$FFFF 4000          fdb   Main ;Reset vector
*****Symbol Table*****
Controller $4005
DDRT     $0242
Main     $4000
PTT      $0240
Assembly successful
```

Errors that occur during the assembly process (e.g., undefined symbol, illegal opcode, branch destination too far, etc.) are explained in the listing file. Program 2.1 contains the listing file for the program presented as Program 1.5 and shown in Figure 2.1. The listing file created by **TExaS** contains the source code, the object code, and the symbol table.

The **source code** is a file of ASCII characters usually created with an editor. Each source statement is processed completely before the next source statement is read. As each statement is processed, the assembler examines the label, operation code, and operand fields. The operation code table is scanned for a match with a known opcode. During the processing of a standard operation code mnemonic, the standard machine code is inserted into the object file. If an assembler directive is being processed, the corresponding action is taken.

Any errors that are detected by the assembler are displayed. **Object code** is the binary values (instructions and data) that, when executed by the computer, perform the intended function. The listing file contains the address, object code, and a copy of the source code. There will also be a symbol table describing where in memory the program and data will

be loaded. The symbol table is a list of all the names used in the program along with the values. A symbol is created when you put a label starting in column 1. Examples of this type are `Main` and `Controller`. The symbol table value for this type is the absolute memory address where the instruction, variable, or constant will reside in memory. The second type of label is created by the `equ` pseudo-op, e.g., `PTT`. The value for this type of symbol is simply the number specified in the operand field. When the assembler processes an instruction with a symbol in it, it simply substitutes the fixed value in place of the symbol. It is good programming practice to use symbols that clarify (make it easier to understand) our programs. The symbol table for this example is given at the end of the listing file.

A compiler converts high-level language source code into object code. A cross-compiler also converts source code into object code and creates a listing file, except that the object code is created for a target machine that is different from the machine running the cross-compiler. **TExaS** is a cross-assembler because it runs on an Intel computer but creates 9S12 object code. **ImageCraft ICC12** and **Metrowerks CodeWarrior** include both a cross-assembler and a cross-compiler because they run on the PC and create 9S12 object code.

**Checkpoint 2.1:** What does the assembler do in pass 1?

**Checkpoint 2.2:** What does the assembler do in pass 2?

## 2.2.2 Assembly Language Syntax

Programs written in assembly language consist of a sequence of source statements. Each source statement consists of a sequence of ASCII characters ending with a carriage return. Each source statement may include up to four fields: a label, an operation (instruction mnemonic or assembler directive), an operand, and a comment. We use pseudo-op codes in our source code to give instructions to the assembler itself. The `equ` is an assembly directive, and the `ldaa` is a regular machine instruction.

```
PORTA equ $0000 ; Assembly time constant
Inp     ldaa PORTA ; Read data from fixed address I/O data port
```

An assembly language statement contains the following fields.

**Label Field** can be used to define a symbol

**Operation Field** defines the operation code or pseudo-op

**Operand Field** specifies either the address or the data.

**Comment Field** allows the programmer to document the software.

Instructions with inherent mode addressing do not have an operand field. For example,

```
label    clra      ;comment
        deca      ;comment
        cli       ;comment
        inca      ;comment
```

The **label field** begins in the first column of a source statement. The label field can take one of the following three forms:

- A.** An asterisk (\*) or semicolon (;) as the first character in the label field indicates that the rest of the source statement is a comment. Comments are ignored by the assembler, and are printed on the source listing only for the programmer's information. Examples:

```
* This line is a comment
; This line is also a comment
```

- B.** A white-space character (blank or tab) as the first character indicates that the label field is empty. The line has no label and is not a comment. These assembly lines have no labels:

```
ldaa 0
rmb 10
```

- C.** A symbol character as the first character indicates that the line has a label. Symbol characters are the uppercase letters A–Z, lowercase letters a–z, digits 0–9, and the special characters, such as period (.), dollar sign (\$), and underscore (\_). Symbols consist of at least one and at most 99 characters, the first of which must be alphabetic or the special characters period (.) or underscore (\_). All characters are significant and upper and lowercase letters are distinct.

Each label may be defined only once in your program. The exception to this rule is the `set` pseudo-op that allows you to define and redefine the same symbol. We typically use `set` to define the stack offsets for the local variables in a subroutine. The `set` pseudo-op allows two separate subroutines to re-use the same name for their local variables.

With the exception of the `equ =` and `set` directives, a label is assigned the value of the program counter of the first byte of the instruction or data being assembled. The value assigned to the label is absolute. Labels may optionally be ended with a colon (:). If the colon is used, it is not part of the label but merely acts to set the label off from the rest of the source line. Thus, the following code fragments are equivalent:

```
here: deca
      bne here

here  deca
      bne here
```

A label may appear on a line by itself. The assembler interprets this as set the value of the label equal to the current value of the program counter. A label may also occur on a line with a pseudo-op.

The **operation field** occurs after the label field and must be preceded by at least one white-space character. The operation field must contain a legal opcode mnemonic or an assembler directive. Uppercase characters in this field are converted to lowercase before being checked as a legal mnemonic. Thus `nop`, `NOP`, and `NoP` are recognized as the same mnemonic. Entries in the operation field may be opcodes or directives.

*Opcodes* correspond directly to the machine instructions. The operation code includes any register name associated with the instruction. These register names must not be separated from the opcode with any white-space characters. Thus `clra` means clear accumulator A, but `clr a` means clear memory location identified by the label `a`.

*Directives* or *pseudo-ops* are special operation codes known to the assembler that control the assembly process rather than being translated into machine instructions. The directives that **TExaS** supports are described in detail later in this section.

The interpretation of the **operand field** is dependent on the contents of the operation field. The operand field, if required, must follow the operation field, and must be preceded by at least one white-space character. The operand field may contain a symbol, an expression, or a combination of symbols and expressions separated by commas. There can be no white-spaces in the operand field. For example, the following two lines produce identical object code in TExaS because of the space between data and + in the first line:

```
ldaa data + 1
ldaa data
```

**Observation:** The CodeWarrior assembler allows spaces within the operand field, and then requires a semicolon (;) be placed before each comment.

The operand field of machine instructions is used to specify the addressing mode of the instruction, as well as the operand of the instruction. Table 2.1 summarizes the operand field formats on the 9S12. On the 9S12, the index register, `idx`, can be X, Y, SP, or PC.

**Table 2.1**

Example operands for the 9S12.

Operand	Format	Example
no operand	inherent	clra
<expression>	direct, extended, or relative	ldaa 4
#<expression>	immediate	ldaa #4
<expression>,idx	indexed with address register	ldaa 4,x
<expr>,#<expr>	bit set or clear	bset 4,#\$01
<expr>,#<expr>,<expr>	bit test and branch	brset 4,#\$01,there
<expr>,idx ,#<expr>,<expr>	bit test and branch	brset 4,x,#\$01,there

The 9S12 assembly language includes some additional operand formats, as shown in Table 2.2. The accumulator offset, `acc`, is A, B or D, and the index register, `idx`, is X, Y, SP, or PC. The PC is not allowed with any of the predecrement, postdecrement, preincrement, or postincrement addressing modes.

**Table 2.2**

Additional example operands for the 9S12.

Operand	Format	Example
<expression>,idx+	indexed, post increment	ldd 2,SP+
<expression>,idx-	indexed, post decrement	ldaa 4,Y-
<expression>,+idx	indexed, pre increment	ldaa 4,+X
<expression>,-idx	indexed, pre decrement	staa 1,-SP
acc, idx	accumulator offset indexed	ldaa A,X
[<expression>,idx]	indexed indirect	ldaa [4,X]
[D,idx]	RegD indexed indirect	ldaa [D,Y]

### 2.2.3 Memory and Register Transfer Operations

The w w U terminology used in this section was presented earlier in Section 1.6.2. The 8-bit load instructions transfer data from memory into a register. In real life, when we *move* a box, *push* a broom, *load* a rifle, or *transfer* to a new job, there is a single physical object and the action changes the location of that object. Assembly language uses these same verbs, but the action will be different. It creates a copy of the data and places it at the new location. In other words, since the original data still exists, there are now two copies of the information. If the address U is between 0 and \$00FF, then direct addressing mode will be used; otherwise, it will use extended addressing mode.

ldaa	#w	RegA=w	Load an 8-bit constant into RegA
ldaa	U	RegA=[U]	Load an 8-bit memory value into RegA
ldab	#w	RegB=w	Load an 8-bit constant into RegB
ldab	U	RegB=[U]	Load an 8-bit memory value into RegB

Condition code bits are set with R equal to the 8-bit memory contents loaded into the register.

N: result is negative N=R<sub>7</sub>

Z: result is zero Z =  $\overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

V: signed overflow V=0

The 16-bit load instructions also transfer information from memory into a register. Although D usually contains data and X,Y usually contain addresses, it is acceptable programming practice to place address or data information in any of these three registers. The

stack pointer (called either S or SP) will always contain an address specifying the top of the stack. The program counter (PC) will always contain an address specifying the next instruction to execute.

ldd	#W	RegD=W	Load a 16-bit constant into RegD
ldd	U	RegD={U}	Load a 16-bit memory value into RegD
lds	#W	RegS=W	Load a 16-bit constant into RegS
lds	U	RegS={U}	Load a 16-bit memory value into RegS
ldx	#W	RegX=W	Load a 16-bit constant into RegX
ldx	U	RegX={U}	Load a 16-bit memory value into RegX
ldy	#W	RegY=W	Load a 16-bit constant into RegY
ldy	U	RegY={U}	Load a 16-bit memory value into RegY

Condition code bits are set with R equal to the 16-bit memory contents loaded into the register.

N: result is negative N=R<sub>15</sub>

Z: result is zero

$$Z = \overline{R_{15}} \cdot \overline{R_{14}} \cdot \overline{R_{13}} \cdot \overline{R_{12}} \cdot \overline{R_{11}} \cdot \overline{R_{10}} \cdot \overline{R_9} \cdot \overline{R_8} \cdot \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

V: signed overflow V=0

The 9S12 has two very convenient memory to memory move instructions, which set no flags.

movb	#w, addr	[addr]=w	Move an 8-bit constant into memory
movb	addr1, addr2	[addr2]=[addr1]	Move an 8-bit value memory to memory
movw	#W, addr	{addr}=W	Move a 16-bit constant into memory
movw	addr1, addr2	{addr2}={addr1}	Move a 16-bit value memory to memory

The 8-bit store instructions move data from a register to memory. The data in the register remains intact, so after executing one of these instructions there are two copies of the data (both in the register and memory).

staa	U	[U]=RegA	Store RegA into memory
stab	U	[U]=RegB	Store RegB into memory

Condition code bits are set with R equal to the 8-bit register contents stored into memory.

N: result is negative N=R<sub>7</sub>

$$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

V: signed overflow V=0

The 16-bit store instructions move from a register to memory.

std	U	{U}=RegD	Store RegD into memory
sts	U	{U}=RegS	Store RegS into memory
stx	U	{U}=RegX	Store RegX into memory
sty	U	{U}=RegY	Store RegY into memory

Condition code bits are set with R equal to the 16-bit register contents stored into memory.

N: result is negative N=R<sub>15</sub>

Z: result is zero

$$Z = \overline{R_{15}} \cdot \overline{R_{14}} \cdot \overline{R_{13}} \cdot \overline{R_{12}} \cdot \overline{R_{11}} \cdot \overline{R_{10}} \cdot \overline{R_9} \cdot \overline{R_8} \cdot \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

V: signed overflow V=0

The following transfer operations use inherent addressing:..

xgdx	Swap RegD and RegX
xgdy	Swap RegD and RegY
clc	Clear carry bit, C=0
cli	Clear interrupt mask bit, enable interrupts, I=0
clv	Clear overflow bit, V=0
sec	Set carry bit, C=1
sei	Set interrupt mask bit, disable interrupts, I=1
sev	Set overflow bit, V=1
tap	Transfer A to CC, (can not change X bit from 0 to 1)
tpa	Transfer CC to A

## 2.2.4 Indexed Addressing Mode

The 9S12 instruction set has 15 addressing modes. Five of the modes were presented earlier, and the remaining modes are presented next.

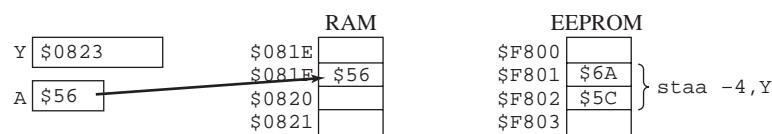
1. **Indexed** addressing mode uses a fixed offset with the 16-bit registers: X, Y, SP, or PC. On the 6811, instructions that use register Y take more memory and run slower than the equivalent instruction using register X. The 9S12 eliminates this speed and memory cost of using Reg Y. In addition, the 9S12 indexing modes can be used with the stack pointer (SP) and the program counter (PC). The offset can be 5-bit (-16 to +15), 9-bit (-256 to +127), or 16-bit. Five-bit (-16 to +15) index mode requires one machine byte to encode the operand. In the first example that uses 5-bit indexed mode, \$6A is the staa instruction and \$5C is the index mode operand. Tables A.3 and A.4 of the Freescale S12CPU Reference Manual show the machine codes for the indexed instructions.

machine code	opcode	operand	comment
\$6A5C	staa	-4, Y	; [Y-4] = RegA

Assuming Register Y=\$0823, the instruction staa -4, Y will store a copy of the value in Register A at \$081F leaving Register Y unchanged, as shown in Figure 2.2. The effective address (EA) is \$0823-4=\$081F.

**Figure 2.2**

Example of the 9S12 indexed addressing mode.



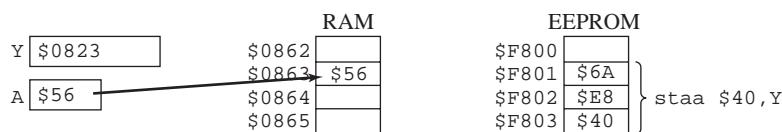
Nine-bit (-256 to +255) indexed mode requires two machine bytes to encode the operand.

machine code	opcode	operand	comment
\$6AE840	staa	\$40, Y	; [Y+\$40] = RegA

Assuming Register Y=\$0823, the instruction staa \$40, Y will store a copy of the value in Register A at \$0863 leaving Register Y unchanged, as shown in Figure 2.3. The effective address (EA) is \$0823+\$40=\$0863.

**Figure 2.3**

Another example of the 9S12 indexed addressing mode.



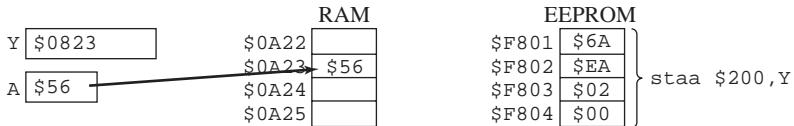
Sixteen-bit indexed mode requires three machine bytes to encode the operand.

machine code	opcode	Operand	comment
\$6AEA0200	staa	\$200,Y	; [Y+\$200] = RegA

Assuming Register Y=\$0823, the instruction staa \$200,Y will store a copy of the value in Register A at \$0A23 leaving Register Y unchanged, as shown in Figure 2.4. The effective address (EA) is \$0823+\$200=\$0A23.

**Figure 2.4**

A third example of the 9S12 indexed addressing mode.



Due to the properties of 16-bit addition, the 16-bit offset can be interpreted either as unsigned (0 to 65535) or signed (-32768 to +32767.) Indexed mode is useful when addressing the data structures and information on the stack. In each case, the 16-bit register used as a pointer (index) is not modified by the instruction.

**Common error:** SP relative indexed addressing with a negative constant is usually defined as an illegal stack access.

**2. Auto Pre/Post Decrement/Increment Indexed** addressing modes use the 16-bit registers: X, Y, or SP. The PC cannot be used with these index modes that modify the index register. In each case, the 16-bit register used as a pointer (index) is modified either before (pre) or after (post) the memory access. These modes are useful when addressing the data structures. The 9S12 allows the programmer to specify the amount added to (subtracted from) the index register from 1 to 8. In each case assume RegY is initially 2345.

Post-increment examples are as follows:

```
staa 1,Y+ ;Store the value in RegA at 2345, then RegY=2346
staa 4,Y+ ;Store the value in RegA at 2345, then RegY=2349
```

Pre-increment examples are as follows:

```
staa 1,+Y ;RegY=2346, then store the value in RegA at 2346
staa 4,+Y ;RegY=2349, then store the value in RegA at 2349
```

Post-decrement examples are as follows:

```
staa 1,Y- ;Store value in RegA at 2345, then RegY=2344
staa 4,Y- ;Store value in RegA at 2345, then RegY=2341
```

Pre-decrement examples are as follows:

```
staa 1,-Y ;RegY=2344, then store the value in RegA at 2344
staa 4,-Y ;RegY=2341, then store the value in RegA at 2341
```

**Observation:** Usually we would add/subtract one when accessing an 8-bit value and add/subtract two when accessing a 16-bit value.

**Common error:** The improper use of these index modes with the SP can result in an illegal stack access or unbalanced stack.

**3. Accumulator Offset Indexed** addressing mode uses two registers. The offset is located in one of the accumulators A, B, or D, and the index (memory address) uses the 16-bit registers: X, Y, SP, or PC. In each case, the accumulator used for the offset and the index register used as a pointer (index) are not modified by the instruction. Examples:

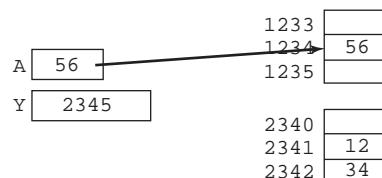
```
ldab #4
ldy #2345
staa B,Y      ;Store the value in RegA at 2349 (B & Y unchanged)
```

**4. Indexed Indirect** addressing mode uses a fixed offset with the 16-bit registers: X, Y, SP, or PC. The fixed offset is always 16 bits. The fixed 16-bit value is added to the index register (X, Y, SP, or PC), and used to fetch a second 16-bit big endian address from memory. The load or store is performed at this second address, as shown in Figure 2.5. Indexed indirect mode is useful when data structures contain pointers. In each case, the 16-bit index register and the memory pointer are not modified by the instruction. For example,

```
ldy #$2345
staa [-4,Y]    ;fetch 16-bit address from $2341, store $56 at $1234
```

**Figure 2.5**

Example of the 9S12 indexed-indirect addressing mode.

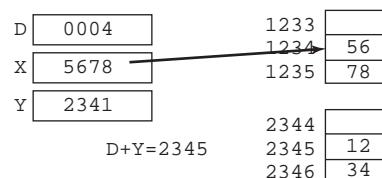


**5. Accumulator D Offset Indexed Indirect** addressing mode uses two registers. The offset is located in accumulator D, and the index (memory address) is in one of the 16-bit registers: X, Y, SP, or PC. The value in D is added to the index register (X, Y, SP, or PC), and is used to fetch a second 16-bit big endian address from memory, as shown in Figure 2.6. The load or store is performed at this second address. This mode is also useful when data structures contain pointers. In each case, accumulator D and the index register used as a pointer (index) are not modified by the instruction. For example,

```
ldd #4
ldy #$2341
stx [D,Y]  ;Store the value in RegX at $1234
```

**Figure 2.6**

Example of the 9S12 accumulator-offset indexed-indirect addressing mode.



The 9S12 load effective address instructions can only be used with indexed addressing mode. These instructions are very useful for manipulating the 16-bit registers. Let idx represent one of the preceding index addressing modes. They do not affect any condition code bits.

```
leax idx      ;RegX=EA
leay idx      ;RegY=EA
leas idx      ;RegS=EA
```

The basic idea is that the effective address is calculated in the usual manner. But rather than fetching the memory contents at that address as a regular load instruction (`ldaa` `ldab` `ldd` `ldx` `ldy` `lds`) would, this instruction puts the effective address itself into the register. In each of the following cases, the effective address, EA, is loaded into Register X.

```

leax m,r      ;IDX 5-bit index, EA=r+m (-16 to 15)
leax v,+r     ;IDX pre-increment r=r+v, EA=r (1 to 8)
leax v,-r     ;IDX pre-decrement r=r-v, EA=r (1 to 8)
leax v,r+     ;IDX post-increment, EA=r, r=r+v (1 to 8)
leax v,r-     ;IDX post-decrement, EA=r, r=r-v (1 to 8)
leax A,r      ;IDX Reg A offset EA=r+A, zero padded
leax B,r      ;IDX Reg B offset EA=r+B, zero padded
leax D,r      ;IDX Reg D offset EA=r+D
leax q,r      ;IDX1 9-bit index EA=r+q (-256 to 255)
leax W,r      ;IDX2 16-bit index EA=r+W (-32768 to 65535)

```

where r is Reg X, Y, SP, or PC, and the fixed constants are

m is any signed 5-bit -16 to +15

q is any signed 9-bit -256 to +255

v is any unsigned 3-bit 1 to 8

w is any signed 16-bit -32768 to +32767 or any unsigned 16-bit 0 to 65535

**Observation:** The `leas -4,sp` instruction subtracts four from the stack pointer which causes 4 bytes to be allocated on the stack.

**Checkpoint 2.3:** Write 9S12 assembly code that sets Register Y equal to X+10.

## 2.2.5 Arithmetic Operations

It is important to remember that arithmetic operations (addition, subtraction, multiplication, and division) have constraints when performed with finite precision on a microcomputer. An overflow error occurs when the result of an arithmetic operation cannot fit into the finite precision of the result. For example, when two 8-bit numbers are added, the sum is 9 bits, and thus it may not fit into the 8-bit result. The same digital hardware (instructions) will be used to add and subtract unsigned and signed numbers. On the other hand, we will need separate overflow detection for signed and unsigned addition and subtraction (C bit and V bit).

It is common for computers to perform arithmetic operations using a register such as Register A. As we have seen, a *register* is a high-speed storage inside the processor. An *accumulator*, such as Register A, is a register with which arithmetic and logic operations can be performed. The following instructions are a few of the arithmetic functions available on the 9S12, which fetch data from memory and add/subtract it from the register. With immediate mode (#w) the 8-bit constant is located in the instruction itself. With direct mode and extended mode (U) the 8-bit data is fetched from memory location U. Recall that direct/extended mode affects the size of the address, not the size of the data. The size of the data will be determined by the size of the register into which the operation will be performed.

All microcomputers have a *condition code register* (CC or CCR) that specifies the status of the most recent operation. In this section, we will introduce the four condition code bits common to most microcomputers, shown in Table 2.3. If the two inputs to an addition or subtraction operation are considered as unsigned, then the C bit (carry) will be set if the result does not fit. In other words, after an unsigned operation, the C bit is set if the answer is wrong. If the two inputs to an addition or subtraction operation are considered as signed, then the V bit (overflow) will be set if the result does not fit. In other words, after a signed operation, the V bit is set if the answer is wrong.

**Table 2.3**

Condition code bits contain the status of the previous arithmetic or logical operation.

Bit	Name	Meaning after Addition or Subtraction
N	negative	result is negative
Z	zero	result is zero
V	overflow	signed overflow
C	carry	unsigned overflow

The `adda` and `addb` instructions add an 8-bit value from memory to the corresponding register. These instructions work for both signed and unsigned data.

adda	#w	$\text{RegA} = \text{RegA} + w$	Add 8-bit constant to RegA
adda	U	$\text{RegA} = \text{RegA} + [U]$	Add 8-bit memory value to RegA
addb	#w	$\text{RegB} = \text{RegB} + w$	Add 8-bit constant to RegB
addb	U	$\text{RegB} = \text{RegB} + [U]$	Add 8-bit memory value to RegB

Condition code bits are set after  $R = X + M$ , where X is initial register value, R is the final register value.

N: result is negative  $N = R_7$

Z: result is zero  $Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

V: signed overflow  $V = X_7 \cdot M_7 \cdot \overline{R_7} + \overline{X_7} \cdot \overline{M_7} \cdot R_7$

C: unsigned overflow  $C = X_7 \cdot M_7 + M_7 \cdot \overline{R_7} + \overline{R_7} \cdot X_7$

Let N and M be 8-bit unsigned locations. The following assembly code implements  $M = N + 25$ :

```
ldaa N
adda #25 ;RegA=N+25, error if C is set
staa M
```

The `adddd` instruction adds a 16-bit value from memory to Register D. This instruction works for both signed and unsigned data.

adddd	#W	$\text{RegD} = \text{RegD} + W$	Add 16-bit constant to RegD
adddd	U	$\text{RegD} = \text{RegD} + [U]$	Add 16-bit memory value to RegD

Condition code bits are set after  $R = D + M$ , where D is initial register value, R is the final register value.

N: result is negative  $N = R_{15}$

Z: result is zero  $Z = \overline{R_{15}} \cdot \overline{R_{14}} \cdot \overline{R_{13}} \cdot \overline{R_{12}} \cdot \overline{R_{11}} \cdot \overline{R_{10}} \cdot \overline{R_9} \cdot \overline{R_8} \cdot \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

V: signed overflow  $V = D_{15} \cdot M_{15} \cdot \overline{R_{15}} + \overline{D_{15}} \cdot \overline{M_{15}} \cdot R_{15}$

C: unsigned overflow  $C = D_{15} \cdot M_{15} + M_{15} \cdot \overline{R_{15}} + \overline{R_{15}} \cdot D_{15}$

Let N and M be 16-bit unsigned locations. The following assembly code implements  $M = N + 1000$ .

```
ldd N
adddd #1000 ;RegD=N+1000, error if C is set
std M
```

**Checkpoint 2.4:** Write assembly code that adds a constant 2000 to Register Y.

These instructions subtract an 8-bit memory value from a register. The operation works for both signed and unsigned values. The compare and test instructions do not change the

register value. The condition code bits can be used by a conditional branch instruction to compare the two values. If the numbers represent unsigned values, then follow a subtraction/compare with an unsigned conditional branch: `beq bne bhi bhs blo bls`. If the numbers represent signed values, then follow a subtraction/compare with a signed conditional branch: `beq bne bgt bge blt ble`.

<code>cmpa</code>	<code>#w</code>	RegA-w	Compare RegA to 8-bit constant
<code>cmpa</code>	<code>U</code>	RegA-[U]	Compare RegA to 8-bit memory value
<code>cmpb</code>	<code>#w</code>	RegB-w	Compare RegB to 8-bit constant
<code>cmpb</code>	<code>U</code>	RegB-[U]	Compare RegB to 8-bit memory value
<code>suba</code>	<code>#w</code>	RegA=RegA-w	Subtract 8-bit constant from RegA
<code>suba</code>	<code>U</code>	RegA=RegA-[U]	Subtract 8-bit memory value from RegA
<code>subb</code>	<code>#w</code>	RegB=RegB-w	Subtract 8-bit constant from RegB
<code>subb</code>	<code>U</code>	RegB=RegB-[U]	Subtract 8-bit memory value from RegB
<code>tsta</code>		RegA-0	Test RegA
<code>tstb</code>		RegB-0	Test RegB

Condition code bits are set after  $R=X-M$ , X is initial register value, and R is the final register value.

N: result is negative  $N=R_7$

Z: result is zero  $Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

V: signed overflow  $V = X_7 \cdot \overline{M_7} \cdot \overline{R_7} + \overline{X_7} \cdot M_7 \cdot R_7$

C: unsigned overflow  $C = \overline{X_7} \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \overline{X_7}$

Let N and M be 8-bit unsigned locations. The following assembly code implements  $M=N-10$ .

```
ldaa N
suba #10      ;RegA=N-10, error if C is set
staa M
```

These instructions subtract a 16-bit memory value from a register. Just like the 8-bit subtraction operators, these operators work for both signed and unsigned values. Again, the condition code bits can be used by a conditional branch instruction to compare the two values.

<code>cpd</code>	<code>#W</code>	RegD-W	Compare RegD to 16-bit constant
<code>cpd</code>	<code>U</code>	RegD-{U}	Compare RegD to 16-bit memory value
<code>cpx</code>	<code>#W</code>	RegX-W	Compare RegX to 16-bit constant
<code>cpx</code>	<code>U</code>	RegX-{U}	Compare RegX to 16-bit memory value
<code>cpy</code>	<code>#W</code>	RegY-W	Compare RegY to 16-bit constant
<code>cpy</code>	<code>U</code>	RegY-{U}	Compare RegY to 16-bit memory value
<code>subd</code>	<code>#W</code>	RegD=RegD-W	Subtract 16-bit constant from RegD
<code>subd</code>	<code>U</code>	RegD=RegD-{U}	Subtract 16-bit memory value from RegD

Condition code bits are set after  $R=X-M$ , X is initial register value, and R is the final register value.

N: result is negative  $N=R_{15}$

Z: result is zero  $Z = \overline{R_{15}} \cdot \overline{R_{14}} \cdot \overline{R_{13}} \cdot \overline{R_{12}} \cdot \overline{R_{11}} \cdot \overline{R_{10}} \cdot \overline{R_9} \cdot \overline{R_8} \cdot \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

V: signed overflow  $V = X_{15} \cdot \overline{M_{15}} \cdot \overline{R_{15}} + \overline{X_{15}} \cdot M_{15} \cdot R_{15}$

C: unsigned overflow  $C = \overline{X_{15}} \cdot M_{15} + M_{15} \cdot R_{15} + R_{15} \cdot \overline{X_{15}}$

Let N and M be 16-bit unsigned locations. The following assembly code implements  $M=N-1000$ .

```
ldd N
subd #1000 ;RegD = N-1000, error if C is set
std M
```

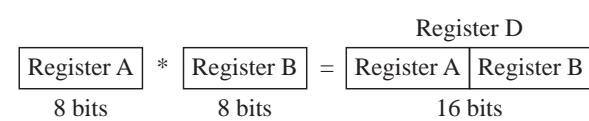
There are increment and decrement instructions, which operate properly on either signed or unsigned values. These instructions use inherent addressing. The Z bit is set if the result is zero.

deca	RegA=RegA-1	Decrement RegA
decb	RegA=RegA-1	Decrement RegB
dex	RegX=RegX-1	Decrement RegX
dey	RegY=RegY-1	Decrement RegY
inca	RegA=RegA+1	Increment RegA
incb	RegB=RegB+1	Increment RegB
inx	RegX=RegX+1	Increment RegX
iny	RegY=RegY+1	Increment RegY

The `mul` instruction performs an 8-bit by 8-bit into 16-bit unsigned multiply, giving `RegD` equal to `RegA` times `RegB`, as shown in Figure 2.7. No overflow is possible.

**Figure 2.7**

The `mul` instruction takes two 8-bit inputs and generates a 16-bit product.



Condition code bits are set after  $R=A*B$ .

C:  $R_7$ , set if bit 7 of the 16-bit result is one

**Checkpoint 2.5:** Prove the `mul` instruction can't overflow when multiplying two 8-bit unsigned numbers yielding a 16-bit product.

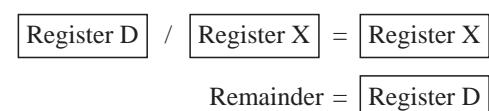
Let N and M be 8-bit unsigned locations. The following assembly code implements  $M=3*N$ .

```
ldaa N
ldab #3
mul      ;RegD=*N, error if RegA is not zero
stab M
```

The `idiv` instruction performs a 16-bit by 16-bit unsigned divide with remainder, giving  $RegX=RegD/RegX$ , as shown in Figure 2.8. Register D is the remainder.

**Figure 2.8**

The `idiv` instruction takes two 16-bit inputs and generates a 16-bit quotient and a 16-bit remainder.



Condition code bits are set after quotient=dividend/divisor or  $Q=D/X$ .

Z: result is zero,  $Z = \overline{Q_{15}} \cdot \overline{Q_{14}} \cdot \overline{Q_{13}} \cdot \overline{Q_{12}} \cdot \overline{Q_{11}} \cdot \overline{Q_{10}} \cdot \overline{Q_9} \cdot \overline{Q_8} \cdot \overline{Q_7} \cdot \overline{Q_6} \cdot \overline{Q_5} \cdot \overline{Q_4}$   
 $\quad \quad \quad \cdot \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0}$

V: 0  
 C: divide by zero,  $C = \overline{X_{15}} \cdot \overline{X_{14}} \cdot \overline{X_{13}} \cdot \overline{X_{12}} \cdot \overline{X_{11}} \cdot \overline{X_{10}} \cdot \overline{X_9} \cdot \overline{X_8} \cdot \overline{X_7} \cdot \overline{X_6} \cdot \overline{X_5} \cdot \overline{X_4}$   
 $\quad \cdot \overline{X_3} \cdot \overline{X_2} \cdot \overline{X_1} \cdot \overline{X_0}$

**Checkpoint 2.6:** Give a single mathematical equation relating the dividend, divisor, quotient, and remainder. This equation gives a unique solution as long as you assume the remainder is strictly less than the divisor.

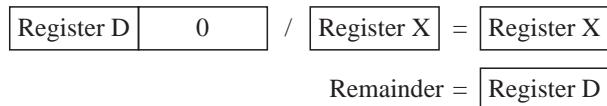
Let N and M be 8-bit unsigned locations. The following assembly code implements  $M=(53*N+50)/100$ , using promotion. Notice that this overall operation can not overflow because the result will be less than or equal to  $(53*255+50)/100=135$ .

```
ldaa N      ;RegA=N  (between 0 and 255)
ldab #53
mul         ;RegD=53*N (between 0 and 13515)
addd #50   ;RegD=53*N+50 (between 0 and 13565)
ldx  #100
idiv       ;RegX=(53*N+50)/100 (between 0 and 135)
xgdx       ;RegB
stab M
```

**Checkpoint 2.7:** Let N and M be 8-bit unsigned locations. Write assembly code to implement  $M=(10*N)/51$ .

The `fdiv` instruction also performs a 16-bit by 16-bit unsigned divide with remainder. In contrast, this instruction calculates  $RegX=(2^{16}*RegD)/RegX$ , as shown in Figure 2.9.  $RegD$  is the remainder.

**Figure 2.9**  
 The `fdiv` instruction takes two 16-bit inputs and generates a 16-bit quotient and a 16-bit remainder.



Condition code bits are set after  $R=(65536*D)/X$ .

Z: result is zero,  $Z = \overline{R_{15}} \cdot \overline{R_{14}} \cdot \overline{R_{13}} \cdot \overline{R_{12}} \cdot \overline{R_{11}} \cdot \overline{R_{10}} \cdot \overline{R_9} \cdot \overline{R_8} \cdot \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4}$   
 $\quad \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

V: overflow if  $RegX$  is less than or equal to  $RegD$ , result >\$FFFF

C: divide by zero,  $C = \overline{X_{15}} \cdot \overline{X_{14}} \cdot \overline{X_{13}} \cdot \overline{X_{12}} \cdot \overline{X_{11}} \cdot \overline{X_{10}} \cdot \overline{X_9} \cdot \overline{X_8} \cdot \overline{X_7} \cdot \overline{X_6} \cdot \overline{X_5} \cdot \overline{X_4}$   
 $\quad \cdot \overline{X_3} \cdot \overline{X_2} \cdot \overline{X_1} \cdot \overline{X_0}$

Let N and M be 16-bit unsigned locations. The following assembly code implements  $M=12.34*N$ . We approximate 12.34 by 65536/5311.

```
ldd  N
ldx  #5311
fdiv       ;RegX=(65536*N)/5311
stx  M
```

**Checkpoint 2.8:** Let N and M be 16-bit unsigned locations. Write assembly code using `fdiv` to implement  $M=2.5*N$ .

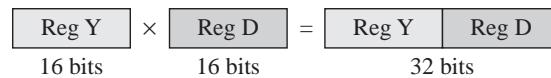
## 2.2.6 Extended Precision Arithmetic Instructions on the 9S12

When designing the 9S12, Freescale added a few instructions not available on the 6811. These instructions are quite useful when implementing mathematical calculations. We will use these operations when designing digital filters and digital controllers, allowing for complex operations to execute quickly.

The `emul` instruction performs a 16-bit by 16-bit unsigned multiply  $\text{RegY:D} = \text{RegY} * \text{RegD}$ , as shown in Figure 2.10. The `emuls` instruction is a 16-bit by 16-bit signed multiply, using the same registers and generating the same condition code bits.

**Figure 2.10**

The `emul` and `emuls` instructions take two 16-bit inputs and generate a 32-bit product.



Condition code bits after  $R=Y*D$

N: result is negative,  $N=R_{31}$

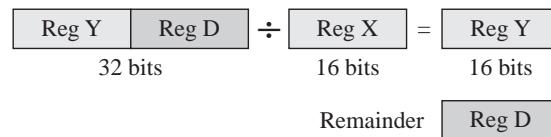
Z: result is zero,  $Z=\overline{R_{31}} \cdot \overline{R_{30}} \cdot \dots \cdot \overline{R_1} \cdot \overline{R_0}$

C:  $R_{15}$ , bit 15 of the result

The `ediv` instruction performs a 32-bit by 16-bit unsigned divide  $\text{RegY}=(Y:D)/\text{RegX}$ ;  $\text{RegD}$  is remainder, as shown in Figure 2.11. The `edivs` instruction is a 32-bit by 16-bit signed divide, using the same registers. The overflow bit calculation is different, but the other three condition code bits are the same.

**Figure 2.11**

The `ediv` and `edivs` instructions perform extended precision division.



Condition code bits after  $R=(Y:D)/X$

N: result is negative (undefined after an overflow or a divide by zero),  $N=R_{15}$

Z: result is zero (undefined after an overflow or a divide by zero)

$$Z = \overline{R_{15}} \cdot \overline{R_{14}} \cdot \overline{R_{13}} \cdot \overline{R_{12}} \cdot \overline{R_{11}} \cdot \overline{R_{10}} \cdot \overline{R_9} \cdot \overline{R_8} \cdot \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \\ \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

V: overflow (undefined after a divide by zero),

`ediv` result  $>\$FFFF$

`edivs` result  $>\$7FFF$  or less than  $-\$8000$

$$C = \overline{X_{15}} \cdot \overline{X_{14}} \cdot \overline{X_{13}} \cdot \overline{X_{12}} \cdot \overline{X_{11}} \cdot \overline{X_{10}} \cdot \overline{X_9} \cdot \overline{X_8} \cdot \overline{X_7} \cdot \overline{X_6} \cdot \overline{X_5} \cdot \overline{X_4} \\ \cdot \overline{X_3} \cdot \overline{X_2} \cdot \overline{X_1} \cdot \overline{X_0}$$

Let N and M be 16-bit unsigned locations. The following assembly code implements  $M=(53*N+50)/100$ , using promotion. Notice that this overall operation cannot overflow because the result will be less than or equal to  $(53*65535+50)/100=34734$ .

```

ldd N      ;RegA=N (between 0 and 65535)
ldy #53
emul      ;RegY:D=53*N (between 0 and 3473355)
addd #50   ;RegD=53*N+50 (between 0 and 3473405)
bcc skip
iny

```

```

skip ldx #100
      ediv      ;RegY=(53*N+50)/100 (between 0 and 34734)
      sty M

```

The `ediv` instruction performs a 16-bit by 16-bit signed multiply, followed by a 32-bit signed addition. It uses indexed addressing to access the two 16-bit inputs and extended addressing to access the 32-bit sum. Recall that {X} and {Y} represent the 16-bit contents pointed to by Registers X and Y, respectively. If we define  $\langle U \rangle$  as the 32-bit contents of memory location U, then `ediv` U calculates

$$\langle U \rangle = \langle U \rangle + \{X\} * \{Y\}$$

Condition code bits, first  $P=X*Y$ , then  $R=M+P$  ( $M,P,R$  are 32 bits)

N: result is negative,  $N=R_{31}$

Z: result is zero,  $Z = \overline{R_{31}} \cdot \overline{R_{30}} \cdot \dots \cdot \overline{R_1} \cdot \overline{R_0}$

V: signed overflow (after addition),  $V = P_{31} \cdot M_{31} \cdot \overline{R_{31}} + \overline{P_{31}} \cdot \overline{M_{31}} \cdot R_{31}$

C: unsigned overflow (after addition),  $C = P_{31} \cdot M_{31} + M_{31} \cdot \overline{R_{31}} + \overline{R_{31}} \cdot P_{31}$

This instruction is quite useful for calculating fixed-point equations, as illustrated in Program 2.2. We place the input variables xx,yy,zz consecutively in RAM, and the constants 902, -1810,45 consecutively in ROM. Registers X and Y are not automatically incremented, so the program performs that task explicitly.

## 2.2.7 Shift Operations

The shift instructions use inherent addressing. The N bit is set if the result is negative. The Z bit is set if the result is zero. The V bit is set on a signed overflow, and is detected by a change in the sign bit. The C bit is the carry out after the shift.

<code>asl a</code>	$RegA=RegA*2$	Signed shift left, same as <code>lsla</code>
<code>asl b</code>	$RegB=RegB*2$	Signed shift left, same as <code>lslb</code>
<code>asl d</code>	$RegD=RegD*2$	Signed shift left, same as <code>lsld</code>
<code>lsla</code>	$RegA=RegA*2$	Unsigned shift left, same as <code>asl a</code>
<code>lslb</code>	$RegB=RegB*2$	Unsigned shift left, same as <code>asl b</code>
<code>lsld</code>	$RegD=RegD*2$	Unsigned shift left, same as <code>asl d</code>
<code>asra</code>	$RegA=RegA/2$	Signed shift right
<code>asrb</code>	$RegB=RegB/2$	Signed shift right
<code>asrd</code>	$RegD=RegD/2$	Signed shift right
<code>lsra</code>	$RegA=RegA/2$	Unsigned shift right
<code>lsrb</code>	$RegB=RegB/2$	Unsigned shift right
<code>lsrd</code>	$RegD=RegD/2$	Unsigned shift right
<code>rol a</code>		Rotate RegA (C A7 ... A0 C)
<code>rol b</code>		Rotate RegB (C B7 ... B0 C)
<code>rora</code>		Rotate RegA (C → A7 → ... → A0 → C)
<code>rorb</code>		Rotate RegB (C → B7 → ... → B0 → C)

When programming in C, the shift is a binary operation. In other words, the `<<` and `>>` operators take two inputs and yield one output, e.g.,  $r=m>>n$ . But at the machine level (i.e., assembly programming), the shift operators are actually unary operations, e.g.,  $r=m>>1$ . The assembly instructions used for shifting will shift one bit at a time. If you want to shift multiple times, you will have to execute the instruction multiple times. The logical shift right (LSR) is the equivalent to an unsigned divide by 2, as shown in Figure 2.12. A zero is shifted into the most significant position, and the carry flag will hold the bit shifted out.

**Program 2.2**

Fixed-point calculation using the 9S12 emac instruction.

```

        org    $3800
; rr=0.902*xx-1.81*yy+0.045*zz
; first we can convert this equation to fixed point
; without introducing any error:
; rr=(902*xx-1810*yy+45*zz)/1000
xx    ds    2
yy    ds    2
zz    ds    2
rr    ds    2      ; result
acc   ds    4      ; temporary 32-bit result
org   $4000
cc    dc.w  902,-1810,45
Calc ldx   #xx    ; pointer to data
ldy   #cc    ; pointer to coefficients
movw #0,acc ; initially clear temporary result
movw #0,acc+2
ldaa #3      ; number of terms
loop emacs acc ; acc=acc+{X}*{Y}
leax 2,x
leay 2,y
dbne A,loop
ldy acc
ldd acc+2 ; Y:D=902*xx-1810*yy+45*zz
ldx #1000
edivs
sty rr
rts

```

**Figure 2.12**

8-bit logical shift right.



The arithmetic shift right (ASR) is the equivalent to a signed divide by 2, as shown in Figure 2.13. Notice that the sign bit is preserved and the carry flag will hold the bit shifted out.

**Figure 2.13**

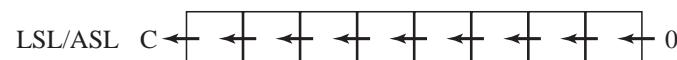
8-bit arithmetic shift right.



The same shift left operation works for both unsigned and signed multiply by 2, as shown in Figure 2.14. In other words, the arithmetic shift left (ASL) is identical to the logical shift left (LSL). A zero is shifted into the least significant position, and the carry bit will contain the bit that was shifted out.

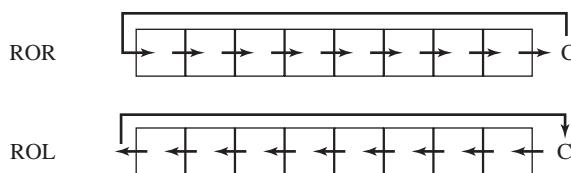
**Figure 2.14**

8-bit shift left.



The **roll** operations can be used to create multiple-byte shift functions. Roll right and roll left are shown in Figure 2.15. In each case, the carry is shifted into the 8-bit byte, and the carry bit will contain the bit that was shifted out.

**Figure 2.15**  
8-bit roll right and 8-bit roll left.



## 2.2.8 Logical Operations

Most 8-bit logical instructions take two inputs, one from a register and the other from memory. All but the `bita` `bitb` instructions put the result back in the register. The N bit will be set if the result is negative. The Z bit will be set if the result is zero. These logical instructions will clear the V bit and leave the C bit unchanged.

<code>anda #w RegA=RegA&amp;w</code>	Logical and RegA with a constant
<code>anda U RegA=RegA&amp;[U]</code>	Logical and RegA with a memory value
<code>andb #w RegB=RegB&amp;w</code>	Logical and RegB with a constant
<code>andb U RegB=RegB&amp;[U]</code>	Logical and RegB with a memory value
<code>bita #w RegA&amp;w</code>	Logical and RegA with a constant
<code>bita U RegA&amp;[U]</code>	Logical and RegA with a memory value
<code>bitb #w RegB&amp;w</code>	Logical and RegB with a constant
<code>bitb U RegB&amp;[U]</code>	Logical and RegB with a memory value
<code>coma</code>	$RegA = \$FF - RegA$ , $RegA = \sim RegA$ Complement RegA
<code>comb</code>	$RegB = \$FF - RegB$ , $RegB = \sim RegB$ Complement RegB
<code>eora #w RegA=RegA ^ w</code>	Exclusive or RegA with a constant
<code>eora U RegA=RegA ^ [U]</code>	Exclusive or RegA with a memory value
<code>eorb #w RegB=RegB ^ w</code>	Exclusive or RegB with a constant
<code>eorb U RegB=RegB ^ [U]</code>	Exclusive or RegB with a memory value
<code>oraa #w RegA=RegA   w</code>	Logical or RegA with a constant
<code>oraa U RegA=RegA   [U]</code>	Logical or RegA with a memory value
<code>orab #w RegB=RegB   w</code>	Logical or RegB with a constant
<code>orab U RegB=RegB   [U]</code>	Logical or RegB with a memory value

Condition code bits are set, where R is the result of the operation.

N: result is negative  $N=R_7$

Z: result is zero  $Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

V: signed overflow  $V=0$

The following C code uses the `shift` and `or` operations to combine two parts into one number. `High` and `Low` are unsigned 4-bit components, which will be combined into a single unsigned 8-bit `Result`. We will assume both `High` and `Low` are bounded within the range of 0 to 15. The expression `High<<4` will perform four logical shift lefts.

```
Result = (High<<4) | Low;
```

The assembly code for this operation is

```
ldaa High ;read value of High
lsla ;shift into position
lsla
lsla
lsla
oraa Low ;combine the two parts together
staa Result ;save answer
```

To illustrate how the foregoing program works, let  $0\ 0\ 0\ 0\ h_3\ h_2\ h_1\ h_0$  be the value of `High`, and let  $0\ 0\ 0\ 0\ l_3\ l_2\ l_1\ l_0$  be the value of `Low`. The `ldaa` instruction brings `High` into register A. The four `lsla` instructions move the `High` into bit positions 4–7, the `oraa`

instruction combines High and Low, and the `staa` instruction stores the combination into Result.

0	0	0	0	$h_3$	$h_2$	$h_1$	$h_0$	value of High
0	0	0	$h_3$	$h_2$	$h_1$	$h_0$	0	after first <code>lsla</code>
0	0	$h_3$	$h_2$	$h_1$	$h_0$	0	0	after second <code>lsla</code>
0	$h_3$	$h_2$	$h_1$	$h_0$	0	0	0	after third <code>lsla</code>
$h_3$	$h_2$	$h_1$	$h_0$	0	0	0	0	after last <code>lsla</code>
0	0	0	0	$l_3$	$l_2$	$l_1$	$l_0$	value of Low
$h_3$	$h_2$	$h_1$	$h_0$	$l_3$	$l_2$	$l_1$	$l_0$	result of the <code>oraa</code> instruction

**Checkpoint 2.9:** Assume PORTB is an output port. Write assembly code that just clears bit 4, leaving the other 7 bits unchanged.

**Checkpoint 2.10:** Assume PORTB is an output port. Write assembly code that just sets bit 3, leaving the other 7 bits unchanged.

## 2.2.9

### Subroutines and the Stack

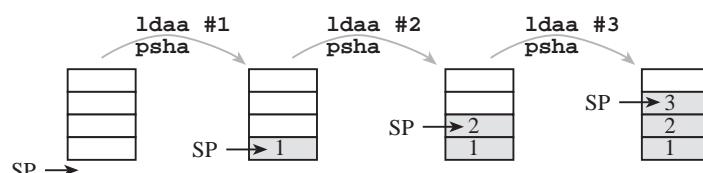
We begin this section with a general description of the stack, and introduce the basic concepts common to most microcontrollers. In general, we initialize the stack pointer (RegSP) into RAM using the `lds` instruction, which is usually done once at the beginning of the program. In the classical definition of the stack, there are just two operations one can perform: **push** and **pull**. Some computers define the two stack operations as push and pop. The push function saves data on the top of the stack, and the pull function removes data from the top of the stack. For example, the `psha` instruction will push the value in RegA onto the stack, leaving RegA unchanged. The `pula` instruction will pull (or pop) a value off the stack bringing it into RegA. The pull operation does modify the stack such that the pulled data is no longer on the stack. The stack implements last in first out (LIFO) behavior. The following code pushes the numbers 1, 2, and 3 in that order.

```
ldaa #1
psha           ; push 1 on the stack
ldaa #2
psha           ; push 2 on the stack
ldaa #3
psha           ; push 3 on the stack
```

After these three push operations, the stack would contain these numbers with the 3 on the top, as shown in Figure 2.16. The top entry of the stack contains the newest data (i.e., the data pushed last). On the 9S12, SP points to the top element.

**Figure 2.16**

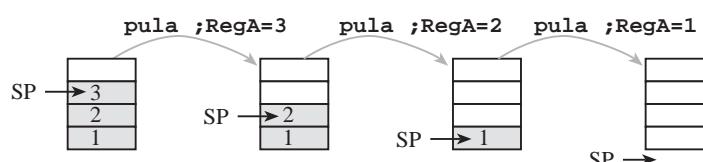
The stack as elements are pushed.



At this point if one were to pull from the stack (e.g., execute `pula`), the 3 would be returned, and 2 would now be on the top of the stack, as shown in Figure 2.17.

**Figure 2.17**

The stack as elements are pulled.



The push and pull instructions use inherent addressing and do not modify the condition code. The push instructions produce two copies of the data, one on the stack and the other still in the register. The pull instructions remove the data from the stack, so there will be only one copy of the data left, which is in the register.

psha	Push RegA on the stack
pshb	Push RegB on the stack
pshx	Push RegX on the stack
pshy	Push RegY on the stack
des	RegSP=RegSP-1 (reserve space on the stack)
pula	Pull value from stack, put in RegA
pulb	Pull value from stack, put in RegB
pulx	Pull value from stack, put in RegX
puly	Pull value from stack, put in RegY
ins	RegSP=RegSP+1 (discard top of stack)

The stack is used for many purposes. A common use is temporary storage. If a piece of information is important, we can push it on the stack. Later, when we wish to retrieve the data, we pull it off the stack.

**Checkpoint 2.11:** Assume you have two 8-bit global variables M and N. Write assembly code that switches the values in M and N using just the `ldaa staa psha` and `pula` instructions.

There are two additional stack operations called **stack read** and **stack write**. The stack read operation allows you to retrieve data previously pushed on the stack without modifying the data or the stack pointer. The stack write operation allows you to change a value previously pushed on the stack without changing the stack pointer. Even though these two stack operations are not part of the classical definition of a stack, they will be essential for implementing parameter passing and local variables. The following are important instructions that greatly facilitate the use of the stack. The instruction `tsx` will move a copy of the stack pointer into Register X.

tsx	Transfer RegSP to RegX
tsy	Transfer RegSP to RegY
txs	Transfer RegX to RegSP
tys	Transfer RegY to RegSP

Procedures and functions are programs that can be called to perform specific tasks. Some programming environments differentiate functions (return a value) from procedures (do not return a value). In assembly language, we use the term **subroutine** for all subprograms whether or not they return a value. Subroutines allow us to develop modular software. In assembly language, we will use either the `bsr` or `jsr` instruction to call a subroutine, and we will use the `rts` instruction to return from the subroutine. The `bsr` and `jsr` instructions will push the return address on the stack. The return address is the address of the instruction immediately after the branch to subroutine instruction. The `rts` will pull the return address from the stack, returning the program to the place from which the subroutine was called.

**Observation:** Since the `bsr` instruction uses relative addressing, it can only be used to call a subroutine near the current instruction. Since the `jsr` instruction allows extended addressing, it can be used to call a subroutine anywhere in memory.

We will study the concept of subroutines using the simple example shown in Program 2.3. The input parameter to the subroutine is passed in using RegA. The subroutine adds one and returns the result also in RegA. The main program calls the subroutine using the `bsr` instruction. The subroutine returns back to the main program using the `rts` instruction. The numbers on the right specify the sequence of execution for the first three times through the loop.

**Program 2.3**

Simple program showing how to use the `bsr` and `rts` instructions to implement a subroutine.

```

        org $4000
;*****Add1*****
;Purpose: Add one from RegA
; Input: RegA
;Output: RegA=Input+1
Add1 inca      ;adds one.....4 8 12
            rts      ;.....5 9 13
main lds #$_4000 ;.....1
            clra    ;.....2
loop bsr Add1   ; call .....3 7 11
            bra loop ;.....6 10 14
            org $FFE
            fdb main

```

**Program 2.4**

9S12 assembly listing of Program 2.3.

```

$4000          org $4000
;*****Add1*****
;Purpose: Add one from RegA
; Input: RegA
;Output: RegA=Input+1
$4000 42      [1]{O }Add1 inca      ; adds one
$4001 3D      [5]{UfPPP }      rts
$4002 CF4000  [2]{OP }main lds #$_4000
$4005 87      [1]{O }      clra
$4006 07F8  [4]{PPPS }loop bsr Add1 ; call
$4008 20FC  [3]{PPP }      bra loop
$FFFE          org $FFE
$FFFE 4002      fdb main

```

We begin the study by looking at the listing file generated by the assembler, shown as Program 2.4.

Figure 2.18 shows the stack before and after the `bsr` instruction is executed. During the first two cycles, it fetches the opcode and operand. At this point the PC is 4008, which will be the return location. The effective address (4000) is calculated. The last two cycles push the return address on the stack.

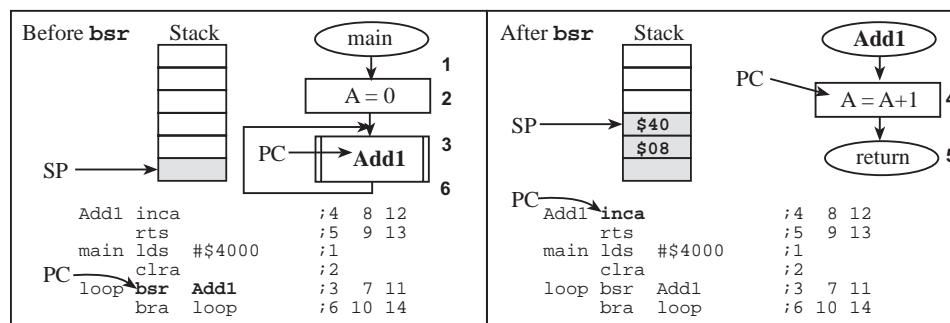
```

Opcode fetch R 0x4006 0x07 from EEPROM
Operand fetch R 0x4007 0xF8 from EEPROM
Stack store lsbW 0x3FFF 0x08 to RAM
Stack store msbW 0x3FFE 0x40 to RAM

```

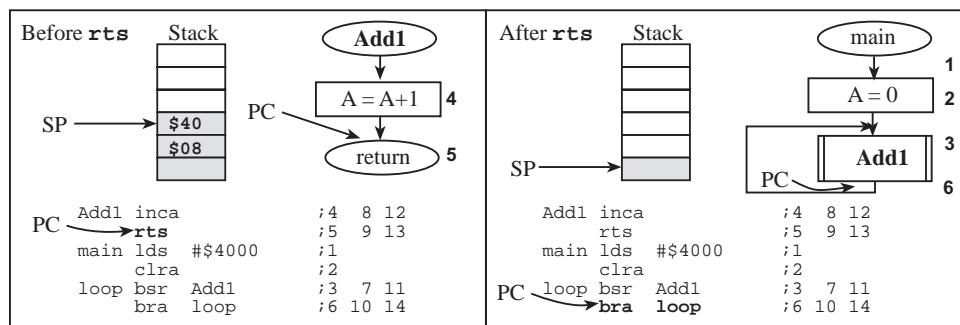
**Figure 2.18**

The stack before and after execution of the `bsr` instruction.



**Figure 2.19**

The stack before and after execution of the `rts` instruction.



The `rts` instruction will return to the program that called the subroutine. Figure 2.19 shows the stack before and after the `rts` instruction is executed. During the first cycle it fetches the opcode. The last two cycles pull the return address from the stack.

```
Opcode fetch R 0x4001 0x3D from EEPROM
Stack read msb R 0x3FFE 0x40 from RAM
Stack read lsb R 0x3FFF 0x08 from RAM
```

## 2.2.10 Branch Operations

Normally the computer executes one instruction after another in a linear fashion. In particular, the next instruction to execute is found immediately following the current instruction. We use branch instructions to deviate from this straight line path.

<code>bcc target</code>	Branch to target if C=0
<code>bcs target</code>	Branch to target if C=1
<code>beq target</code>	Branch to target if Z=1
<code>bne target</code>	Branch to target if Z=0
<code>bmi target</code>	Branch to target if N=1
<code>bpl target</code>	Branch to target if N=0
<code>bra target</code>	Branch to target always
<code>brn target</code>	Branch to target never
<code>bvc target</code>	Branch to target if V=0
<code>bvs target</code>	Branch to target if V=1
<code>jmp target</code>	Branch to target always, extended addressing

The following branch instructions must follow a subtract compare or test instruction, such as `suba` `subb` `sbca` `sbcu` `subd` `cba` `cmpa` `cmpb` `cpd` `cpx` `cpy` `tsta` `tstb` `tst`.

<code>bge target</code>	Branch if signed greater than or equal to, if $(N \wedge V) = 0$ , or $(\sim N \cdot V + N \cdot \sim V) = 0$
<code>bgt target</code>	Branch if signed greater than, if $(Z + N \wedge V) = 0$ , or $(Z + \sim N \cdot V + N \cdot \sim V) = 0$
<code>ble target</code>	Branch if signed less than or equal to, if $(Z + N \wedge V) = 1$ , or $(Z + \sim N \cdot V + N \cdot \sim V) = 1$
<code>blt target</code>	Branch if signed less than, if $(N \wedge V) = 1$ , or $(\sim N \cdot V + N \cdot \sim V) = 1$
<code>bhs target</code>	Branch if unsigned greater than or equal to, if C=0, same as <code>bcc</code>
<code>bhi target</code>	Branch if unsigned greater than, if C+Z=0
<code>blo target</code>	Branch if unsigned less than, if C=1, same as <code>bcs</code>
<code>bls target</code>	Branch if unsigned less than or equal to, if C+Z=1

Conditional execution is an important aspect of software programming. Two values are compared, and certain blocks of program are executed or skipped depending on the results of the comparison. In assembly language it is important to know the precision (e.g., 8-bit, 16-bit) and the format of the two values (e.g., unsigned, signed). It takes three steps to perform a comparison. You begin by reading the first value into a register. For 8-bit values you can use either Register A or Register B. 16-bit values can be loaded into Register D, Register X, or Register Y. The second step is to compare the first value with the second value. You can use either a subtract instruction (`suba` `subb` `subd`) or a compare instruction (`cmpa` `cmpb` `cpd` `cpx` `cpx`). These instructions set the condition code bits. The last step is a conditional branch. Table 2.4 lists some simple comparisons. When testing for equal, or not equal, it doesn't matter whether the numbers are signed or unsigned. In the following examples, we assume `G1`, `G2` are 8-bit variables.

**Table 2.4**

Conditional structures that test for equality.

C Code	Assembly Code
<pre>if(G2 == G1){     isEqual(); }</pre>	<pre>ldaa G2 cmpa G1 bne next      ;skip if not equal jsr isEqual   ;G2==G1 next</pre>
<pre>if(G2 != G1){     isNotEqual(); }</pre>	<pre>ldaa G2 cmpa G1 beq next      ;skip if equal jsr isNotEqual ;G2!=G1 next</pre>

**Common error:** It is an error to use an 8-bit comparison to test two 16-bit values.

When testing for greater than or less than, it does matter whether the numbers are signed or unsigned. Table 2.5 lists some 8-bit unsigned comparisons. When comparing unsigned values, the instructions `bhi` `blo` `bhs` and `bls` should follow the subtraction or comparison instruction. To convert these examples to 16 bits, change the `ldaa G2` to `ldd H2` and the `cmpa G1` to `cpd H1`.

When comparing signed values, the instructions `bgt` `bls` `bge` and `ble` should follow the subtraction or comparison instruction.

**Checkpoint 2.12:** When implementing `if(N>25)isGreater();` why is it important to know whether `N` is signed or unsigned?

**Common error:** It is an error to use an unsigned conditional branch when comparing two signed values. Similarly, it is a mistake to use a signed conditional branch when comparing two unsigned values.

Quite often the microcomputer is asked to wait for events or to search for objects. Both of these operations are solved using the `while` or `do-while` structure. The `while` loop, illustrated in the Figure 2.20, will wait until Port A bit 0 equals 1. The operation is defined by the C code

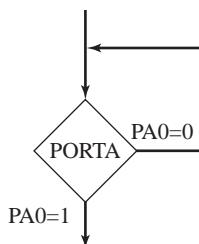
```
while( (PORTA&0x01)==0 ) { }
```

The program begins with reading Port A. Bit 0 is selected using the logical *and* function. Selecting certain bits with the *and* function is called **masking**. The `anda` instruction sets

**Table 2.5**  
Unsigned 8-bit  
conditional structures.

C Code	Assembly Code
<pre>if(G2 &gt; G1){     isGreater(); }</pre>	<pre>ldaa G2 cmpa G1 bls next      ;skip if G2&lt;=G1 jsr isGreater ;G2&gt;G1 next</pre>
<pre>if(G2 &gt;= G1){     isGreaterEq(); }</pre>	<pre>ldaa G2 cmpa G1 blo next      ;skip if G2&lt;G1 jsr isGreaterEq ;G2&gt;=G1 next</pre>
<pre>if(G2 &lt; G1){     isLess(); }</pre>	<pre>ldaa G2 cmpa G1 bhs next      ;skip if G2&gt;=G1 jsr isLess    ;G2&lt;G1 next</pre>
<pre>if(G2 &lt;= G1){     isLessEq(); }</pre>	<pre>ldaa G2 cmpa G1 bhi next      ;skip if G2&gt;G1 jsr isLessEq  ;G2&lt;=G1 next</pre>

**Figure 2.20**  
Flowchart of a while  
structure.



the Z bit, which is utilized by the `beq` instruction. The software will loop over and over while PA0 equals zero. The operation in assembly is

```
loop ldaa PORTA
      anda #$01 ;look at just PA0
      beq loop ;continue when PA0 is 1
```

### 2.2.11 Assembler Pseudo-ops

Pseudo-ops are specific commands to the assembler that are interpreted during the assembly process. An alternative name for pseudo-op is **assembly directive**. A few of them create object code, but most do not. There are many assemblers available developing 9S12 assembly code. Although they all use the standard Freescale opcodes, the spelling of the pseudo-op codes varies. The **TEXaS** assembler supports many of the various dialects. The pseudo-op codes supported by this assembler are shown in Table 2.6. If you plan to export software developed with **TEXaS** to another application, then you should limit your use only the pseudo-ops compatible with that application. Group A is supported by Freescale's MCUEz, and HiWare (now Metrowerks). Group B is supported by Freescale's DOS level AS11 and AS12. Group C are used by ImageCraft's ICC11 and ICC12.

**Table 2.6**

Assembly directives supported by **TEXaS**.

Group A	Group B	Group C	Meaning
org	org	.org	Specific absolute address to put subsequent object code
=	equ		Define a constant symbol
	set		Define or redefine a constant symbol
dc.b	db	.byte	Allocate byte(s) of storage with initialized values
	fcb		Create an ASCII string (no termination character)
dc.w	dw	.word	Allocate word(s) of storage with initialized values
dc.l	dl	.long	Allocate 32-bit long word(s) of storage with initialized values
ds	ds.b	.blk	Allocate bytes of storage without initialization
ds.w		.blkw	Allocate bytes of storage without initialization
ds.l		.blk1	Allocate 32-bit words of storage without initialization
end	end	.end	Signifies the end of the source code ( <b>TEXaS</b> ignores these)

### Equate symbol to a value

```
<label> equ <expression> (<comment>)
<label> = <expression> (<comment>)
```

The `equ` (or `=`) directive assigns the value of the expression in the operand field to the label; see Program 2.5. The label cannot be redefined anywhere else in the program. The expression cannot contain any forward references or undefined symbols. Equates with forward references are flagged as a phasing error.

### Program 2.5

A constant implemented with `equ` or `#define` will make the program easier to change.

<pre>org \$3800 SIZE equ 5 Data rmb SIZE org \$4000 Sum ldaa #SIZE ldx #Data clrb loop addb 1,x+ dbne A,loop rts</pre>	<pre>#define SIZE 5 unsigned char Data[SIZE]; unsigned char Sum(void){     unsigned char i,total;     total = 0;     for(i=0; i&lt;SIZE; i++){         total = total + Data[i];     }     return total; }</pre>
--	---

The `equ` pseudo-op is used to define the I/O ports and to access the elements of a data structure.

**Programming tip:** Use `equ` definitions only if it makes the program easier to understand, to debug, or to change.

### Redefinable equate symbol to a value

```
<label> set <expression> (<comment>)
```

The `set` directive assigns the value of the expression in the operand field to the label. The `set` directive assigns a value other than the program counter to the label. Unlike the `equ` pseudo-op, the label can be redefined within the program. Although allowed, it is probably a mistake to use forward references. The use of this pseudo-op with forward references will not be flagged with a phasing error. Local variable names created with the `set` directive could be reused in another subroutine. More information about local variables can be found in Section 2.5.1.

### Form constant byte

```
(<label>) fcb <expr>(<expr>, . . . , <expr>) (<comment>)
(<label>) dc.b <expr>(<expr>, . . . , <expr>) (<comment>)
(<label>) db <expr>(<expr>, . . . , <expr>) (<comment>)
(<label>) .byte <expr>(<expr>, . . . , <expr>) (<comment>)
```

The `fcb` directive may have one or more operands separated by commas. The value of each operand is truncated to eight bits and is stored in a single byte of the object program. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one or more of them can be null (two adjacent commas), in which case a single byte of zero will be assigned for that operand. If an operand is larger than the range of an 8-bit number ( $-128$  to  $+255$ ), the result is truncated without a warning, and the least significant 8 bits are used.

A string can be included, which is stored as a sequence of ASCII characters. The delimiters supported by **TExaS** are `"` and `\`. The string does not include a null-termination, so if desired, the programmer must explicitly terminate it. The following three examples produce identical null-terminated strings.

```
str1 fcb "Hello World",0
str2 fcb 'Hello World',0
str3 fcb \Hello World\,0
```

The stepper motor controller shown in Program 2.6 uses the `fcb` definitions to store the four stepper motor output values.

### Program 2.6

A stepper motor controller using `fcb`.

<pre>SIZE equ 4 org \$4000 Steps fcb 5,6,10,9 ;out sequence main bset DDRT,#\$0F ;PT3-0 outputs run ldAA #SIZE       ldx #Steps step movB 1,x+,PTT ;step motor       dbne A,step       bra run       org \$FFFE       fdb main</pre>	<pre>#define SIZE 4 unsigned char const Steps[SIZE] = {5,6,10,9}; void main(void){unsigned char i;   DDRT  = 0x0F;   while(1){     for(i=0; i&lt;SIZE; i++){       PTT = Steps[i];     }   } }</pre>
--	--

### Form constant character string

```
(<label>) fcc <delimiter><string><delimiter> (<comment>)
```

The `fcc` directive is used to store ASCII strings into consecutive bytes of memory. The byte storage begins at the current program counter. The label is assigned to the address of the first byte in the string. Any of the printable ASCII characters can be contained in the string. The string is specified between two identical delimiters. The first non-blank character after the `fcc` directive is used as the delimiter. The delimiters supported by **TExaS** are `"` and `\`. Examples:

```
LABEL1 FCC  'ABC'
LABEL2 fcc  "Jon Valvano"
LABEL4 fcc  /Welcome to FunCity!/
```

The first line creates the ASCII characters **ABC** at location `LABEL1`. Be careful to position the `fcc` code away from executable instructions. The assembler will produce object code as it would for regular instructions, one line at a time. For example, the following would

crash because after executing the `ldx` instruction, the microcontroller would try to execute the ASCII characters "Trouble" as instructions.

```
ldaa 100
ldx #Strg
Strg fcc "Trouble"
```

Typically we collect all the `fcc`, `fcb`, `fdb` together and place them at the end of our program, so that the microcomputer does not try to execute the constant data. The ASCII string generated by `fcc` is not null-terminated, so if a termination is needed, you must add it explicitly using either

```
Strg1 fcc "happy"
fcb 0
```

or

```
Strg2 fcb "happy",0
```

### Form double byte

```
(<label>) fdb <expr>(<expr>, <expr>, ..., <expr>) (<comment>)
(<label>) dc.w <expr>(<expr>, <expr>, ..., <expr>) (<comment>)
(<label>) dw <expr>(<expr>, <expr>, ..., <expr>) (<comment>)
(<label>) .word <expr>(<expr>, <expr>, ..., <expr>) (<comment>)
```

The `fdb` directive may have one or more operands separated by commas. The 16-bit value corresponding to each operand is stored into two consecutive bytes of the object program (big endian). The storage begins at the current program counter. The label is assigned to the address of the first 16-bit value. Multiple operands are stored in successive 16-bit words. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one or more of them can be null (two adjacent commas), in which case two bytes of zeros will be assigned for that operand. The `fdb` has been used many times so far in the book to define the reset vector.

### Define 32-bit constant

```
(<label>) dc.l <expr>(<expr>, <expr>, ..., <expr>) (<comment>)
(<label>) dl <expr>(<expr>, <expr>, ..., <expr>) (<comment>)
(<label>) .long <expr>(<expr>, <expr>, ..., <expr>) (<comment>)
```

The `dl` directive may have one or more operands separated by commas. The 32-bit value corresponding to each operand is stored into four consecutive bytes of the object program (big endian). The storage begins at the current program counter. The label is assigned to the address of the first 32-bit value. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one or more of them can be null (two adjacent commas), in which case four bytes of zeros will be assigned for that operand. In the following finite state machine, the `dl` definitions are used to define 32-bit constants.

```
S1 dl 100000,$12345678
S2 .long 1,10,100,1000,10000,100000,1000000,10000000
S3 dc.l -1,0,1
```

### Set program counter origin

```
org<expression> (<comment>)
.org<expression> (<comment>)
```

The `org` directive changes the program counter to the value specified by the expression in the operand field. Subsequent statements are assembled into memory locations

starting with the new program counter value. If no `org` directive is encountered in a source program, the program counter is initialized to zero. Expressions cannot contain forward references or undefined symbols. The `org` statements in Programs 2.5 and 2.6 place the variables in RAM and the programs in EEPROM. The `org` statement is also used to set the reset vector.

### Reserve multiple bytes

```
(<label>) rmb <expression> (<comment>)
(<label>) ds <expression> (<comment>)
(<label>) ds.b <expression> (<comment>)
(<label>) .blk8 <expression> (<comment>)
```

The `rmb` directive causes the location counter to be advanced by the value of the expression in the operand field. This directive reserves a block of memory the length of which in bytes is equal to the value of the expression. The block of memory reserved is not initialized to any given value. The expression cannot contain any forward references or undefined symbols. This directive is commonly used to reserve a scratchpad or table area for later use.

**Checkpoint 2.13:** Why can't you use a forward reference in an `rmb` directive?

### Reserve multiple words

```
(<label>) ds.w <expression> (<comment>)
(<label>) .blkw <expression> (<comment>)
```

The `ds.w` directive causes the location counter to be advanced by 2 times the value of the expression in the operand field. This directive reserves a block of memory the length of which in words (16-bit) is equal to the value of the expression. The block of memory reserved is not initialized to any given value. The expression cannot contain any forward references or undefined symbols. This directive is commonly used to reserve a scratchpad or table area for later use.

### ds.l Reserve multiple 32-bit words

```
(<label>) ds.l <expression> (<comment>)
(<label>) .blk1 <expression> (<comment>)
```

The `ds.l` directive causes the location counter to be advanced by 4 times the value of the expression in the operand field. This directive reserves a block of memory the length of which in words (32-bit) is equal to the value of the expression. The block of memory reserved is not initialized to any given value. The expression cannot contain any forward references or undefined symbols. This directive is commonly used to reserve a scratchpad or table area for later use.

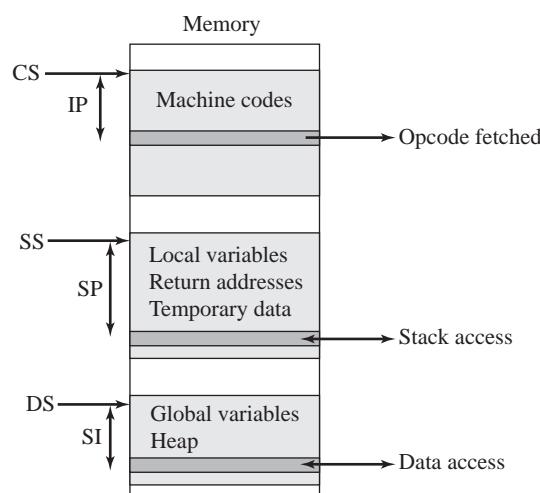
## 2.2.12 Memory Allocation

Memory allocation is the decision of where in memory we put the various pieces of our software. The memory on a PC-compatible computer is physically configured as a simple linear array. In other words, if you have 256 mebibytes of RAM, then this memory exists as a continuous linear object with no fundamental difference in the behavior of one memory cell to the next. Although the memory itself forces no structure in the way it is used, the Intel x86 processors have implemented a memory access scheme that requires the programmer to separate in memory segments (e.g., machine codes, global variables, and local variables). The term **x86** refers to any Intel processor from the 8086 through the current Pentiums. There can be more than three segments, but three are enough to illustrate the point. The mechanism to access these segments is called **segmentation**. Figure 2.21 shows a simple view of the memory allocation on the Intel x86 family.

In particular, when the Pentium fetches a machine code, it uses two registers. The code segment selector (CS) points to the beginning of the code segment, and the instruction

**Figure 2.21**

The Intel x86 uses segmented memory allocation.

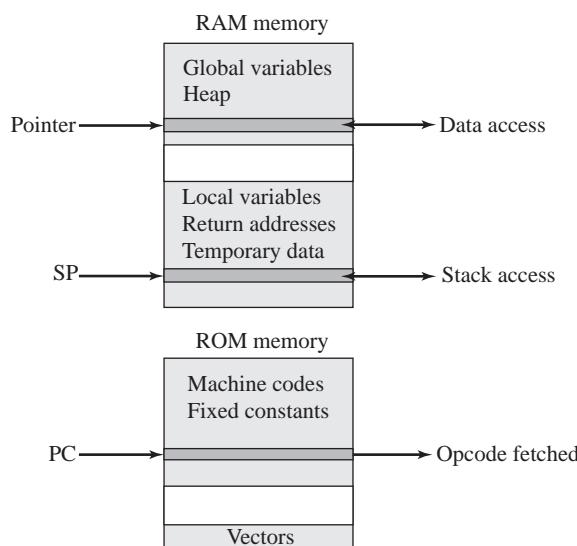


pointer (IP) contains the offset within this segment of the opcode to fetch. Similarly, when the Pentium accesses a global variable, it uses two different registers. The data segment selector (DS) points to the beginning of the data segment, and a data pointer (i.e., DI) contains the offset within this segment of global variable. Lastly, when the Pentium accesses a local variable, it uses a stack segment selector (SS) and either the stack pointer (SP) or the base pointer (BP). The stack segment selector (SS) points to the beginning of the stack segment, and a stack pointer (SP or BP) contains the offset within this segment of local variable. Segmentation forces the programmer to allocate in memory information that has similar properties. In other words, all the machine codes are placed in one group, the global variables are in another group, and the stack is in a third group. This allocation scheme provides for protection so that the errors of stack overflow, stack underflow, or accessing an illegal pointer do not modify machine codes.

We will allocate memory on our embedded system in a fashion similar to segmentation but for a different reason. Because different types of memory on an embedded computer behave in different fashions, it makes sense to group together in memory information that has similar properties or usage. Typical examples of this grouping include global variables, the heap, local variables, fixed constants, and machine instructions. Figure 2.22 shows a typical memory allocation scheme for an embedded system.

**Figure 2.22**

We place variables in RAM and programs in ROM on an embedded system.



Global variables are permanently allocated and are usually accessible by more than one program. We must use global variables for information that must be permanently available, or for information that is to be shared by more than one module. We will see many applications later in this book of the first-in-first-out (FIFO) queue, a global data structure that is shared by more than one module. Some software systems use a heap to dynamically allocate and release memory (e.g., **malloc**). This information can be shared or not shared depending on which modules have pointers to the data. The heap is efficient in situations where storage is needed for only a limited amount of time. Local variables are usually allocated on the stack at the beginning of the subroutine/function, used within the subroutine/function, and deallocated at the end of the subroutine/function. Local variables are not shared with other modules. Fixed constants do not change and include information such as numbers, strings, sounds, and pictures. Just as with the heap, the fixed constants can be shared, or not shared, depending on which modules have pointers to the data. As we saw in the previous chapter, the assembler or compiler translates our software into machine instruction (opcodes and operands) that when executed perform the intended operations. For single-chip microcomputers, there are three types of memory. The RAM contains temporary information that is lost when the power is shun off (i.e., volatile.) This means that all variables allocated in RAM must be explicitly initialized at run time by the software. Most C compilers initialize all RAM-based global variables to zero, but others do not. It is good software development practice to set globals to the desired initial value explicitly. As we saw in the previous chapter, many Freescale microcomputers have flash EEPROM. The ROM is a low-cost nonvolatile storage that can be programmed only once.

In an embedded application, we must put global variables, the heap, and local variables in RAM because these types of information can change during execution. When software is to be executed on a regular computer, the machine instructions are usually read from a mass storage device (e.g., a disk) and loaded into memory. Because the embedded system usually has no mass storage device, the machine instructions and fixed constants must be stored in nonvolatile memory. If there is both EEPROM and ROM on our microcomputer, we put some fixed constants in EEPROM and some in ROM. If it is information that we may wish to change in the future, we could put it in EEPROM. Examples include language-specific strings, calibration constants, finite-state machines, and system ID numbers. This allows us to make minor modifications to the system by reprogramming the EEPROM without throwing the chip away. If our project involves producing a small number of devices, then the machine instructions can be placed in EPROM or flash EEPROM. For a project with a large volume it will be cost-effective to place the machine instructions in ROM.

Program 2.7 is a simple illustration of how we allocate various sections of our software using the **org** pseudo-op. The program outputs to Port T on the sequence 0,5,10,15, . . . .

### Program 2.7

Memory allocation places variables in RAM and programs in ROM.

<pre>         org \$3800 ;RAM Count rmb 1      ;global         org \$4000 ;EEPROM Five  fcb 5      ;amount to add init  movb #\$FF,DDRT ;outputs       clr Count       rts main  lds #\$4000 ;sp=&gt;RAM       bsr init loop   ldaa Count       staa PTT    ;output       adda Five       staa Count       bra loop         org \$FFE ;EEPROM fdb   main ;reset vector </pre>	<pre> unsigned char Count; unsigned char const Five=5; void init(void){   DDRT = 0xFF;   Count = 0; } void main(void){   init();   while(1){     PTT = Count;     Count = Count+Five;   } } </pre>
---	--

The global variable is placed at the start of the RAM, the stack is initialized to the top of RAM (and grows down), and the program is placed in ROM. The constants are placed in the EEPROM.

## 2.3 Self-Documenting Code

**2.3.1 Comments** The goal of this section is to present ideas concerning software documentation in general and writing comments in particular. Maintaining software is the process of fixing bugs, adding new features, optimizing for speed or memory size, porting to new computer hardware, and configuring the software system for new situations. Maintenance is the *most important* phase of software development. Documentation should therefore assist software maintenance. In many situations the software is not static but is continuously undergoing changes. Because of this liquidity, we believe that flowcharts and software manuals are not good mechanisms for documenting programs because it is difficult to keep these types of documentation up to date when modifications are made. Therefore, the term *documentation* in this book refers almost exclusively to comments that are included in the software itself. There are two types of readers of our comments. Our client is someone who will use our software, incorporating it into a larger system. Client comments focus on the *policies* of the software. What are the possible valid inputs? What are the resulting outputs? What does the software do? How does one call the software? What are the error conditions? The other reader of our comments is a colleague, who is someone charged with software maintenance. Colleague comments focus on the *mechanisms* of the software. How does it work? What algorithms are used? How was the software tested? How does one change the software? If it is not written down, then it doesn't exist.

**Observation:** The simulator, TExaS, that accompanies this book is one of the few regular software development applications that allows you to add color drawings into the comment fields.

As software developers, our goal is to produce code that not only solves our current problem but can serve as the basis of solutions to our future problems. To reuse software we must leave our code in a condition such that future programmers (including ourselves) can easily understand its purpose, constraints, and implementation. Documentation is not something tacked onto software after it is done, but rather it is a discipline built into it at each stage of the development. Writing comments as we develop the software forces us to think about what the software is doing and more importantly why we are doing it. Therefore, we should carefully develop a programming style that provides appropriate comments. A comment that tells us why we perform certain functions is more informative than comments that tell us what the functions are. We should assume the reader of our comment already knows the syntax of the language. The following are examples of bad comments because they provide no additional information:

```
X=X+4;      /* add 4 to X */  
Flag=0;      /* set Flag=0 */
```

**Common error:** A comment that simply restates the operation does not add to the overall understanding.

**Common error:** Putting a comment on every line of software often hides the important information.

Good comments assist us now while we are debugging and will assist us later when we are modifying the software, adding new features, or using the code in a different context.

The following are examples of good comments because they explain why the function is being executed:

```
X=X+4; /* 4 is added to correct for the offset (mV) in the transducer */
Flag=0; /* means no key has been typed */
```

When a variable is defined, we should add comments to explain how the variable is used. If the variable has units, then it is appropriate to include them in the comments. It may be relevant to specify the minimum and maximum values. A typical value and what it means often will clarify the usage of the variable. For example:

```
short SetPoint;
/* The desired temperature for the temperature control system
16-bit signed temperature with a resolution of 0.5C, a range of -55C to +125C
a value of 25 means 12.5C, a value of -25 means -12.5C */
```

When a constant is used, we could add comments to explain what the constant means. If the number has units, then it is appropriate to include them in the comments. For example:

```
V=999; /* 999mV is the maximum possible voltage */
err=1; /* error code of 1 means out of range */
```

When a subroutine or function is defined, there will be two types of comments. The first type is directed to the user of the routine (client). These comments explain how the function is to be used, how to pass parameters, what sort of errors might happen, and how the results are returned. If we are writing in C language, then these comments should be included in the \*.h file along with the function prototypes. If we are writing in assembly language, then these comments should be included at the beginning of the subroutine. If the parameters have units, then it is appropriate to include them in the comments. Just like a variable, it may be relevant to specify the minimum and maximum values for the I/O parameters. Typical I/O values and what they mean often will clarify the usage of the function. Frequently we give entire software examples showing how the functions could be used. The second type of comments is directed to the programmer responsible for debugging and software maintenance (colleague). These comments explain how the function works. Generally we separate these comments from the ones intended for the user of the function. This separation is the first of many examples in this book of the concept “separation of policy from mechanism.” The policy is what the function does, and the mechanism is how it works. Specifically, we place this second type of comments within the body of the function. If we are writing in C, then these comments should be included in the \*.c file along with the function implementation.

Self-documenting code is software written in a simple and obvious way such that its purpose and function are self-apparent. Descriptive names for variables, constants, and functions will go a long way to clarify their usage. To write wonderful code like this, we first must formulate the problem, organizing it into clear, well-defined subproblems. How we break a complex problem into small parts goes a long way toward making the software self-documenting. The concepts of abstraction, modularity, and layered software, all presented later in this chapter, address this important issue of software organization.

**Observation:** The purpose of a comment is to assist in debugging and maintenance.

We should use careful indenting, and descriptive names for variables, functions, labels, and I/O ports. Liberal use of C language `#define` and assembly language `equ` provide explanation of software function without cost of execution speed or memory requirements. A disciplined approach to programming is to develop patterns of writing that you consistently follow. Software developers are not like short story writers. When writing software

it is a good design practice to use the same *function outline* over and over again. In the programs of this chapter, notice the following assembly language style issues:

1. Begins and ends with a line of \*s
2. States the purpose of the function
3. Gives the I/O parameters, what they mean, and how they are passed
4. Different phases (submodules) of the code delineated by a line of -'s

**Observation:** It is better to write clear and simple software that is easy to understand without comments than to write complex software that requires a lot of extra explanation to understand.

#define statements, if used properly, can clarify our software and make our software easy to change. Notice in Program 2.8, that if one changes the value of `size`, then a bug would occur in `initialize`.

### Program 2.8

An inappropriate use of  
#define.

```
#define size 10
short data[size];
void initialize(void){ short j;
    for(j=0;j<10;j++)
        data[j]=0;
}
```

It is proper to use `size` in all places that refer to the size of the data array (Program 2.9).

### Program 2.9

An appropriate use of  
#define.

```
#define size 10
short data[size];
void initialize(void){ short j;
    for(j=0;j<size;j++)
        data[j]=0;
}
```

**Common error:** A programmer may employ a #define or equ for the sole purpose of making the software easier to read, and a software bug may occur if you change the value of the constant.

Software documentation is an important communication tool between software developers. It also provides invaluable information for software that will be modified in the future. The approach to good documentation is to provide information that enhances understanding, use, and modification. It is good practice to tailor documentation specifically to the intended reader. For example, we might give different information to a user of our module (programmer writing code that calls our module) than to a developer (programmer responsible for testing and upgrading). Clearly state in the comments:

Purpose of the module

Input parameters

    How passed (call by value, call by reference)

    Appropriate range (does the module assume the input is within range?)

    Format (8 bit/16 bit, signed/unsigned, etc.)

Output parameters

    How passed (return by value, return by reference)

    Format (8 bit/16 bit, signed/unsigned, etc.)

Example inputs and outputs if appropriate

Error conditions

Example calling sequence

Local variables and their significance

### 2.3.2 Naming Convention

Choosing names for variables and functions involves creative thought, and it is intimately connected to how we feel about ourselves as programmers. Of the policies presented in this section, our naming conventions may be the hardest habit for us to break. The difficulty is that there are many conventions that satisfy the “easy to understand” objective. Good names reduce the need for documentation. Poor names promote confusion, ambiguity, and mistakes. Poor names can occur because code has been copied from a different situation and inserted into our system without proper integration (i.e., changing the names to be consistent with the new situation). They can also occur in the cluttered mind of a second-rate programmer, who hurries to deliver software before it is finished.

*Names should have meaning.* If we observe a name out of the context of the program in which it exists, the meaning of the object should be obvious. The object `TxFifo` is clearly the transmit first-in-first-out circular queue. The function `LCD_OutString` will output a string to the LCD display.

*Avoid ambiguities.* Don’t use variable names in our system that are vague or have more than one meaning. For example, it is vague to use `temp`, because there are many possibilities for temporary data—in fact it might even mean temperature. Don’t use two names that look similar but have different meanings.

*Give hints about the type.* We can further clarify the meaning of a variable by including phrases in the variable name that specify its type. For example, `dataPt timePt putPt` are pointers. Similarly, `voltageBuf timeBuf pressureBuf` are data buffers. Other good phrases include `Flag Mode U L Index Cnt`, which refer to Boolean flag, system state, unsigned 16-bit, signed 32-bit, index into an array, and a counter, respectively.

*Use the same name to refer to the same type of object.* For example, everywhere we need a local variable to store an ASCII character we could use the name `letter`. Another common example is to use the names `i j k` for indices into arrays. The names `V1 R1` might refer to a voltage and a resistance. The exact correspondence is not part of the policies presented in this section, just that a correspondence should exist. Once another programmer learns which names we use for which types of object, understanding our code becomes easier.

*Use a prefix to identify public objects.* An underline character will separate the module name from the function name. As an exception to this rule, we can use the underline to delimit words in all upper-case name (e.g., `#define MIN_PRESSURE 10`). Functions that can be accessed outside the scope of a module will begin with a prefix specifying the module to which it belongs. It is poor style to create public variables, but if they need to exist, they too would begin with the module prefix. The prefix matches the file name containing the object. For example, if we see a function call, `LCD_OutString("Hello world")`; we know the public function belongs to the LCD module, where the policies are defined in `LCD.h` and the implementation in `LCD.c`. Notice the similarity between this syntax (e.g., `LCD_init()`) and the corresponding syntax we would use if programming the module as a class in C++ (e.g., `LCD.init()`). Using this convention, we can easily distinguish public and private objects.

*Use upper and lower case to specify the scope of an object.* We will define I/O ports and constants using no lower-case letters, as if typing with caps-lock on. In other words, names without lower-case letters refer to objects with fixed values. `TRUE FALSE` and `NULL` are good examples of fixed-valued objects. As mentioned earlier, constant names formed from multiple words will use an underline character to delimit the individual words—for example `MAX_VOLTAGE UPPER_BOUND FIFO_SIZE`. Global objects will begin with a capital letter, but may also include some lowercase letters. Local variables will begin with a lower-case letter, and may or may not include uppercase letters. Since all functions are global, we can start function names with either an uppercase or lowercase letter. Using this convention, we can distinguish constants, globals, and locals.

**Observation:** An object’s properties (public/private, local/global, constant/variable) are always perfectly clear at the place where the object is defined. The importance of the naming policy is to extend that clarity also to the places where the object is used.

*Use capitalization to delimit words.* Names that contain multiple words should be defined using a capital letter to signify the first letter of the word. Recall that the case of the first letter specifies whether it is local or global. Some programmers use the underline as a word-delimiter, but except for constants, we will reserve underline to separate the module name from the variable name. Table 2.7 presents examples of the naming convention used in this book.

**Table 2.7**

Examples of names.

Type	Examples
constants	CR_SAFE_TO_RUN PORTA STACK_SIZE START_OF_RAM
local variables	maxTemperature lastCharTyped errorCnt
private global variable	MaxTemperature LastCharTyped ErrorCnt
public global variable	DAC_MaxVoltage Key_LastCharTyped Network_ErrorCnt
private function	ClearTime wrapPointer InChar
public function	Timer_ClearTime RxFifo_Put Key_InChar

**Checkpoint 2.14:** How can you tell whether a function is private or public?

**Checkpoint 2.15:** How can you tell whether a variable is local or global?

## 2.4 Abstraction

**2.4.1 Definitions** *Software abstraction* is when we can define a complex problem with a set of basic abstract principles. If we can construct our software system using these building blocks, then we have a better understanding of the problem, because we can separate what we are doing from the details of how we are getting it done. This separation also makes it easier to optimize. It provides for a proof of correct function and simplifies both extensions and customization. A good example of abstraction is the *finite-state machine* (FSM) implementation. The abstract principles of FSM development are the inputs, outputs, states, and state transitions. If we can take a complex problem and map it into a FSM model, then we can solve it with simple FSM software tools. Our FSM software implementation will be easy to understand, debug, and modify. Other examples of software abstraction include *proportional integral derivative* (PID) digital controllers, fuzzy logic digital controllers, neural networks, and linear systems of differential equations. In each case, the problem is mapped into a well-defined model with a set of abstract yet powerful rules. Then, the software solution is a matter of implementing the rules of the model.

Linked lists are lists or nodes where one or more of the entries is a pointer (link) to other nodes of similar structure. We can have statically allocated fixed-size linked lists that are defined at assemble or compile time and exist throughout the life of the software. On the other hand, we implement dynamically allocated variable-size linked lists that are constructed at run time and can grow and shrink in size. We will use a data structure similar to a linked list, called a *linked structure*, to build a FSM controller. Linked structures are very flexible and provide a mechanism to implement abstraction.

The FSM controller is a good example of the concept of program abstraction. A well-defined model or framework is used to solve our problem (implemented with a linked structure). The three advantages of abstraction are (1) it can be faster to develop because a lot of the building blocks preexist, (2) it is easier to debug (prove correct) because it separates conceptual issues from implementation, and (3) it is easier to change. An important factor when implementing FSMs using linked structures is that there should be a clear and one-to-one mapping between the FSM and the linked structure; that is, there should be one structure for each state.

We will present two implementations of finite state machines. The **Moore FSM** has an output that depends on state, and the next state depends on input and current state. On the other hand, the **Mealy FSM** has an output that depends on both the input and the state, and the next state depends on input and current state. We will use a Moore implementation if

there is an association between a state and an output. There can be multiple states with the same output, but the output defines in part what it means to be in that state. For example, in a traffic light controller, the state of green light on the North road (red light on the East road) is caused by outputting a specific pattern to the traffic light. Conversely, we will use a Mealy implementation if the output causes the state to change. In this situation, we do not need a specific output to be in that state; rather, the outputs are required to cause the state transition. For example, to make a robot stand up, we perform a series of outputs causing the state to change from sitting to standing. Although we can rewrite any Mealy machine as a Moore machine and vice versa, it is better to implement the format that is more natural for the particular problem. In this way the state graph will be easier to understand.

**Checkpoint 2.16:** What are the differences between a Mealy and Moore finite-state machine?

One of the common features in many finite-state machines is a time delay. We will learn very elaborate mechanisms to handle time in Chapters 4 through 6, but in this section we will implement a simple time delay. The 9S12 has a 16-bit timer register, called TCNT, which increments at a regular rate. This counter is incremented at a fixed rate, and the software can read its value to know the current time.

## 2.4.2 9S12 Timer Details

Table 2.8 shows some of the 9S12 timer registers. The default E-clock rate is determined by the crystal, but the software can change the rate with the PLL. On the 9S12, TCNT is a 16-bit unsigned counter that is incremented at a rate determined by three bits (PR2, PR1, and PR0) in the TSCR2 register as shown in Table 2.9. Every time the TCNT register overflows

Address	msb																	lsb	Name
\$0044	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TCNT		
Address	Bit 7	6	5	4	3	2	1	Bit 0	Name										
\$0046	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0	TSCR1										
\$004D	TOI	0	0	0	TCRE	PR2	PR1	PR0	TSCR2										
\$004F	TOF	0	0	0	0	0	0	0	TFLG2										

**Table 2.8**  
9S12 timer ports.

PR2	PR1	PR0	Divide by	E = 4 MHz		E = 8 MHz		E = 24 MHz	
				TCNT period	TCNT frequency	TCNT period	TCNT frequency	TCNT period	TCNT frequency
0	0	0	1	250 ns	4 MHz	125 ns	8 MHz	41.7 ns	24 MHz
0	0	1	2	500 ns	2 MHz	250 ns	4 MHz	83.3 ns	12 MHz
0	1	0	4	1 µs	1 MHz	500 ns	2 MHz	167 ns	6 MHz
0	1	1	8	2 µs	500 kHz	1 µs	1 MHz	333 ns	3 MHz
1	0	0	16	4 µs	250 kHz	2 µs	500 kHz	667 ns	1.5 MHz
1	0	1	32	8 µs	125 kHz	4 µs	250 kHz	1.33 µs	667 kHz
1	1	0	64	16 µs	62.5 kHz	8 µs	125 kHz	2.67 µs	333 kHz
1	1	1	128	32 µs	31.25 kHz	16 µs	62.5 kHz	5.33 µs	167 kHz

**Table 2.9**  
Given an E clock frequency, the PR2, PR1, and PR0 bits define the TCNT rate.

from \$FFFF to 0, the TOF flag in the TFLG2 register is set. The TOF condition will cause an interrupt if the arm bit TOI equals 1. Chapter 4 discusses **TOF** interrupts.

### 2.4.3 Time Delay Software Using the Built-in Timer

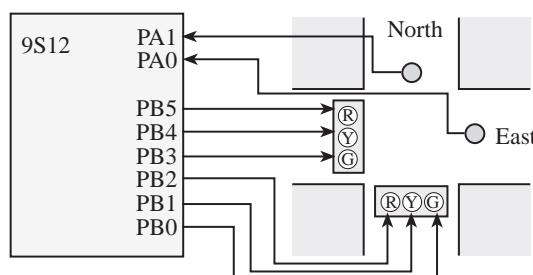
In order to use TCNT on the 9S12, you must first set bit 7 of the TSCR1 register. Program 5.10 shows how to use TCNT to create a time delay on a 9S12 running at 4 MHz. The delay parameter to the assembly Timer\_Wait subroutine can be any number from 1 to 32767.

**Checkpoint 2.17:** Explain how the timer prescale affects the range and resolution of the Timer\_Wait function.

**Example 2.1:** We will design a traffic light controller using a Moore FSM. The goal is to maximize traffic flow, minimize waiting time at a red light, and avoid accidents. The outputs of a Moore FSM are only a function of the current state. The intersection has two one-way roads: North and East, as shown in Figure 2.23. It will have two inputs (car sensors on North and East roads) and six outputs (one for each light in the traffic signal). The six traffic lights are interfaced to Port B. The two sensors are connected to Port A, such that

- 00 means no cars exist on either road
- 01 means there are cars on the East road
- 10 means there are cars on the North road
- 11 means there are cars on both roads

**Figure 2.23**  
Traffic light interface.



**Solution:** The first step in designing the FSM is to create some states. A Moore implementation was chosen because the output pattern (which lights are on) defines which state we are in. Each state is given a symbolic name:

- goN, 100001 makes it green on North and red on East
- waitN, 100010 makes it yellow on North and red on East
- goE, 001100 makes it red on North and green on East
- waitE, 010100 makes it red on North and yellow on East

The output pattern for each state is drawn inside the state circle. The time to wait for each state is also included. How the machine operates will be dictated by the input-dependent state transitions. We create decision rules defining what to do for each possible input and for each state. For this design, we can list heuristics describing how the traffic light is to operate:

If no cars are coming, we will stay in a green state.

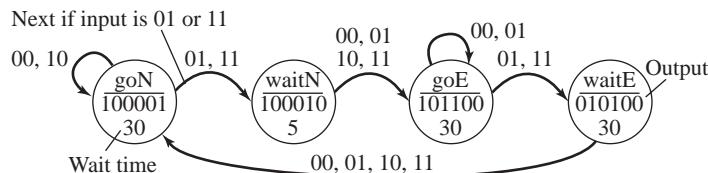
To change from green to red, we will implement a yellow light of exactly 5 seconds. Green lights will last at least 30 seconds.

If cars are only coming in one direction, we will move to and stay green in that direction.

If cars are coming in both directions, we will cycle through all four states.

Finally, we implement the heuristics by defining the state transitions, as illustrated in Figure 2.24. Instead of using a graph to define the finite state machine, we could have used a table, as shown in Table 2.10.

**Figure 2.24**  
Graphical form of a Moore FSM that implements a traffic light.



**Table 2.10**  
Tabular form of a Moore FSM that implements a traffic light.

State \ Input	00	01	10	11
goN, 100001, 30	goN	waitN	goN	waitN
waitN, 100010, 5	goE	goE	goE	goE
goE, 001100, 30	goE	goE	waite	waite
waite, 010100, 5	goN	goN	goN	goN

The first step in designing the software is to decide on the sequence of operations.

1. Initialize timer and directions registers
2. Specify initial state
3. Perform FSM controller
  - a) Output to traffic lights, which depends on the state
  - b) Delay, which depends on the state
  - c) Input from sensors
  - d) Change states, which depends on the state and the input

The second step is to define the FSM graph using a data structure. Program 2.10 shows a table implementation of the Moore FSM. This implementation uses a table data structure, where each state is an entry in the table and state transitions are defined as indices into this table. Program 2.11 uses a linked structure, where each state is a node and state transitions are defined as pointers to other nodes. The four `Next` parameters define the input-dependent state transitions. The wait times are defined in the software as fixed-point decimal numbers with units of 0.01 s, giving a range of 10 ms to 655.35 s. Using good labels makes the program easier to understand; in other words `goN` is more descriptive than `&fsm[ 0 ]`.

**Observation:** The table implementation requires less memory space for the FSM data structure, but the pointer implementation will run faster.

On microcontrollers that have both ROM and EEPROM we can place the FSM data structure in EEPROM and the assembly language program in ROM. This allows us to make minor modifications to the finite-state machine (add/delete states, change input/output values) by changing the linked structure without modifying the assembly language controller. In this way small modifications/upgrades/options to the finite-state machine can be made by reprogramming the EEPROM without throwing the chip away.

```

        org $4000 ; Put in ROM
OUT    equ 0      ;offset for output
WAIT   equ 1      ;offset for time
NEXT   equ 3      ;offset for next state
gN    equ 0       ; go north, stop east
wN    equ 1       ;wait north, stop east
gE    equ 2       ;stop north, go   east
wE    equ 3       ;stop north, wait east
FSM    fcb $21    ;North green, East red
        fdb 3000  ;30sec
        fcb gN,wN,gN,wN
FSM1   fcb $22    ;North yellow, East red
        fdb 500   ;5sec
        fcb gE,gE,gE,gE
FSM2   fcb $0C    ;North red, East green
        fdb 3000  ;30 sec
        fcb gE,gE,wE,wE
FSM3   fcb $14    ;North red, East yellow
        fdb 500   ;5sec
        fcb gN,gN,gN,gN
main   lds #$4000 ;stack init
        bsr Timer_Init ;enable TCNT
        movb #$FF,DDRB ;PB5-0 are lights
        movb #$00,DDRA ;PA1-0 are sensors
        ldaa #gN      ;State index
FSM    ldx #FSM
        ldab #7      ;size of state
        mul          ;D=7*n
        leax D,X     ;X points to state
        ldb OUT,x
        stab PORTB   ;Output
        ldy WAIT,x   ;Time delay
        bsr Timer_Wait10ms
        ldab PORTA   ;Read input
        andb #$03    ;just bits 1,0
        abx          ;add 0,1,2,3
        ldaa NEXT,x  ;Next state
        bra FSM
        org $FFFFE
        fdb main     ;reset vector

```

```

const struct State {
    unsigned char Out;
    unsigned short Time;
    unsigned char Next[4];
} typedef const struct State STyp;
#define gN 0
#define wN 1
#define gE 2
#define wE 3
STyp FSM[4]={
{0x21,3000,{gN,wN,gN,wN}},
{0x22, 500,{gE,gE,gE,gE}},
{0x0C,3000,{gE,gE,wE,wE}},
{0x14, 500,{gN,gN,gN,gN}}};
void main(void){
    unsigned char n; // state number
    unsigned char Input;
    Timer_Init();
    DDRB = 0xFF;
    DDRA &= ~0x03;
    n = 9N;
    while(1){
        PORTB = FSM[n].Out;
        Timer_Wait10ms(FSM[n].Time);
        Input = PORTA&0x03;
        n = FSM[n].Next[Input];
    }
}

```

### Program 2.10

Table implementation of a Moore FSM.

The FSM approach makes it easy to change. To change the wait time for a state, we simply change the value in the data structure. To add more states (e.g., put a red/red state after each yellow state, which will reduce accidents caused by bad drivers running the yellow light), we simply increase the size of the `fsm[ ]` structure and define the `Out`, `Time`, and `Next` fields for these new states.

To add more output signals (e.g., walk and left-turn lights), we simply increase the precision of the `Out` field. To add two more input lines (e.g., wait button, left-turn car sensor), we increase the size of the `next` field to `Next[16]`. Because now there are four input lines, there are 16 possible combinations, where each input possibility requires a `Next` value specifying where to go if this combination occurs. In this simple scheme, the size of the `Next[ ]` field will be 2 raised to the power of the number of input signals.

```

        org $4000 ; Put in ROM
OUT    equ 0      ;offset for output
WAIT   equ 1      ;offset for time
NEXT   equ 3      ;offset for next state
goN    fcb $21  ;North green, East red
       fdb 3000 ;30sec
       fdb goN,waitN,goN,waitN
waitN  fcb $22  ;North yellow, East red
       fdb 500  ;5sec
       fdb goE,goE,goE,goE
goE    fcb $0C  ;North red, East green
       fdb 3000 ;30 sec
       fdb goE,goE,waitE,waitE
waitE  fcb $14  ;North red, East yellow
       fdb 500  ;5sec
       fdb goN,goN,goN,goN
main   lds #$4000 ;stack init
       bsr Timer_Init ;enable TCNT
       movb #$FF,DDRB ;PB5-0 are lights
       movb #$00,DDRA ;PA1-0 are sensors
       ldx #goN      ;State pointer
FSM    ldab OUT,x
       stab PORTB    ;Output
       ldy WAIT,x    ;Time delay
       bsr Timer_Wait10ms
       ldab PORTA    ;Read input
       andb #$03    ;just bits 1,0
       lslb          ;2 bytes/address
       abx           ;add 0,2,4,6
       ldx NEXT,x   ;Next state
       bra FSM
       org $FFFF
       fdb main     ;reset vector

```

```

const struct State {
    unsigned char Out;
    unsigned short Time;
} const struct State *Next[4];
typedef const struct State STyp;
#define goN &FSM[0]
#define waitN &FSM[1]
#define goE &FSM[2]
#define waitE &FSM[3]
STyp FSM[4]={
{0x21,3000,{goN,waitN,goN,waitN}},
{0x22, 500,{goE,goE,goE,goE}},
{0x0C,3000,{goE,goE,waitE,waitE}},
{0x14, 500,{goN,goN,goN,goN}}};
void main(void){
    STyp *Pt; // state pointer
    unsigned char Input;
    Timer_Init();
    DDRB = 0xFF;
    DDRA &= ~0x03;
    Pt = goN;
    while(1){
        PORTB = Pt->Out;
        Timer_Wait10ms(Pt->Time);
        Input = PORTA&0x03;
        Pt = Pt->Next[Input];
    }
}

```

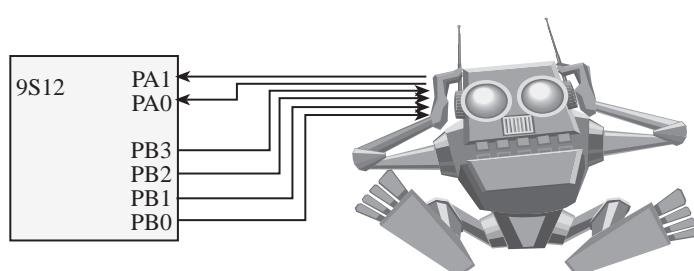
**Program 2.11**

Pointer implementation of a Moore FSM.

**Example 2.2:** The goal is to design a finite-state machine robot controller, as illustrated in Figure 2.25. Because the outputs cause the robot to change states, we will use a Mealy implementation. The outputs of a Mealy FSM depend on both the input and the current

**Figure 2.25**

Robot interface.



state. This robot has mood sensors, that are interfaced to Port A. The robot has four possible mutually exclusive conditions

- 00 OK, the robot is feeling fine
- 01 Tired, the robot energy levels are low
- 10 Curious, the robot senses activity around it
- 11 Anxious, the robot senses danger

**Solution:** There are four actions this robot can perform, which are triggered by pulsing (make high, then make low) one of the four signals interfaced to Port B.

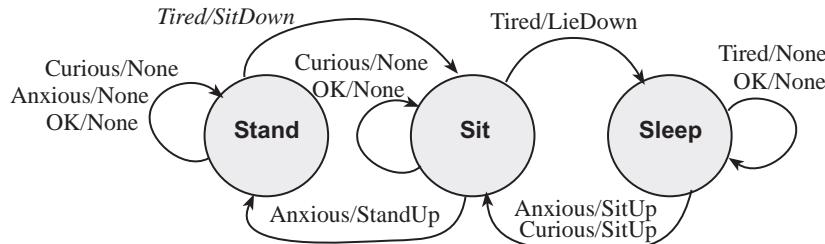
- PB3 SitDown, assuming the robot is standing, it will perform a sequence of moves to sit down
- PB2 StandUp, assuming the robot is sitting, it will perform a sequence of moves to stand up
- PB1 LieDown, assuming the robot is sitting, it will perform a sequence of moves to lie down
- PB0 SitUp, assuming the robot is sleeping, it will perform a sequence of moves to sit up

For this design, we can list heuristics describing how the robot is to operate:

- If the robot is OK, it will stay in the state it is currently in.
- If the robot's energy levels are low, it will go to sleep.
- If the robot senses activity around it, it will awaken from sleep.
- If the robot senses danger, it will stand up.

These rules are converted into a finite-state machine graph, as shown in Figure 2.26. Each arrow specifies both an input and an output. For example, the “Tired/SitDown” arrow from Stand to Sit states means if we are in the Stand state and the input is Tired, then we will output the SitDown command and go to the Sit state. Mealy machines can have time delays, but this example just didn’t have time delays.

**Figure 2.26**  
Mealy FSM for a robot controller.



The first step in designing the software is to decide on the sequence of operations.

1. Initialize directions registers
2. Specify initial state
3. Perform FSM controller
  - a) Input from sensors
  - b) Output to the robot, which depends on the state and the input
  - c) Change states, which depends on the state and the input

The second step is to define the FSM graph using a linked data structure. Programs 2.12 and 2.14 show two possible implementations of the Mealy FSM. The implementation in

```

        org $4000 ;EEPROM
Out      equ 0      ;Index for output
Next     equ 4      ;Index for next
None     equ 0      ;no pulse
SitDown  equ $08   ;pulse on PB3
StandUp  equ $04   ;pulse on PB2
LieDown  equ $02   ;pulse on PB1
SitUp    equ $01   ;pulse on PB0
Stand    fcb None,SitDown,None,None
          fdb Stand,Sit,Stand,Stand
Sit     fcb None,LieDown,None,StandUp
          fdb Sit,Sleep,Sit,Stand
Sleep   fcb None,None,SitUp,SitUp
          fdb Sleep,Sleep,Sit,Sit
main   movb #$FF,DDRB ;PB3-0 output
        movb #$00,DDRA ;PA1-0 inputs
        ldx #Stand    ;current state
loop   ldab PORTA  ;Read Sensors
        andb #$03    ;Just bits1-0
        abx         ;Base+input
        ldaa Out,x  ;Fetch output
        staa PORTB  ;Pulse function
        clr  PORTB
        abx         ;Base+2*input
        ldx Next,x  ;Infinite loop
        bra  loop
org    $FFFF
fdb   main      ;reset vector

```

```

const struct State{
    unsigned char Out[4];           // outputs
    const struct State *Next[4];   // Next
};

typedef const struct State StateType;
#define Stand &fsm[0]
#define Sit   &fsm[1]
#define Sleep &fsm[2]
#define None  0x00
#define SitDown 0x08 // pulse on PB3
#define StandUp 0x04 // pulse on PB2
#define LieDown 0x02 // pulse on PB1
#define SitUp   0x01 // pulse on PB0
StateType FSM[3]={
{{None,SitDown,None,None}, //Standing
 {Stand,Sit,Stand,Stand}},
{{None,LieDown,None,StandUp}, //Sitting
 {Sit,Sleep,Sit,Stand }},
{{None,None,SitUp,SitUp}, //Sleeping
 {Sleep,Sleep,Sit,Sit}}
};
void main(void){
StateType *Pt; // Current State
unsigned char Input;
        DDRB = 0xFF; // Output to robot
        DDRA &= ~0x03; // Input from sensor
        Pt = Stand; // Initial State
        while(1){
            Input = PORTA&0x03; // Input=0-3
            PORTB = Pt->Out[Input]; // Pulse
            PORTB = 0;
            Pt = Pt->Next[Input]; // state
        }
}

```

### Program 2.12

Outputs defined as numbers for a Mealy Finite State Machine.

Program 2.12 defines the outputs as simple numbers, where each pulse is defined as the bit mask required to cause that action. Program 2.13 uses functions to affect the output. The four `Next` parameters define the input-dependent state transitions.

Again proper memory allocation is required if we wish to implement a standalone or embedded system. The `org` pseudoops are used to place the FSM data structure in EEPROM.

**Observation:** In order to make the FSM respond quicker, we could implement a time delay function that returns immediately if an alarm condition occurs. If no alarm exists, it waits the specified delay.

**Checkpoint 2.18:** What happens if the robot is sleeping and then becomes anxious?

<pre>         org \$4000 ;EEPROM Out      equ 0 ;outputs Next     equ 8 ;Index for next  None     rts      ;no pulse SitDown  bset PORTB,#\$08 ;pulse PB3           clr  PORTB           rts StandUp  bset PORTB,#\$04 ;pulse PB2           clr  PORTB           rts LieDown  bset PORTB,#\$02 ;pulse PB1           clr  PORTB           rts SitUp   bset PORTB,#\$01 ;pulse PB0           clr  PORTB           rts Stand   fdb None,SitDown,None,None           fdb Stand,Sit,Stand,Stand Sit     fdb None,LieDown,None,StandUp           fdb Sit,Sleep,Sit,Stand Sleep   fdb None,None,SitUp,SitUp           fdb Sleep,Sleep,Sit,Sit  main   lds  #\$4000         movb #\$FF,DDRB ;PB3-0 output         movb #\$00,DDRA ;PA1-0 inputs         ldx  #Stand    ;current state loop   ldab PORTA    ;Read Sensors         andb #\$03    ;Just bits1-0         lslb          ;0,2,4,6         abx           ;Base+2*input         jsr  [Out,x] ;output         ldx  Next,x         bra  loop     ;Infinite loop         org  \$FFFFE         fdb  main     ;reset vector </pre>	<pre> const struct State{     void (*CmdPt)[4](void);      // outputs     const struct State *Next[4]; // Next };  typedef const struct State StateType; #define Stand &amp;fsm[0] #define Sit   &amp;fsm[1] #define Sleep &amp;fsm[2] void None(void){} void SitDown(void){     PORTB=0x08; PORTB=0;} // pulse on PB3 void StandUp(void){     PORTB=0x04; PORTB=0;} // pulse on PB2 void LieDown(void){     PORTB=0x02; PORTB=0;} // pulse on PB1 void SitUp(void) {     PORTB=0x01; PORTB=0;} // pulse on PB0 StateType FSM[3]={ {{&amp;None,&amp;SitDown,&amp;None,&amp;None}, //Standing {Stand,Sit,Stand,Stand}}, {{&amp;None,&amp;LieDown,&amp;None,&amp;StandUp}, //Sitting {Sit,Sleep,Sit,Stand }}, {{&amp;None,&amp;None,&amp;SitUp,&amp;SitUp}, //Sleeping {Sleep,Sleep,Sit,Sit}}}; };  void main(void){ StateType *Pt; // Current State unsigned char Input;     DDRB = 0xFF; // Output to robot     DDRA &amp;= ~0x03; // Input from sensor     Pt = Standing; // Initial State     while(1){         Input = PORTA&amp;0x03; // Input=0-3         (*Pt-&gt;CmdPt[Input])(); // function         Pt = Pt-&gt;Next[Input]; // next     } } </pre>
---	--

### Program 2.13

Outputs defined as functions for a Mealy Finite State Machine.

## 2.5 Modular Software Development

In this section we introduce the concept of modular programming and demonstrate that it is an effective way to organize our software projects. There are three reasons for forming modules. Functional abstraction allows us to reuse a software module from multiple locations. Complexity abstraction allows us to divide a highly complex system into smaller, less complicated components. The third reason is portability. If we create modules for the I/O devices, then we can isolate the rest of the system from the hardware details. This approach will be presented later in Section 2.6 on layered software.

### 2.5.1 Variables

Variables are an important component of software design, and there are many factors to consider when creating variables. Some of the obvious considerations are the size and

format of the data. Another factor is the *scope* of a variable. The scope of a variable defines which software modules can access the data. Variables with an access that is restricted to one software module are classified as *private*, and variables shared between multiple modules are *public*. In general, a system is easier to design (because the modules are smaller and simpler), easier to change (because code can be reused), and easier to verify (because interactions between modules are well-defined) when we limit the scope of our variables. However, since modules are not completely independent, we need a mechanism to transfer information from one to another. In this chapter, we will develop parameter passing methodologies. Because their contents are allowed to change, all variables must be allocated in registers or in RAM, but not in ROM. On the one hand, *global* variables contain information that is permanent and are usually assigned a fixed location in RAM. On the other hand, *local* variables contain temporary information and are stored in a register or allocated on the stack. One of the important objectives of this chapter is to present design steps for creating, using, and destroying local variables on the stack. In summary, there are three types of variables: public globals (shared permanent), private globals (unshared permanent), and private locals (unshared temporary). Because there is no appropriate way to create a public local variable, we usually refer to private local variables simply as local variables, and the fact that they are private is understood.

A *local variable* contains temporary information. Since we will implement local variables on the stack or in registers, this information can not be shared with other software modules. Therefore, under most situations, we can further classify these variables as private. Local variables are allocated, used, and then deallocated in this specific order. For speed reasons, we wish to assign local variables to a register. When we assign local variable to a register, we can do so in a formal manner. There will be a certain line in the assembly software at which the register begins to contain the variable (allocation), followed by lines where the register contains the information (access or usage), and a certain line in the software after which the register no longer contains the information (deallocation). In C, we define local variables after a brace:

```
void MyFunction(void){ short i; // i is a local variable
    for(i=0; i<10; i++) SCI_OutUDec(i);
}
```

The information stored in a local variable is not permanent. This means if we store a value into a local variable during one execution of the module, the next time that module is executed the previous value is not available. Examples include loop counters and temporary sums. We use a local variable to store data that is temporary in nature. We can implement a local variable using the stack or registers. Some reasons why we choose local variables over global variables include:

- Dynamic allocation/release allows for reuse of RAM memory.
- Limited scope of access (making it private) provides for data protection; only the program that created the local variable can access it.
- Since an interrupt will save registers and create its own stack frame, the code is reentrant.
- Since absolute addressing is not used, the code is relocatable.

Some reasons why we place local variables on the stack rather than using registers include:

- We can use symbolic names for the local variables, making it easier to understand.
- The number of variables is only limited by the size of the stack, which is more than registers.
- Because it is more general, it will be easier to add additional variables in the future.

A *global variable* is allocated at a permanent and fixed location in RAM. A public global variable contains information that is shared by more than one program module. We must use global variables to pass data between the main program (i.e., foreground thread)

and an ISR (i.e., background thread). If a function called from the foreground belongs to the same module as the ISR, then a global variable used to pass data between the function and the ISR is classified as a private global (assuming software outside the module does not directly access the data). Global variables are allocated at assembly time and never deallocated. Allocation of a global variable means the assembler assigns the variable a fixed location in RAM. The information they store is permanent. Examples include time of day, date, calibration tables, user name, temperature, fifo queues, and message boards. We use absolute addressing (direct or extended) to access their information. When dealing with complex data structures, pointers to the data structures are shared. In general, it is a poor design practice to employ public global variables. On the other hand, private global variables are necessary to store information that is permanent in nature. In C, we define global variables outside of the function:

```
short Count=0; // Count is a global variable
void MyFunction(void){
    Count++;
}
```

**Checkpoint 2.19:** How do you create a local variable in C?

In C, a `static` local has permanent allocation, which means it maintains its value from one call to the next. It is still local in scope, meaning it is only accessible from within the function. Therefore, modifying a local variable with `static` changes its allocation (it is now permanent) but doesn't change its scope (it is still private). In the following example, `count` contains the number of times `MyFunction` is called. The initialization of a static local occurs just once, during startup.

```
void MyFunction(void){ static short count=0;
    count++;
}
```

In C, we create a private global variable using the `static` modifier. Modifying a global variable with `static` does not change its allocation (it is still permanent), but it does reduce its scope. Regular globals can be accessed from any function in the system (public), whereas a `static` global only can be accessed by functions within the same file. Static globals are private. Functions can be static also, meaning they can be called only from other functions in the file.

```
static short myPrivateGlobalVariable; // file only
void static MyPrivateFunction(void){
}
```

In C, a `const` global is read-only. It is allocated in the ROM portion of memory. Constants, of course, must be initialized at compile time.

```
const short Slope=21;
const char SinTable[8]={0,50,98,142,180,212,236,250};
```

**Common error:** If you leave off the `const` modifier in the `SinTable` example, the table will be allocated twice: once in ROM containing the initial values and once in RAM containing data to be used at run time. Upon startup, the system copies the ROM-version into the RAM-version.

**Maintenance tip:** It is good practice to specify whether an assembly variable is signed or unsigned in the comments. If the information has units (e.g., volts, seconds etc.), this should be included also.

**Common error:** In C, global variables are initialized to zero by default, but local variables are not initialized.

**Observation:** Sometimes we store temporary information in global variables out of laziness. This practice is to be discouraged because it wastes memory and may cause the module not to be reentrant.

**Checkpoint 2.20:** How do you create a global variable in C?

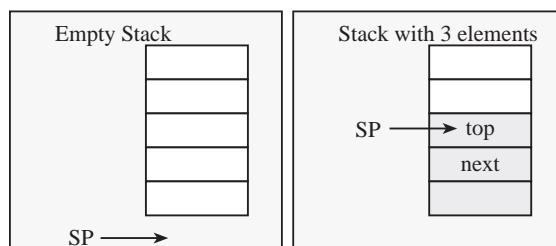
**Checkpoint 2.21:** How does the `static` modifier affect locals, globals, and functions in C?

**Checkpoint 2.22:** How does the `const` modifier affect a global variable in C?

One of the more flexible means to create local variables will be the stack. In this section, we define a set of rules for proper use of the stack. A last-in-first-out (LIFO) stack is implemented in hardware by most computers. The stack can be used for local variables (temporary storage), saving return addresses during subroutine calls, passing parameters to subroutines, and saving registers during the processing of an interrupt. The first advantage of placing local variables on the stack is that the storage can be dynamically allocated before usage and deallocated after usage. The second advantage is the facilitation of reentrant software.

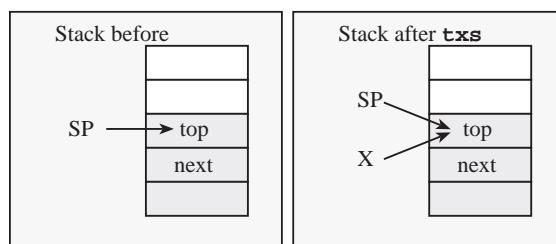
The stack pointer (SP) on the 9S12 points to the *top* entry of the stack, as shown in Figure 2.27. If it exists, we define the data immediately below the top (larger memory address) as *next to top*. To *push* a byte on the stack, we first decrement the stack pointer (SP), and then we store the byte at the location pointed to by the SP. To *pull* a byte from the stack, first we read the byte from memory pointed to by SP, and then we increment the SP.

**Figure 2.27**  
9S12 stack.



The instruction `tsx` will move a copy of the stack pointer into Register X as shown in Figure 2.28. The `tsx` and `tsy` instructions do not modify the stack pointer.

**Figure 2.28**  
The `tsx` instruction creates a stack frame pointer.



Then, index mode reads and writes are possible. For example, to read the next to the top byte,

```
tsx           ;RegX points to the top byte of the stack
ldaa 1,x     ;RegA = the next to the top byte
```

The LIFO stack has a few rules:

1. Program segments should have an equal number of pushes and pulls;
2. Stack accesses (PUSH or PULL) should not be performed outside the allocated area;
3. Stack reads and writes should not be performed within the *free area*,
4. Stack PUSH should first decrement SP, then store the data,
5. Stack PULL should first read the data, then increment SP.

Programs that violate rule number 1 will probably crash when an `rts` instruction pulls an illegal address of the stack at the end of a subroutine. The **TExaS** simulator will usually recognize this error as an illegal memory access when the processor tries to fetch an opcode at this incorrect address. The `backdump` command will be useful to retrace the steps leading up to the crash.

Violations of rule number 2 can be caused by a stack underflow or overflow. Stack underflow is caused when there are more pulls than pushes, and is always the result of a software bug. The **TExaS** simulator will recognize this error as an illegal memory access when the processor tries to pull data from an address that doesn't exist. A stack overflow can be caused by two reasons. If the software mistakenly pushes more than it pulls, then the stack pointer will eventually overflow its bounds. Even when there is exactly one pull for each push, a stack overflow can occur if the stack is not allocated large enough. Stack overflow is a very difficult bug to recognize, because the first consequence occurs when the computer pushes data onto the stack and overwrites data stored in a global variable. At this point the local variables and global variables exist at overlapping addresses. Setting a breakpoint at the first address of the allocated stack area allows you to detect a stack overflow situation.

**Checkpoint 2.23:** How do you specify the size of the stack?

The following assembly code violates rule 3 and will not work if interrupts are active. The objective is to save register A onto the stack. When an interrupt occurs, registers will automatically be pushed on the stack, destroying the data.

```
staa -1,sp      ;Store zero onto the stack (**illegal***)
```

To use the stack, one first allocates, then saves. The following assembly code also violates rule 3, because it first stores it on the stack, then allocates space. The objective is to push a zero onto the stack.

```
tsx          ;RegX points to the top of the stack
dex
clr 0,x    ;Store zero onto the stack (**illegal***) 
des          ;Make space for the zero
```

If an interrupt were to occur between the `clr` and `des` instructions, the zero will be destroyed when all the registers are pushed on the stack by the interrupt process. The proper technique is to allocate first,

```
des      ;Allocate stack space first
tsx      ;RegX points to the top of the stack
clr ,x   ;Store zero onto the stack
```

Stack implementation of local variables has four stages: binding, allocation, access, and deallocation.

**1. Binding** is the assignment of the address (not value) to a symbolic name. This address will be the actual memory location to store the local variable. The assembler binds the symbolic name to a stack index. The computer calculates the actual location during execution. For example:

```
sum set 0    ;16-bit local variable, stored on the stack
```

**Checkpoint 2.24:** Why is `set` better than `equ` for binding?

**2. Allocation** is the generation of memory storage for the local variable. The computer allocates space during execution by decrementing the SP. In this first example, the software allocates the local variable by pushing a register on the stack. An 8-bit push (e.g., `psha`) creates an uninitialized 8-byte local variable, and a 16-bit push (e.g., `pshx`) creates an uninitialized 16-byte local variable. The value in the register is irrelevant, the instruction is used because it decrements the SP.

```
pshx    ;allocate 16-bit sum
```

In this next example, the software allocates the local variable by decrementing the stack pointer. This local variable is also uninitialized.

```
des    ;allocate 16-bit sum
des
```

If you wished to allocate the 16-bit local and initialize it to zero, you could execute.

<code>ldx #0</code>	<code>movw #0,2,-sp ;allocate sum=0</code>
<code>pshx ;allocate sum=0</code>	

In this last example, the technique provides a mechanism for allocating large amounts of uninitialized stack space. This example allocates 20 bytes for the structure `big[20]`. Local variables are so important that the 9S12 has special instructions to simplify the implementation of local variables.

<code>tsx    ;allocate big[20]</code>	<code>leas -20,sp ;allocate big[20]</code>
<code>xgdx</code>	
<code>subd #20</code>	
<code>xgdx</code>	
<code>txs</code>	

**3. The access** to a local variable is a read or write operation that occurs during execution. In the next code fragment, the local variable `sum` is set to 0.

<code>tsx    ;X points to locals</code>	<code>movw #0,sum,sp ;sum=0</code>
<code>ldd #0</code>	
<code>std sum,x ;sum=0</code>	

In the next code fragment, the local variable `sum` is incremented.

<code>tsx</code>	<code>ldd sum,sp</code>
<code>ldd sum,x</code>	<code>addd #1</code>
<code>addd #1</code>	<code>std sum,sp ;sum=sum+1</code>
<code>std sum,x ;sum=sum+1</code>	

**4. Deallocation** is the release of memory storage for the location variable. The computer deallocates space during execution by incrementing SP. In this first example, the software deallocates the local variable by pulling a register from the stack.

```
pulx    ;deallocate 16-bit sum
```

**Observation:** When the software uses the “push-register” technique to allocate and the “pull-register” technique to deallocate, it looks as though it were saving and restoring the register. Because most applications of local variables involve storing into the local, the value pulled will NOT match the value pushed.

In this next example, the software deallocates the local variable by incrementing the stack pointer.

```
ins
ins      ;deallocate 16-bit sum
```

In this last example, the technique provides a mechanism for deallocating large amounts of stack space.

<pre>tsx      ;deallocate big[20] ldab #20 abx txs</pre>	<pre>leas 20,sp ;deallocate big[20]</pre>
--	---

**Checkpoint 2.25:** Write a 9S12 subroutine that allocates then deallocates three 8-bit locals.

The 9S12 provides a negative offset index addressing mode. It is possible to establish a stack frame pointer using either register X or Y. It is important in this implementation that once the stack frame pointer is established (e.g., the `tsx` instruction), that the stack frame register (X) not be modified. The term **frame** refers to the fact that the value of the stack frame pointer is fixed. Because the stack frame pointer should not be modified, every subroutine will save the old stack frame pointer of the function that called the subroutine (e.g., `pshx` at the top) and restore it before returning (e.g., `pulx` at the bottom.) The stack frame will allow you to use the `txs` to deallocate the local variables. Local variable access uses indexed addressing mode. The difference between the two versions is the position of the stack frame pointer. This example will be extended to include subroutine parameters later in the chapter. Notice the subroutine deallocates the locals by moving the stack frame pointer back into SP with the `txs` instruction.

**Observation:** One advantage of using a stack frame is that the `tsx` instruction needs to be executed only once at the beginning of the function.

**Observation:** Another advantage of using a stack frame is that you can push and pull within the body of the function, and still be able to access local variables using their symbolic name.

**Observation:** One disadvantage of using a stack frame is that a register is dedicated as the frame pointer, and thus it is unavailable for general use.

This example calculates the sum of the first 100 numbers. Program 2.14 shows the assembly code that implements this C function. The result will be returned by value in Register D.

```
unsigned short calc(void){ unsigned short sum,n;
sum = 0;
for(n=100;n>0;n--) {
    sum=sum+n;
}
return sum;
}
```

<pre> ; *****binding phase***** sum  set  0  16-bit number n   set  2  16-bit number ; *****allocation phase ***** calc pshx    ;save old Reg X             pshx    ;allocate n             pshx    ;allocate sum             tsx     ;stack frame pointer ; *****access phase *****             ldd  #0             std  sum,x  ;sum=0             ldd  #100             std  n,x    ;n=100 loop ldd  n,x    ;RegD=n             addd sum,x  ;RegD=sum+n             std  sum,x  ;sum=sum+n             ldd  n,x    ;n=n-1             subd #1             std  n,x             bne  loop ; *****deallocation phase ***             leas 4,sp      ;deallocation             pulx          ;restore old X             rts           ;RegD=sum </pre>	<pre> ; *****binding phase***** sum  set -4 16-bit number n   set -2 16-bit number ; *****allocation phase ***** calc pshx    ;save old Reg X             tsx     ;stack frame pointer             leas -4,sp ;allocate n,sum ; *****access phase *****             movw #0,sum,x ;sum=0             movb #100,n,x ;n=100 loop ldd  n,x    ;RegD=I             adddd sum,x  ;RegD=sum+n             std  sum,x  ;sum=sum+n             ldd  n,x    ;n=n-1             subd #1             std  n,x             bne  loop ; *****deallocation phase ***             txs      ;deallocation             pulx    ;restore old X             rts     ;RegD=sum </pre>
---	---

### Program 2.14

Assembly language implementation of a function with two local 16-bit variables.

**2.5.2 Modules** The key to completing any complex task is to break it down into manageable subtasks. Modular programming is a style of software development that divides the software problem into distinct and independent modules. The parts are as small as possible, yet relatively independent. Complex systems designed in a modular fashion are easier to debug because each module can be tested separately. Industry experts estimate that 50 to 90% of software development cost is spent in maintenance. All five aspects of software maintenance:

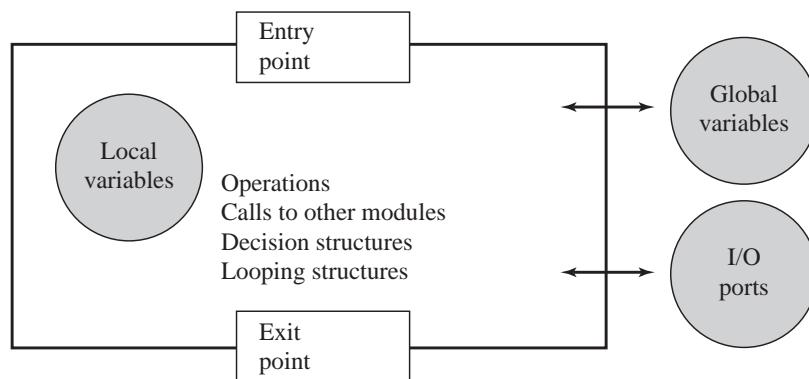
- Correcting mistakes
- Adding new features
- Optimizing for execution speed or program size
- Porting to new computers or operating systems
- Reconfiguring the software to solve a similar related problem

are simplified by organizing the software system into modules. The approach is particularly useful when a task is large enough to require several programmers (Figure 2.29).

A *program module* is a self-contained software task with clear entry and exit points. We make the distinction between a module and the assembly language subroutine or C language function. A module can be a collection of subroutines or functions that in its entirety performs a well-defined set of tasks. The collection of serial port I/O functions presented in Section 2.7.2 can be considered one module. A collection of 32-bit math operations is another example of a module. Modular programming involves both the specification of the individual modules and the connection scheme whereby the modules are connected together to form the software system. While the module may be called from many locations throughout the software, there should be a common interface.

**Figure 2.29**

Block diagram of a software module.



In assembly language, the entry point is used to call the subroutine. In this example, the input parameter (e.g., global variable `ss`) is first pushed on the stack by the instruction `movb ss,1,-sp`.

Next, the instruction `jsr sqrt` calls the subroutine. After the subroutine returns, the output parameter is in Register B. The input parameter is no longer needed, so `ins` is executed to discard it. Finally, the output parameter is stored in the global variable `tt` by the instruction `stab tt`.

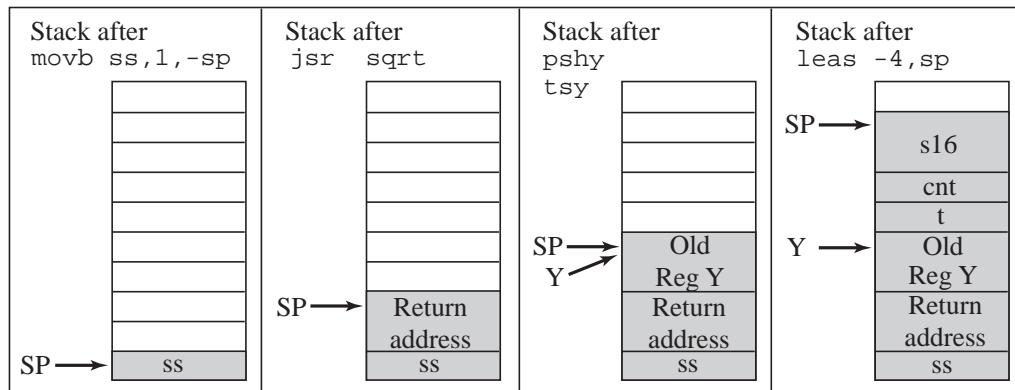
```

movb ss,1,-sp ;push parameter on the stack (binary fixed point)
jsr sqrt      ;subroutine call to the module "sqrt"
ins           ;discard input parameter
stab tt       ;save result
  
```

The stack contents are shown in Figure 2.30 as the calling routine pushes the input parameter `ss` on the stack, calls the subroutine (Figure 2.31) and then the subroutine allocates three local variables.

**Common error:** In many situations the input parameters have a restricted range. It would be inefficient for both the module and the calling routine to check for valid input. On the other hand, an error may occur if neither one checks for valid input.

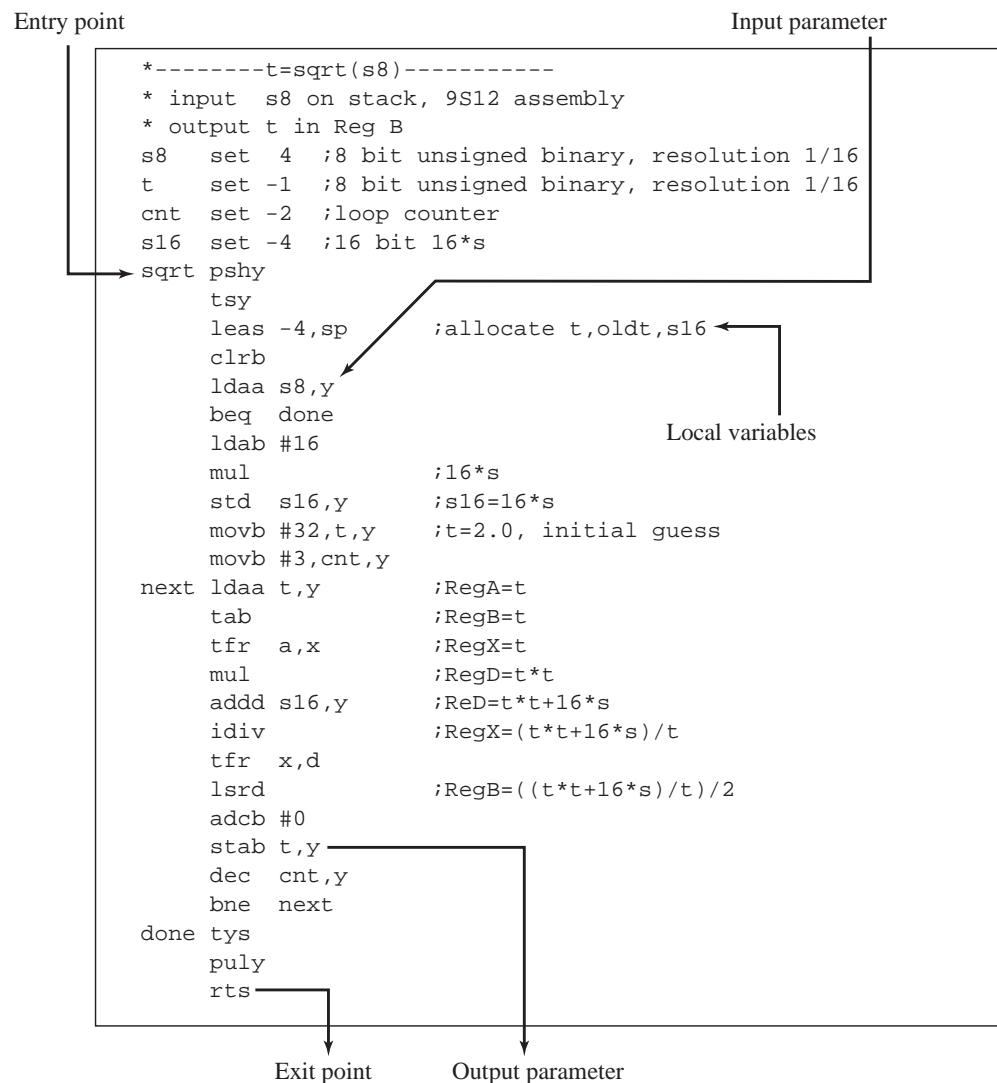
The entry point for a C module (Figure 2.32) is also the name of the function, and it is also used to call the function [e.g., `tt=sqrt(ss);`].

**Figure 2.30**

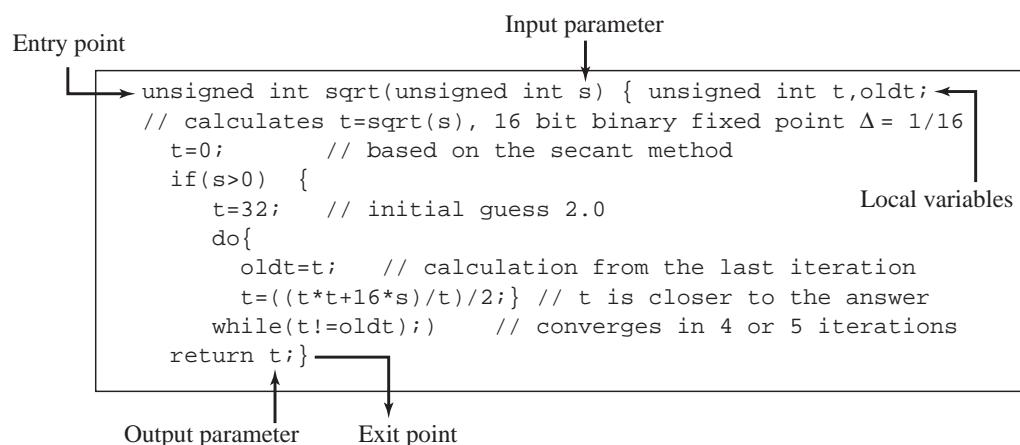
9S12 stack contents before and after the function call.

**Figure 2.31**

Example of a 9S12 assembly language module.

**Figure 2.32**

Example of a C language module.



The main module is the entry point of the entire program. The Freescale microcomputers employ a reset vector to specify where to begin execution after a reset. For Freescale microcomputers, the reset vector location is usually the last 2 bytes of the program ROM (EEPROM) space. The following assembly language software defines the reset vector:

```
org $FFFE
fdb main
```

An *exit point* is the ending point of a program module. The exit point of a subroutine is used to return to the calling routine (e.g., `rts`). We need to be careful about exit points. It is important that the stack be properly balanced at all exit points. Similarly if the subroutine returns parameters, then all exit points should return parameters in an acceptable format.

**Common error:** It is an error if all the exit points of an assembly subroutine do not balance the stack and return parameters in the same way.

If the main program has an exit point, it either stops the program or returns to the debugger. The *input parameters* (or arguments) are pieces of data passed from the calling routine into the module during execution. The *output parameter* (or argument) is information returned from the module back to the calling routine after the module has completed its task.

It is easy to return multiple parameters in assembly language. If just a few parameters need to be returned, we can use the registers. In this simple example, the numbers 1, 2, 3, 4 are to be returned:

```
module: ldaa #1
        ldab #2
        ldx #3
        ldy #4
        rts      ; returns four parameters in 4 registers
*****calling sequence*****
jsr module
* Reg A,B,X,Y have four results
```

If many parameters are needed, then the stack can be used. Space for the output parameters is allocated by the calling routine, and the module stores the results into those stack locations.

```
data1 equ 2
data2 equ 3
data3 equ 4
data4 equ 5
module movb #1,data1,sp ;first parameter onto stack
        movb #2,data2,sp ;second parameter onto stack
        movb #3,data3,sp ;third parameter onto stack
        movb #4,data4,sp ;fourth parameter onto stack
        rts
*****calling sequence*****
leas -4,sp ;allocate space for results
jsr module
pula      ;first parameter from stack
staa first
pula      ;second parameter from stack
staa second
pula      ;third parameter from stack
staa third
pula      ;fourth parameter from stack
staa fourth
```

There are two approaches for returning multiple parameters in C. In the first approach, we pass a pointer to the module and return the parameters through the pointer.

```
void module(short *pt){
    (*pt)=1;  *(pt+1)=2;  *(pt+2)=3;  *(pt+3)=4;
}
void main(void){ short data[4];
    module(data);}

```

In the second approach we create a structure, and return the parameters within the structure:

```
struct data{ short first,second,third,fourth;};
typedef struct data dataType;
dataType module(void){ dataType myData;
    myData.first=1;  myData.second=2;  myData.third=3;  myData.fourth=4;
    return myData;}
void main(void){ dataType theData;
    theData=module();}

```

### 2.5.3 Dividing a Software Task into Modules

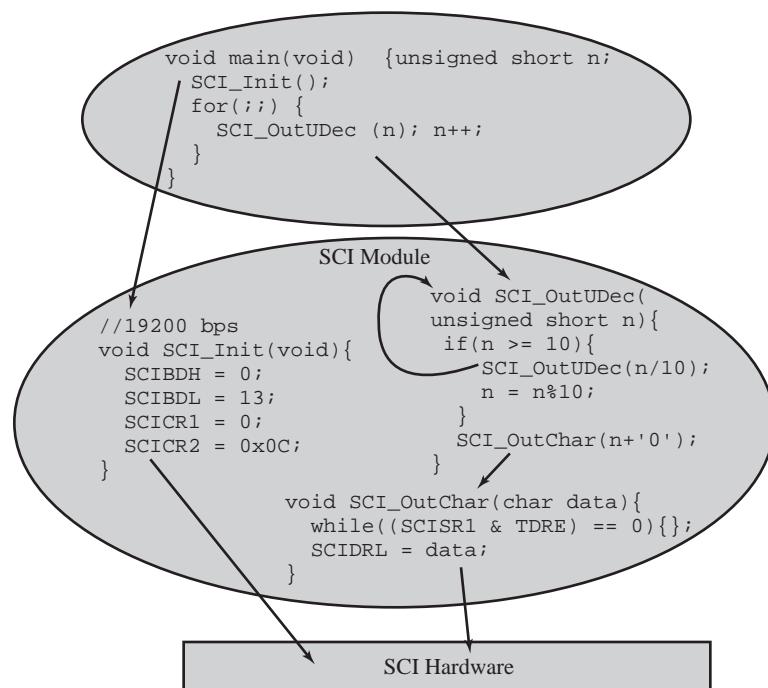
The overall goal of modular programming is to enhance clarity. The smaller the task, the easier it will be to understand. *Coupling* is defined as the influence one module's behavior has on another module. To make modules more independent we strive to minimize coupling. Obvious and appropriate examples of coupling are the I/O parameters explicitly passed from one module to another. On the other hand, information stored in shared global variables can be quite difficult to track. In a similar way shared accesses to I/O ports can also introduce unnecessary complexity. Global variables cause coupling between modules that complicates the debugging process, because now the modules may not be able to be separately tested. On the other hand, we must use global variables to pass information into and out of an interrupt service routine and from one call to an interrupt service routine to the next call. Another problem specific to embedded systems is the need for fast execution, coupled with the limited support for local variables. On many microcontrollers it is possible, but inefficient, to implement local variables on the stack. Consequently, many programmers opt for the less elegant, yet faster approach of global variables. When passing information through global variables is required, it is better to use a well-defined abstract technique like a FIFO queue. We assign a logically complete task to each module. The module is logically complete when it can be separated from the rest of the system and placed into another application. The interfaces are extremely important. The interfaces determine the policies of our modules. In other words, the interfaces define the operations of our software system. The interfaces also represent the coupling between modules. In general we wish to minimize the amount of information passing (i.e., bandwidth) between the modules yet maximize the number of modules. Of the following three objectives when dividing a software project into subtasks, only the first one really matters:

- Make the software project easier to understand
- Increase the number of modules
- Decrease the interdependency (minimize coupling)

We can develop and connect modules in a hierarchical manner: Construct new modules by combining existing modules. In Figure 2.33, the modules of the software project are organized into a call graph. An arrow points from the calling routine to the module it calls. The I/O ports are organized into groups (e.g., all the serial port I/O registers are in one group). This graph allows us to see the organization of the project. Figure 2.33 shows a call graph for the simple example of outputting decimal numbers via the serial port. To make simpler call graphs on large projects we can combine multiple related subroutines into a single module. The main program is at the top, and the I/O ports are at the bottom. In a hierarchical system the modules are organized both in a horizontal fashion (grouped together by function) and in a vertical fashion (decisions on overall policies at the top and

**Figure 2.33**

A call graph showing how the main program outputs numbers.



implementation details at the bottom). Since one of the advantages of breaking a large software project into subtasks is concurrent development, it makes sense to consider concurrency when dividing the tasks. In other words, the modules should be partitioned in such a way that multiple programmers can develop the subtasks as independently as possible. On the other hand, careful and constant supervision is required as interfaces are designed and modules are connected together.

**Observation:** Recursive functions do not fall into this hierarchical calling structure.

**Observation:** If module A calls module B and module B calls module A, then you have created a special situation that must account for these mutual calls.

In Figure 2.33, `SCI_Init` will initialize the serial port interface, establishing the baud rate and protocol format. `SCI_OutChar` transmits a single 16-bit byte using the serial port. `SCI_OutDec` accepts an 8-bit unsigned byte and calls `SCI_OutChar` multiple times to transmit the three ASCII characters. For example, if  $n=145$ , then the three ASCII characters will be \$31, \$34, and \$35. The details of this example will be presented later in Chapters 3 and 7.

There are two approaches to hierarchical programming. The top-down approach starts with a general overview, like an outline of a paper, and builds refinement into subsequent layers. A top-down programmer was once quoted as saying:

"Write no software until every detail is specified"

It provides a better global approach to the problem. Managers like top-down because it gives them tighter control over their workers. The top-down approach works well when an existing operational system is being upgraded or rewritten. On the other hand, the bottom-up approach starts with the smallest detail, building up the system "one brick at a time." The bottom-up approach provides a realistic appreciation of the problem because we often cannot appreciate the difficulty or the simplicity of a problem until we have tried it. It allows programmers to start coding immediately and gives programmers more input into the design. For example, a low-level programmer may be able to point out features that are not possible and suggest other

features that are even better. Some software projects are flawed from their conception. With bottom-up design, the obvious flaws surface early in the development cycle.

We believe bottom-up is better when designing a complex system and specifications are open-ended. On the other hand, top-down is better when you have a very clear understanding of the problem specifications and the constraints of your computer system. The **TExaS** simulator was actually written twice. The first pass was programmed bottom-up and served only to provide a clear understanding of the problem and the features and limitations of our hardware. We literally threw all the source code in the trash and programmed the second version in a top-down manner.

One of the biggest mistakes beginning programmers make is the inappropriate usage of I/O calls (e.g., screen output and keyboard input). Our explanation for their foolish behavior is that they haven't had the experience yet of trying to reuse software they have written for one project in another project. For example, assume you wrote and tested a function that found the median of three numbers. The goal of this first program was to display the result, so you solved it as shown in Program 2.15.

```
unsigned int Median (unsigned int u1, unsigned int u2, unsigned int u3) { unsigned int result;
printf("The inputs are %d, %d, %d.\n",u1,u2,u3);
if(u1>u2)
    if(u2>u3)    result=u2;      // u1>u2,u2>u3      u1>u2>u3
    else
        if(u1>u3) result=u3;      // u1>u2,u3>u2,u1>u3 u1>u3>u2
        else      result=u1;      // u1>u2,u3>u2,u3>u1 u3>u1>u2
    else
        if(u3>u2)    result=u2;      // u2>u1,u3>u2      u3>u2>u1
        else
            if(u1>u3) result=u1;      // u2>u1,u2>u3,u1>u3 u2>u1>u3
            else      result=u3;      // u2>u1,u2>u3,u3>u1 u2>u3>u1
printf("The median is %d.\n",result);
return(result);}
```

### Program 2.15

A median function that is not very portable.

Software portability of this function is diminished because it is littered with `printf`'s. To use this function in another situation, you will almost certainly have to remove the `printf` statements. In general we avoid interactive I/O at the lowest levels of the hierarchy; rather we return data and flags and let the higher-level program do the interactive I/O. Often we add keyboard input and screen output calls when testing our software. It is important to remove the I/O that is not directly necessary as part of the module function. This allows you to reuse these functions in situations where screen output is not available or appropriate. Obviously screen output is allowed if that is the purpose of the routine.

**Common error:** Performing unnecessary I/O in a subroutine makes it harder to reuse at a later time.

From a formal perspective, I/O devices are considered as global because I/O devices reside permanently at fixed addresses. From a syntactic viewpoint any module has access to any I/O device. To reduce the complexity of the system we will restrict the number of modules that actually do access the I/O device. It will be important to clarify which modules have access to I/O devices and when they are allowed to access it. When more than one module accesses an I/O device, it is important to develop ways to arbitrate (which module goes first if two or more want to access simultaneously) or synchronize (make a second

module wait until the first is finished). These arbitration issues will be presented in detail in both Chapters 4 and 5.

*Information hiding* is similar to minimizing coupling. It is better to separate the mechanisms of software from its policies. We should separate what the function does (the relationship between its inputs and outputs) from how it does it. It is good to hide certain inner workings of a module, and simply interface with the other modules through the well-defined I/O parameters. For example, we could implement a FIFO by maintaining the current byte count in a global variable, CNT. A good module will hide how CNT is implemented from its users. If the user wants to know how many bytes are in the FIFO, it calls one of the FIFO routines that returns the count. A badly written module will not hide CNT from its users. The user simply accesses the global variable CNT. If we update the FIFO routines, making them faster or better, we might have to update all the programs that access CNT, too. The object-oriented programming environments provide well-defined mechanisms to support information hiding. This separation of policies from mechanisms is discussed further in Section 2.6 on layered software.

The *keep it simple stupid* approach tries to generalize the problem so that it fits an abstract model. Unfortunately, the person who defines the software specifications may not understand the implications and alternatives. As software developers, we always ask ourselves these questions:

- “How important is this feature?”
- “What if it worked this different way?”

Sometimes we can restate the problem to allow for a simpler (and possibly more powerful) solution.

We can classify the coupling between modules as highly coupled, loosely coupled, or uncoupled. A highly coupled system is not desirable, because there is a great deal of interdependence between modules. A loosely coupled system is optimal, because there is some dependence but interconnections are weak. An uncoupled system, one with no interconnections at all, is typically inappropriate in an embedded system, because all components should be acting towards a common objective. There are three ways in which modules can be coupled. The natural way in which modules are coupled is where one module calls or invokes a function in a second module. This type of coupling, called *invocation coupling*, can be visualized with a call graph. A second way modules can be coupled is by data transfer. If information flows from one module to another, we classify this as *bandwidth coupling*. The bandwidth, which is the information transfer rate, is a quantitative measure of coupling. Bandwidth coupling can be visualized with a data flow graph. The third type of coupling, called *control coupling*, occurs when actions in one module affect the control path within another module. For example, Module A sets a global flag, and Module B uses the global flag to decide its execution path. Control coupling is hard to visualize and hard to debug. Therefore, it is a poor design to employ this type of coupling.

Another way to categorize coupling is to examine how information is passed or shared between modules. We will list the mechanisms from poor to excellent. It is extremely poor design to allow Module A directly modify data or flags within Module B. Similarly poor design is to organize important data into a common shared global space and allow modules to read and write these data. It is okay to allow Module A to call Module B and pass it a control flag. This control flag will in turn affect the execution within Module B. It is good design to have one module pass data to another module. A *stamp* is defined as structured data passed from one module to another. *Primitive data* passed between modules is unstructured.

Coupling is a way to describe how modules connect with each other, but it is also important to analyze how various components within one module interact with each other. *Cohesion* is the degree of interrelatedness of internal parts within the module. In general, we wish to maximize cohesion. A cohesive module has all components of the module

directed towards and essential for the same task. It is also important to analyze how components are related as we design modules. *Coincidental cohesion* occurs when components of the module are unrelated, resulting poor design. Examples of coincidental cohesion would be a collection of frequently used routines, collection of routines written by a single programmer, or a collection of routines written during a certain time interval. *Logical cohesion* is a grouping of components into a single module (because they perform similar functions). An example, of logical cohesion is a collection of serial output, LCD output, and printer output routines into one module (because all routines perform output). Organizing modules in this fashion is a poor design and results in modules are hard to reuse. *Temporal cohesion* combines components if they are connected in time sequence. If we are baking bread, we activate the yeast in warm water in one bowl, then we combine the flour, sugar, and spices in another bowl. These two steps are connected only in a sense that we first do one, then we do another when making bread. If we were making cookies, we would need the flour, sugar, and spices but not the yeast. We want to mix and match existing modules to create new designs, as such, we expect the sequence of module execution to change. Another poor design, called *procedural cohesion*, groups functions together in order to ensure mandatory ordering. For example, an embedded system might have an input port, an output port, and a timer module. In order to work properly, all three must be initialized. It would be hard to reuse code if we placed all three initialization routines into one module.

We next present appropriate reasons to group components into one module. *Communicational cohesion* exists when components operate on the same data. In Chapter 12, we will learn how to collect real time data, and in Chapter 15, we will learn ways to filter the data. An example of communicational cohesion would be a collection of routines that filter and extract features from the data. *Sequential cohesion* occurs when components are grouped into one module, because the output from one component is the input to another component. Sequential cohesion is a natural consequence of minimizing bandwidth between modules. An example of sequential cohesion is a fuzzy logic controller, which we will learn about in Chapter 13. This controller has five stages: crisp input, fuzzification, rules, defuzzification, and crisp output. The output of each stage is the input to the next stage. The input bandwidth to the controller and the output bandwidth from the controller can be quite low, but the amount of information transferred between stages can be thousands of times larger. The best kind of cohesion is *functional cohesion*, where all components combine to implement a single objective, and each component has a necessary contribution to the objective. I/O device drivers, which are a collection of routines for a single I/O device, exhibit functional cohesion.

Another way to classify good and bad modularity is to observe fan-in and fan-out behavior. In a data flow graph, the tail of an arrow defines a data output, and the head of an arrow defines a data input. The *fan-in* of a module is the number of other modules that have direct control on that particular module. Fan-in can be visualized by counting the number of arrowheads that terminate on the module in the data flow graph, shown previously in Figure 1.9. The fan-out of a module is number of other modules directly controlled by this module. Fan-out can be visualized by counting the number of tails of arrows that originate on the module in the data flow graph. In general, systems with high fan-out are a poor design, because that one module may constitute a bottleneck or a critical safety path. In other words, the module with high fan-out is probably doing too much, performing the tasks that should be distributed to other modules. High fan-in is not necessarily a poor design, depending on the application.

## 2.5.4 Rules for Developing Modular Software in Assembly Language

The objective of this section is to present modular design rules and illustrate these rules with assembly language examples. This set of rules is meant to guide, not control. In other words, the rules serve as general guidelines rather than as fundamental law.

*The single entry point is at the top.* In assembly language, we place a single entry point at the first line of the code. This guarantees that registers will be saved and local variables will be properly allocated on the stack. By default, C functions

have a single entry point. Placing the entry point at the top provides a visual marker for the beginning of the subroutine.

*The single exit point is at the bottom.* We prefer to use a single exit point as the last line of the subroutine. Many good programmers employ multiple exit points for efficiency reasons. In general, we must guarantee that the registers, stack, and return parameters are at a similar and consistent state for each exit point. In particular we must deallocate local variables properly. If you do employ multiple exit points, then we suggest you develop a means to visually delineate where one subroutine ends and the next one starts. You could use one line of comments to signify the start of a subroutine and a different line of comments to show the end of it, as in Program 2.16.

### Program 2.16

An assembly subroutine that uses comments to delineate its beginning and end.

```
;*****Abs*****
; Input: RegA is signed 8 bit
; Output: Reg A is absolute value 0 to 127
Abs: tsta    ; already positive?
      bpl ok
      nega
ok   rts
* -----end of Abs -----
```

**Observation:** Having the first and last lines of a module be the entry and exit points makes it easier to debug, because it will be easy to place debugging instruments (like breakpoints).

**Common error:** If you place a debugging breakpoint on the last `rts` of a subroutine with multiple exit points, then sometimes the subroutine will return without generating the break.

**Write structured programs.** A structured program is one that adheres to a strict list of program structures (e.g. sequence, if-then, and do-while). When we program in C (with the exception of `goto`, which, by the way, you should never use), we are forced to write structured programs because of the syntax of the language. One technique for writing structured assembly language is to adhere to the same strict list of program structures available in C. In other words, restrict the assembly language branching to configurations that mimic the software behavior of `if`, `if-else`, `do-while`, `while`, `for`, and `switch`. Assembly language examples of these control structures are included with the **TEXaS** simulator in the files `UIF.RTF`, `SIF.RTF`, `WHILE.RTF`, and `FOR.RTF`. Structured programs are much easier to debug, because execution proceeds only through a limited number of well-defined pathways. When we reuse existing assembly branching structures, then our debugging can focus more on the overall function and less on how the details are implemented.

**The registers must be saved.** When working on a software team, it is important to establish a rule whether or not subroutines will save/restore registers. Establishing this convention is especially important when a mixture of assembly and C is being used or if the software project remains active for long periods of time. It is safest to save and restore registers that are modified (most programmers do not save/restore the CCR). Exceptions to this rule can be made for those portions of the code where speed is most critical.

**Common error:** If the calling routine expects a subroutine to save/restore registers and it doesn't, then information will be lost.

**Observation:** If the calling routine does not expect a subroutine to save/restore registers and it does, then the system executes a little slower and the object code is a little bigger than it could be.

**Common error:** When a mixture of C and assembly language programs is integrated, then an error may occur when the compiler is upgraded because there may be a change if registers are saved/restored or in how parameters are passed.

*Use high-level languages whenever possible.* It may seem odd to have a rule about high-level languages in a section about assembly language programming. In general, we should use high-level languages when memory space and execution speed are less important than portability and maintenance. When execution speed is important, you could write the first version in a high-level language, run a profiler (which will tell you which parts of your program are executed the most), then optimize the critical sections by writing them in assembly language. If a C language implementation just doesn't run fast enough, you could consider a more powerful compiler or a faster microcomputer.

*Minimize conditional branching.* Every time software makes a conditional branch, there are two possible outcomes that must be tested (branch or not branch). For example, assume we wish to add two 16-bit numbers  $u_3 = u_2 + u_1$ . A conditional branch could be avoided by solving the problem in another way, as shown in Program 2.17.

### Program 2.17

Sometimes we can remove a conditional branch and simplify the program.

<pre>; no conditional branch add16b ldaa u1+1 ;lsb     adda u2+1     staa u3+1     ldaa u1    ;msb     adca u2     staa u3     rts</pre>	<pre>; uses conditional branch add16a ldaa u1+1 ;lsb     adda u2+1     staa u3+1     ldaa u1    ;msb     bcc  noc     inca      ;carry     noc     adda u2     staa u3     rts</pre>
--	--

**Observation:** Software can be made easier to understand by reworking the approach to reduce the number of conditional branches.

## 2.6 Layered Software Systems

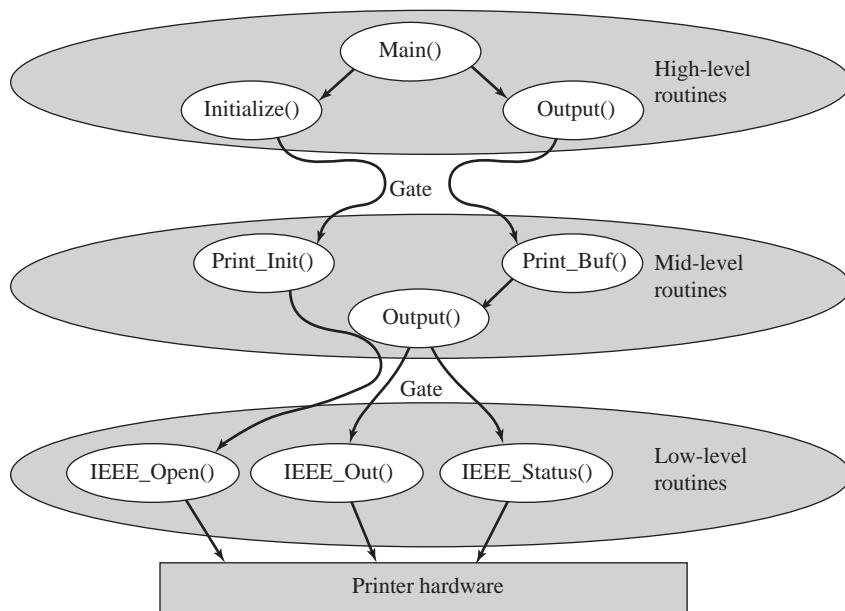
As the size and complexity of our software systems increase, we learn to anticipate the changes that our software must undergo in the future. In particular, we can expect to redesign our system to run on newer and more powerful hardware platforms. A similar expectation is that better algorithms may become available. The objective of this section is to use a layered software approach to facilitate these types of changes.

We can use the call graph defined in Section 2.5 to visualize software layers. A module in a layer can call a module within the same layer or a module in a layer below it. Some layered systems restrict the calls only to modules within the same layer or to a module in the most adjacent layer below it. If we place all the modules that access the I/O hardware in the bottommost layer, we can call this layer a *hardware abstraction layer* (HAL). This bottommost layer also is called *board-support package* (BSP), where I/O devices are referenced in an abstract manner. Each layer of modules only calls modules of the same or lower levels, but not modules of higher level. Usually the top layer consists of the main program. In

a multithreaded environment (e.g., Unix, Windows) there can be multiple main programs at the topmost level, but for now assume there is only one main program. The arrows in Figure 2.34 point from the calling module to the module it calls.

**Figure 2.34**

A layered approach to interfacing an IEEE488 parallel port printer. The bottom layer is the BSP.



To develop a layered software system we begin with a modular system. The main advantage of layered software is the ability to separate the modules into groups or layers such that one layer may be replaced without affecting the other layers. For example, you could change which microcontroller you are using by modifying the low level without any changes to the middle or high levels. Figure 2.34 depicts a layered implementation of a printer interface. In a similar way, you could replace the IEEE488 printer with a serial printer by replacing the bottom two layers. If we were to employ buffering and/or data compression to enhance communication bandwidth, then these algorithms would be added to the middle level. A layered system should allow you to change the implementation of one layer without requiring redesign of the other layers.

A *gate* is used to call from a higher- to a lower-level routine. Another name for this gate is *application program interface* (API). The gates provide a mechanism to link between the layers. Because the size of the software on an embedded system is small, it is possible and appropriate to implement a layered system using standard function calls by simply assembling/compiling all software together. We will see in the next section that the gate can be implemented by creating a header file with prototypes to public functions.

The following rules apply to layered software systems:

1. A module may make a simple call to other modules in the same layer.
2. A module may make a call to a lower-level module only by using the gate.
3. A module may not directly access any function or variable in another layer (without going through the gate).
4. A module may not call a higher-level routine.
5. A module may not modify the vector address of another level's handler(s). For example:

The software interrupt handler address, memory contents at \$FFF6, is specified by the middle level. The low-handler address, memory contents at \$FF80, is specified by the low level.

6. (Optional) A module may not call farther down than the immediately adjacent lower level.
7. (Optional) All I/O hardware access is grouped in the lowest level.
8. (Optional) All user interface I/O (InString, OutString, OutDec, etc.) is grouped in the highest level unless it is the purpose of the module itself to do such I/O.

The purpose of rule 6 is to allow modifications at the low layer to not affect operation at the highest layer. On the other hand, for efficiency reasons you may wish to allow module calls farther down than the immediately adjacent lower layer. To get the full advantage of layered software, it is critical to design functionally complete interfaces between the layers. The interface should support all current functions as well as provide for future expansions.

## 2.7 Device Drivers

### 2.7.1 Basic Concept of Device Drivers

A device driver consists of the software routines that provide the functionality of an I/O device. The driver consists of the interface routines that the operating system or software developer's program calls to perform I/O operations as well as the low-level routines that configure the I/O device and perform the actual I/O. The issue of the separation of policy from mechanism is very important in device driver design. The policies of a driver include the list of functions and the overall expected results. In particular, the policies can be summarized by the interface routines that the operating system or software developer can call to access the device. The mechanisms of the device driver include the specific hardware and low-level software that actually perform the I/O. As an example, consider the variety of mass storage devices that are available. Floppy disk, RAM disks, integrated device electronics (IDE) hard drive, Serial Advanced Technology Attachment (SATA) hard drive, flash drive and even a network can be used to save and recall data files. A simple serial port system might have the following C level interface functions, as explained in the following prototypes:

```
void SCI_Init(unsigned short baudRate); // Enable serial port
char SCI_InChar(void); // Input an ASCII character
void SCI_InString(char *buffer, unsigned short maxSize); // Input a String
unsigned short SCI_InUDec(void); // Input a 16-bit number
void SCI_OutChar(char letter); // Output an ASCII character
void SCI_OutUDec(unsigned short number); // Output a 16-bit number
void SCI_OutString(char *buffer); // Output String
```

Building a HAL or BSP is the same idea as separation of policy from mechanism. A diagram of this layered concept was shown in Figure 2.34. In the above file example, a HAL or BSP would treat all the potential mass storage devices through the same software interface. Another example of this abstraction is the way some computers treat pictures on the video screen and pictures printed on the printer. With the abstraction layer, the software developer's program draws lines and colors by passing the data in a standard format to the device driver, and the operating system redirects the information to the video graphics board or color laserwriter as appropriate. This layered approach allows one to mix and match hardware and software components but does suffer some overhead and inefficiency.

Low-level *device drivers* normally exist in the basic I/O system (BIOS) ROM and have direct access to the hardware. They provide the interface between the hardware and the rest of the software. Good low-level device drivers allow:

1. New hardware to be installed
2. New algorithms to be implemented
  - a. Synchronization with busy-waiting, interrupts, or DMA
  - b. Error detection and recovery methods
  - c. Enhancements like automatic data compression

3. Higher-level features to be built on top of the low level
  - a. Operating system features like blocking semaphores
  - b. Additional features like function keys

and still maintain the same software interface. In larger systems like the Workstation and IBM PC, where the low-level I/O software is compiled and burned in ROM separate from the code that will call it, it makes sense to implement the device drivers as software interrupts (SWIs) and specify the calling sequence in assembly language. We define the “client programmer” as the software developer who will use the device driver. In embedded systems like we use, it is okay to provide `device.H` and `device.C` files that the client programmers can compile with their application. In a commercial setting, you may be able to deliver to the client only the `device.H` together with the object file. *Linking* is the process of resolving addresses to code and programs that have been compiled separately. In this way, the routines can be called from any program without requiring complicated linking. In other words, when the device driver is implemented with a software interrupt, the linking is simple. In our embedded system, the compiler will perform the linking. The device driver software is grouped into four categories. Protected items can only be directly accessed by the device driver itself, and public items can be accessed by the client.

## 2.7.2 Design of a Serial Communications Interface (SCI) Device Driver

The concept of a device driver can be illustrated with the following design of a serial port device driver. There are typically four components of a device driver. In this section, the contents of the header file (`SCI.h`) will be presented, and the implementations will be developed in the next chapter.

**1. Data structures: global (private)** The first component of a device driver is private global data structures. To be private means only programs within the driver itself may directly access these variables. If the user of the device driver (e.g., a client) needs to read or write to these variables, then the driver will include public functions that allow appropriate read/write functions. One example of a private global variable might be an `OpenFlag`, which is true if the serial port has been properly initialized. The implementation developed in Chapter 3 will have no private global variables, but the SCI implementation developed in Chapter 7 will include a first-in-first-out (FIFO) queue, including the functions `Fifo_Init`, `Fifo_Put`, and `Fifo_Get`.

**2. Initialization routines (public, called by the client once in the beginning)** The second component of a device driver includes the public functions used to initialize the device. To be public means the user of this driver can call these functions directly. A prototype to public functions will be included in the header file (`SCI.h`). The names of public functions will begin with `SCI_`. The purpose of this function is to initialize the SCI hardware.

```
-----SCI_Init-----
// Initialize Serial port SCI
// Input: baudRate is the baud rate in bits/sec
// Output: none
void SCI_Init(unsigned short baudRate);
```

**3. Regular I/O calls (public, called by client to perform I/O)** The third component of a device driver consists of the public functions used to perform input/output with the device. Because these functions are public, prototypes will be included in the header file (`SCI.h`). The input functions are grouped, followed by the output functions.

```

-----SCI_InChar-----
// Wait for new serial port input
// Input: none
// Output: ASCII code for key typed
char SCI_InChar(void);

-----SCI_InString-----
// Wait for a sequence of serial port input
// Input: maxSize is the maximum number of characters to look for
// Output: Null-terminated string in buffer
void SCI_InString(char *buffer, unsigned short maxSize);

-----SCI_InUDec-----
// InUDec accepts ASCII input in unsigned decimal format
// and converts to a 16-bit unsigned number (0 to 65535)
// Input: none
// Output: 16-bit unsigned number
unsigned short SCI_InUDec(void);

-----SCI_OutChar-----
// Output 8-bit to serial port
// Input: letter is an 8-bit ASCII character to be transferred
// Output: none
void SCI_OutChar(char letter);

-----SCI_OutString-----
// Output String (NULL termination)
// Input: pointer to a NULL-terminated string to be transferred
// Output: none
void SCI_OutString(char *buffer);

-----SCI_OutUDec-----
// Output a 16-bit number in unsigned decimal format
// Input: 16-bit number to be transferred
// Output: none
// Variable format 1-5 digits with no space before or after
void SCI_OutUDec(unsigned short number);

```

**4. Support software (private).** The last component of a device driver is private functions. Because these functions are private, prototypes will not be included in the header file (`SCI.H`). We place helper functions and interrupt service routines in the category.

Notice that this SCI example implements a layered approach, similar to Figure 2.33. The low-level functions provide the mechanisms and are protected (hidden) from the client programmer. The high-level functions provide the policies and are accessible (public) to the client. When the device driver software is separated into `SCI.H` and `SCI.C` files, you need to pay careful attention as to how many details you place in the `SCI.H` file. A good device driver separates the policy (overall operation, how it is called, what it returns, what it does, etc.) from the implementation (access to hardware, how it works, etc.) In general, you place the policies in the `SCI.H` file (to be read by the client) and the implementations in the `SCI.C` file (to be read by you and your coworkers). Think of it this way: if you were to write commercial software that you wished to sell for profit and you delivered the `SCI.H` file and its compiled object file, how little information could you place in the `SCI.H` file and still have the software system be fully functional? In object-oriented terms the policies will be public, and the implementations will be private.

**Observation:** A layered approach to I/O programming makes it easier for you to upgrade to newer technology.

**Observation:** A layered approach to I/O programming allows you to do concurrent development.

## 2.8 Object-Oriented Interfacing

### 2.8.1 Encapsulated Objects Using Standard C

Object-oriented software development in C++ involves three fundamental issues: encapsulation, polymorphism, and inheritance. *Encapsulation* is the grouping of functions and variables into a single class. C++ provides the mechanisms to implement modular software as described in this section. *Polymorphism* is the ability to reuse function names so that the exact operation depends on which class is being operated. *Inheritance* allows you to derive one class upon a previous class, reusing code and extending its functionality. For embedded systems, encapsulation is much more important than polymorphism and inheritance.

The example in this subsection will show you how to write C code that incorporates most of the important issues of encapsulation. This example also illustrates the top-down approach and includes three modules: the LCD interface, and some timer routines (Program 2.18). Notice that function names are chosen to reflect the module in which they are defined. If you are a C++ programmer, consider the similarities between this C function call `LCD_clear()` and a C++ LCD class and a call to a member function `LCD.clear()`. The \*.H files contain function declarations and the \*.C files contain the implementations.

#### Program 2.18

Main program with three modules.

```
#include "HC12.H"
#include "LCD12.H"
#include "Timer.H"

void main(void){ char letter; int n=0;
    LCD_Init();
    Timer_Init();
    LCD_String("LCD");
    Timer_MsWait(1000);
    LCD_clear();
    letter='a'-1;
    while(1){
        if (letter=='z')
            letter='a';
        else
            letter++;
        LCD_putchar(letter);
        Timer_MsWait(250);
        if(++n==16){
            n=0;
            LCD_clear();
        }}}
```

For every function definition, the compiler generates an assembler directive declaring the function's name to be public. This means that every C function is a potential entry point and so can be accessed externally. One way to create private/public functions is to control which functions have declarations. Now let us look inside the `Timer.H` and `Timer.C` files. To implement private and public functions we place the function declarations of the public functions in the `Timer.H` file (Program 2.19).

#### Program 2.19

`Timer.H` header file has public functions.

```
void Timer_Init(void);
void Timer_Wait10ms(unsigned short delay);
```

The implementations of all functions are included in the `Timer.C` file shown earlier as Program 2.10. We can apply this same approach to private and public global variables.

Notice that in this case the global variable, CyclesPerMs, is private and cannot be accessed by software outside the `Timer.c` file (Program 2.20).

### **Program 2.20**

`Timer.c`  
implementation file  
defines all functions.

```
unsigned short CyclesPerMs; // private global
void Timer_Init(void){ // public function
    TSCR1 |=0x80;           // TEN(enable)
    CyclesPerMs = 4000;    // 4000 counts per ms
}
void wait(unsigned short cycles){ // private function
    unsigned short startTime = TCNT;
    while((TCNT-startTime) <= cycles){}; // wait 10ms
}
void Timer_MsWait(unsigned short time){ // public function
    for(;time>0;time--){
        wait(CyclesPerMs); // 1.00ms wait
    }
}
```

## **2.8.2 Object-Oriented Interfacing Using C++**

The three characteristics of object-oriented programming are encapsulation, polymorphism, and inheritance. We defined these terms in Section 2.8.1 and discussed how to encapsulate modules using standard C. In this section we will introduce the concept of object-oriented interfacing using C++ by discussing the software environment on the IBM PC compatible. Then, we will show an example of how C++ might provide support to improve the portability of our embedded system software.

Since its inception C++ has steadily replaced C as the preferred programming environment for developing both system programs and user applications for the *WinTel* platform (Windows operating system on an Intel microprocessor). The reasons for this software evolution (revolution?) stem from the inherent advantages of C++. Some of the fundamental difficulties for developing software for the *WinTel* platform are:

1. We need a common user interface on top of a multitude of similar but not identical computers.
2. The hardware platform makes a fundamental advancement every 6 months.
3. Many hardware and software companies act in concert to produce a product.
4. The newer software must run on the older computers.
5. The older software must run on the newer computers.
6. The hardware/software configurations may change at run time.

Because of these constraints, a layered software model was adopted so that changes in one aspect of the system could be made without having to reengineer the entire system. Objects in C++ allow the programmer to use hardware and software modules without complete knowledge of how they work. The member functions provide a clean yet powerful mechanism to implement the interface between the software modules. Especially at the hardware interface level, classes provide a mechanism for abstraction. In other words, the HAL is a set of C++ objects that define basic input/output operations.

There are some similarities but many differences between the *WinTel* and embedded platforms. If we examine the same six constraints for an embedded microcomputer, we see that only the first constraint is similar. In other words, we are interested in making embedded system software run on multiple microcomputers (code reuse). Software is portable if it is easy to convert it to run on another platform. The other five constraints for the most part do not exist in the embedded system development environment:

2. Embedded microcomputers have a much longer lifetime than a x86 microprocessor.
3. Usually a single company develops the hardware and software.

- 4., 5. Hardware and software are upgraded together.
6. Configurations are usually well-defined at compile time.

### 2.8.3 Portability Using Standard C and C++

Even though assembly and C are currently the primary software development approaches for embedded systems, it is appropriate to consider software development in C++. As a case study, we will address the issue of portability using C and C++. First, we will show an enhanced C software implementation of the Moore FSM first presented in Example 2.1 and Program 2.11. To make this program more portable using C, we create `#define` macros for those parameters that are likely to change.

Programs 2.21 and 2.22 use the same Traffic Light FSM as Program 2.11.

#### Program 2.21

Enhanced C implementation of the Traffic Light FSM.

```
/* 9S12 Port M bits 1,0 are input, Port T bits 5-0 are output */
#define OutPort (*(unsigned char volatile *) (0x0240))
#define OutDDR (*(unsigned char volatile *) (0x0242))
#define InPort  (*(unsigned char volatile *) (0x0250))
#define InDDR   (*(unsigned char volatile *) (0x0252))

STyp *Pt; // state pointer
unsigned char Input;
void main(void){
    Timer_Init(); // enable TCNT
    OutDDR = 0xFF;
    InDDR &= ~0x03;
    Pt = goN;
    while(1){
        OutPort = Pt->Out;
        Timer_Wait10ms(Pt->Time);
        Input = InPort&0x03;
        Pt = Pt->Next[Input];
    }
}
```

```
// a DDRAddress of 1 means fixed output port with no DDR
// a DDRAddress of 0 means fixed input port with no DDR
template <class T> class port{
protected :
    unsigned char *PortAddress; // pointer to data
    unsigned char *DDRAddress; // pointer to data direction register
public : port(unsigned short ThePortAddress, unsigned short TheDDRAddress){
    PortAddress = (unsigned char *)ThePortAddress; // pointer to I/O port
    DDRAddress = (unsigned char *)TheDDRAddress; // pointer to DDR
}
virtual short Initialize(unsigned char data){
    if((int)DDRAddress==1){ // fixed output port
        return(data==0xFF); // OK if initializing all bits to output
    }
    if(DDRAddress==0){ // fixed input port
        return(data==0); // OK if initializing all bits to input
    }
    (*DDRAddress) = data; // configure direction register
    return 1; // successful
}
```

```

virtual void put(unsigned char data){
    if((int)DDRAddress==0) return; // fixed input
    if((*DDRAddress)==0) return; // all input
    (*PortAddress) = data; // output data to port
}
virtual unsigned char get(void){
    return (*PortAddress); // input data from port
}
/* 9S12 Port M bits 1,0 are input, Port T bits 5-0 are output */
port<unsigned char> OutPort(0x0240,0x0242);
port<unsigned char> InPort(0x0250,0x0252);

void main(void){ StateType *Pt; unsigned char Input;
    Pt=SA; // Initial State
    OutPort.Initialize(0xFF); // Make Output port outputs
    InPort.Initialize(0x00); // Make Input port inputs
    Timer_Init(); // Enable TCNT
    while(1){
        OutPort.put(Pt->Out);
        Timer_Wait10ms(Pt->Time); // Time to wait in this state
        Input=InPort.get()&0x03; // Input=0,1,2,or 3
        Pt=Pt->Next[Input];
    }
}

```

**Program 2.22**

C++ implementation of the Traffic Light FSM.

In C++ we will define a class to describe a generic I/O port. The attributes of the port include its address, the existence and address of its data direction register, and its type T. Program 2.22 uses the same FSM and Wait function as Program 2.21.

To configure this software for the 6811, we change the way the I/O ports are blessed.

```

// 6811 PortC bits 1,0 are input, Port B bits 1,0 are output
port<unsigned char> OutPort(0x1004,1); // fixed output port
port<unsigned char> InPort(0x0003,0x0007); // bidirectional port

```

To make this system truly portable we would have to create an object for the timer functions as well. You can derive classes from this class and override (enhance) the I/O functions. Additional member functions can also be added to enhance this C++ object, as in Program 2.23.

**Program 2.23**

Additional member functions for the I/O port class.

```

T operator = (T data){
    put(data); // output to port
    return data; // returns data itself
operator T () (T data){
    return get(); // returns port data
virtual T operator |= (T data){
    put(data |= get()); // read modify write port access
    return data; // returns new data
virtual T operator &= (T data){
    put(data &= get()); // read modify write port access
    return data; // returns new data
virtual T operator ^= (T data){
    put(data ^= get()); // read modify write port access
    return data; // returns new data

```

**Observation:** The issues of software clarity, portability, and modularity are important no matter which programming language you are using.

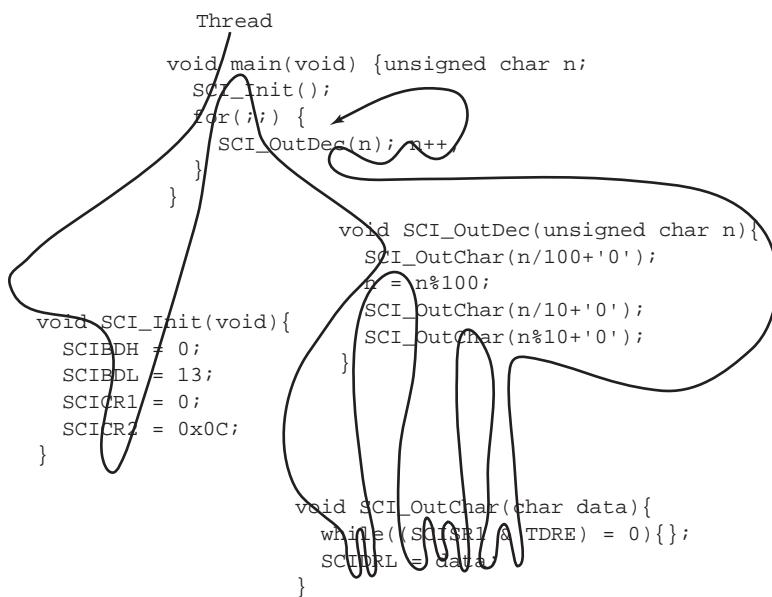
## 2.9 Threads

### 2.9.1 Single-Threaded Execution

Software (e.g., program, code, module, procedure, function, subroutine) is a list of instructions for the computer to execute. A thread, on the other hand, is defined as the path of action of software as it executes. The expression “thread” comes from the analogy shown in Figure 2.35. This simple program prints the 8-bit numbers 000 001 002. . . . If we connect the statements of our executing program with a continuous line (the thread), we can visualize the dynamic behavior of our software.

**Figure 2.35**

Illustration of the definition of a thread.



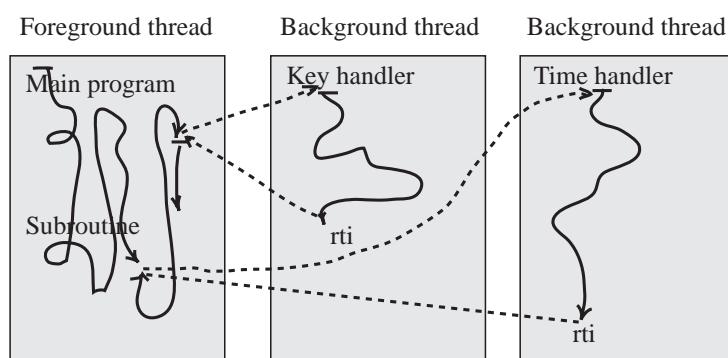
The execution of the main program is called the *foreground thread*. In most embedded applications, the foreground thread executes a loop that never ends. We will learn later that this thread can be broken (execution suspended, then restarted) by interrupts and DMA.

### 2.9.2 Multithreading and Reentrancy

With interrupts we can create multiple threads. Some threads will be created statically, meaning they exist throughout the life of the software, while others will be created and destroyed dynamically. There will usually be one foreground thread running the main program, as in Figure 2.35. In addition to this foreground thread, each interrupt source has its own background thread, which is started whenever the interrupt is requested. Figure 2.36 shows a software system with one foreground thread and two background threads. The “key” thread is invoked whenever a key is touched on the keyboard, and the “time” thread is invoked every 1 ms in a periodic fashion. Because there is but one processor, the currently running thread must be suspended to execute another thread. In Figure 2.36 the suspension of the main program is illustrated by the two breaks in the foreground thread. When a key is touched, the main program is suspended, and a Keyhandler thread is created with an “empty” stack and uninitialized registers. When the Keyhandler is done, it executes `rti` to relinquish control back to the main program. The original stack and registers of the main program will be restored to the state before the interrupt. In a similar way, when the 1-ms timer occurs, the main program is suspended again, and a Timehandler thread

**Figure 2.36**

Interrupts allow us to have multiple background threads.

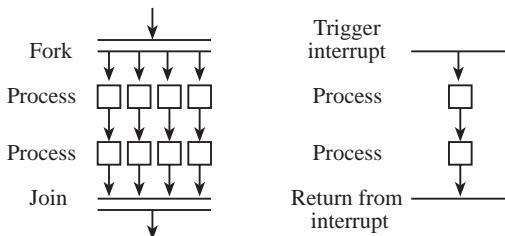


is created with its own “empty” stack and uninitialized registers. We can think of each thread as having its own registers and its own stack area. In Chapter 4, we will discuss in detail this approach to multithreaded programming. In Chapter 5, we will implement a preemptive thread scheduler that will allow our software to have multiple foreground threads and multiple stacks.

*Parallel programming* allows the computer to execute multiple threads at the same time. State-of-the art multi-core processors can execute a separate program in each of its cores. Fork and join are the fundamental building blocks of parallel programming. After a *fork*, two or more software threads will be run in parallel, i.e., the threads will run simultaneously on separate processors. Two or more simultaneous software threads can be combined into one using a *join*. The flowchart symbols for fork and join are shown in Figure 2.37. Software execution after the join will wait until all threads above the join are complete. As an analogy, when a farmer wants to build a barn, he invites his three neighbors over and gives everyone a hammer. The fork operation changes the situation from the farmer working alone to four people ready to build. The four people now work in parallel to accomplish the single goal of building the barn. When the overall task is complete, the join operation causes the neighbors to go home, and the farmer is working alone again.

**Figure 2.37**

Flowchart symbols to describe parallel and concurrent programming.



*Concurrent programming* allows the computer to execute multiple threads, but only one at a time. Interrupts are one mechanism to implement concurrency on real-time systems. Interrupts have a hardware trigger and a software action. An interrupt is a parameterless subroutine call, triggered by a hardware event. The flowchart symbols for interrupts are also shown in Figure 2.37. The trigger is a hardware event signaling it is time to do something. Examples of interrupt triggers we will see in this book include new input data has arrived, output device is idle, and periodic event. The second component of an interrupt-driven system is the software action called an *interrupt service routine* (ISR). The *foreground* thread is defined as the execution of the main program,

and the *background* threads are executions of the ISRs. Consider the analogy of sitting in a comfy chair reading a book. Reading a book is like executing the main program in the foreground. You start reading at the beginning of the book and basically read one page at time in a sequential fashion. You might jump to the back and look something up in the glossary, then jump back to where you were, which is analogous to a function call. Similarly, you might read the same page a few times, which is analogous to a program loop. Even though you skip around a little, the order of pages you read follows a logical and well-defined sequence. Conversely, if the telephone rings, you place a bookmark in the book, and answer the phone. When you are finished with the phone conversation, you hang up the phone and continue reading in the book where you left off. The ringing phone is analogous to hardware trigger and the phone conversation is like executing the ISR.

A program segment is reentrant if it can be concurrently executed by two (or more) threads. In Figure 2.36, we can conceive of the situation where the main program starts executing a subroutine, is interrupted, and the background thread calls that same subroutine. For two threads to share a subroutine, the subroutine must be reentrant. To implement reentrant software, place local variables on the stack and avoid storing into I/O devices and global memory variables. The issue of reentrancy will be covered in detail later in Chapter 4.

## 2.10 Recursion

A recursive program is one that calls itself. Although many algorithms can be defined using recursion, it does require special care when implementing. In particular, recursive subroutines must be reentrant. For some sorting, computational, and database functions, recursion affords a more elegant solution. Recursive algorithms are often easy to prove correct and use less permanent memory but require more temporary stack space and execute slower than nonrecursive algorithms.

There are many types of recursion. The factorial in Program 2.24 is an example of **linear recursion**, because only one call is made to the function within the function. Linear recursive functions are easier to implement iteratively. We draw the execution pattern as a straight or linear path. A **tail recursive** function has the recursive call as the last action taken by the function. A tail recursive function can be implemented in an iterative manner by removing the recursive call and substituting it with a loop. A **binary recursive** function calls itself twice during the course of its execution. The stack size grows exponentially with call depth. So, if you have a binary recursive function, you should use debugging instruments to monitor stack size during execution.

Each time the subroutine is started, a new instantiation occurs. There is a unique set of parameters, registers, and local variables for each instantiation. The stack is a convenient way to separate the parameters and variables of one instantiation from another. In order for the recursive function to finish, there must be a situation where a direct result is generated, which is called the **end condition**. For example, the factorial has two possibilities

Fact(1) = 1	end condition
Fact(n) = n*Fact(n-1) if n>1	recursion

Program 2.24 shows two assembly language implementations of factorial, ignoring the special case of Fact(0). The function iFact uses iteration, and the function rFact uses recursion. It is usually the case that a recursive algorithm can be rewritten in iterative form. Nevertheless, sometimes it is more convenient to implement the algorithm in recursive form.

<pre> ; iterative implementation ; Input: RegD is n ; Output: RegD is Fact(n) iFact tfr D,X      ; counter n loop  dex           ; n=n-1       cpx  #1       ; end condition       bls  done       tfr  X,Y      ; D=r, Y=n       emul          ; r=r*n       bra  loop done  rts           ; RegD=Fact(n) ; recursive implementation ; Input: RegD is n ; Output: RegD is Fact(n) rFact cpd #2      ; end condition       bls  done       pshd          ; save n       subd #1      ; n-1       bsr  rFact   ; RegA=Fact(n-1)       puly          ; RegY=n       emul          ; RegD=n*Fact(n-1) done  rts </pre>	<pre> // iterative implementation unsigned short iFact(unsigned short n){     unsigned short r;     for(r=n; r&gt;2; r--){         n = r*n;     }     return n; }  // recursive implementation unsigned short rFact(unsigned short n){     if(n &lt;= 2){ // end condition         return n;     }     return n*rFact(n-1); // recursion } </pre>
---	---

### Program 2.24

Iterative and recursive implementations of factorial.

Table 2.11 shows the execution time in cycles for these two assembly implementations. Notice that the recursive implementation is slightly shorter, but the execution speed is slightly slower.

**Table 2.11**  
Execution times in cycles for Program 2.25 (including `bsr`).

Input	9S12 Iterative	9S12 Factorial
1	16	14
2	16	14
3	27	36
4	38	58
5	49	80
6	60	102
7	71	124
8	82	146

---

**Example 2.3:** You are given a subroutine, `OutChar`, that outputs one ASCII character. Design a function that outputs a 16-bit unsigned integer.

**Solution:** We will solve this two ways: iteratively and recursively. As always, we ask “What is our starting point?”, “How do we make progress?”, and “When are we done?” The input,  $N$ , is a 16-bit unsigned number (in `RegD`), and we are done when 1 to 5 ASCII characters are displayed, representing the value of  $N$ . The successive refinement approach starts with a general description, then adds details or refinements. The iterative solution has three phases: initialization, creation of digits, and output of the ASCII characters. The digits are created from the remainders occurring by dividing the input,  $N$ , by 10. To get all the digits, we divide by 10 until the quotient is 0. Because the digits are created in the opposite order, each digit will be pushed on the stack during the creation phase and pulled off the stack during the output stage. The counter is needed so the output stage knows how many digits to pull from the stack.

Most recursive functions first check for the end condition. If the end condition is true, it handles the simple case directly. If the end condition is not true, it simplifies the problem (in this case  $N = N/10$ ) and calls itself. Just like the iterative solution, the digits (calculated as R) are calculated in reverse order, and the stack is used to save the intermediate results, so the digits are displayed in proper order.

Program 2.25 shows two implementations of this 16-bit output decimal function. The iterative solution actually has two loops: The first loop determines the digits in opposite order and the second loop outputs the digits in proper order. The recursive solution also uses the stack to calculate the least significant digit first but to output the most significant digit first. There is no fundamental rule that states which is better—iteration or recursion. A good programmer has both in his/her toolbox and uses whichever is easier to understand and easier to debug.

### Program 2.25

Iterative and recursive implementations of output decimal.

<pre> ; iterative method ; Reg D is input, n OutUDec ldy #0 ;RegY= cnt ODloop ldx #10 ;RegD= n     idiv ;RegB= R, digit     pshb ;Save for later     iny ;cnt++     xgdx ;RegD, n=n/10     cpd #0 ;Continue until     bne ODloop ODout pula ;next digit     adda #'0' ;convert ASCII     jsr OutCh     dbne Y,ODout     rts ; recursive method ; Reg D is input, n OutUDec cpd #10 ;end condition     blo end     ldx #10 ;RegD n     idiv ;RegX n= n/10     pshb ;RegB R= n%10     xgdx ;RegD n= n/10     bsr OutUDec     pulb ;RegB R= n%10 end     tba     adda #'0' ;convert ASCII     jsr OutCh out   rts </pre>	<pre> // iterative method void OutUDec(unsigned short n){     unsigned cnt=0;     unsigned char buffer[5];     do{         buffer[cnt] = n%10; // digit         n = n/10;         cnt++;     }     while(n); // repeat until n==0     for(; cnt; cnt--)         OutCh(buffer[cnt-1]+'0');     }  // recursive method void OutUDec(unsigned short n){     if(n &gt;= 10){         OutUDec(n/10); // ms digits         n = n%10; // n is 0-9     }     OutChar(n+'0'); } </pre>
---	---

**Observation:** In general, recursive algorithms are shorter to write but require additional stack space.

## 2.11 Debugging Strategies

All programmers are faced with the need to debug and verify the correctness of their software. In this section we will study hardware-level probes like the logic analyzer and background debug module (BDM); software-level tools like simulators, monitors, and profilers; and manual tools like inspection and print statements.

## 2.11.1 Debugging Tools

Microcomputer-related problems often require the use of specialized equipment to debug the system hardware and software. Useful hardware tools include a logic probe, an oscilloscope, a logic analyzer, and a background debug module. A logic probe is a handheld device with an LED or buzzer. You place the probe on your digital circuit, and the LED/buzzer will indicate whether the signal is high or low. An oscilloscope, or scope, graphically displays information about an electronic circuit where the voltage amplitude versus time is displayed. A scope has one or two channels with many ways to trigger or capture data. A scope is particularly useful when interfacing analog signals using an ADC or DAC. A logic analyzer is essentially a multiple channel digital storage scope with many ways to trigger. As shown in Figure 2.38, we can connect the logic analyzer to digital signals that are part of the system, or we can connect the logic analyzer channels to unused microcontroller pins and add software to toggle those pins at strategic times/places. As a troubleshooting aid, it allows the experimenter to observe numerous digital signals at various points in time and thus make decisions based upon such observations. One problem with logic analyzers is the massive amount of information that it generates. To use an analyzer effectively, one must learn proper triggering mechanisms to capture data at appropriate times, eliminating the need to sift through volumes of output.

**Figure 2.38**  
A logic analyzer and example output.

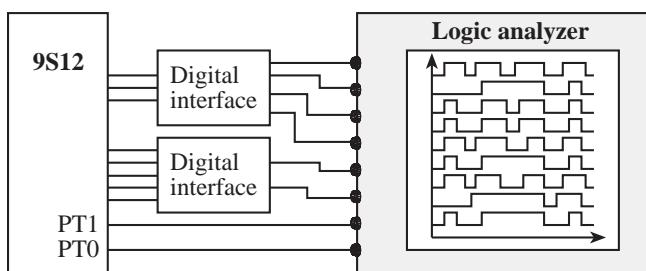
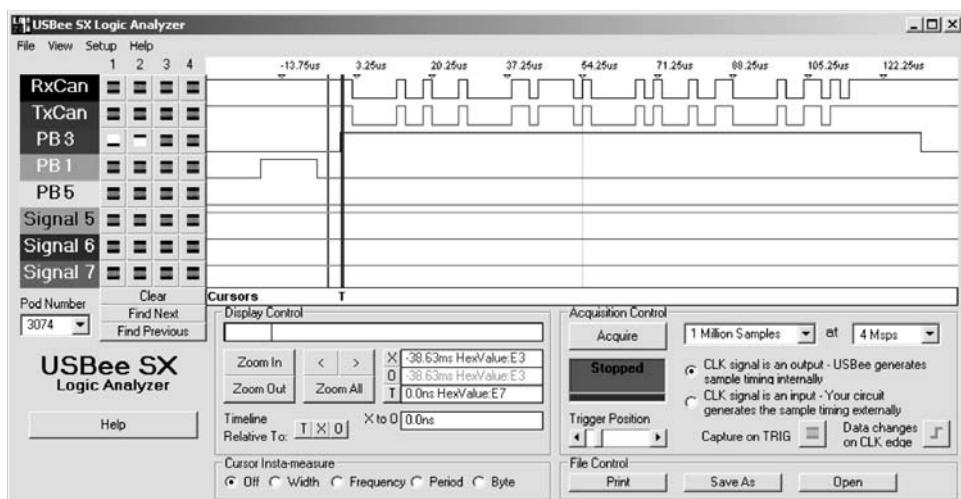


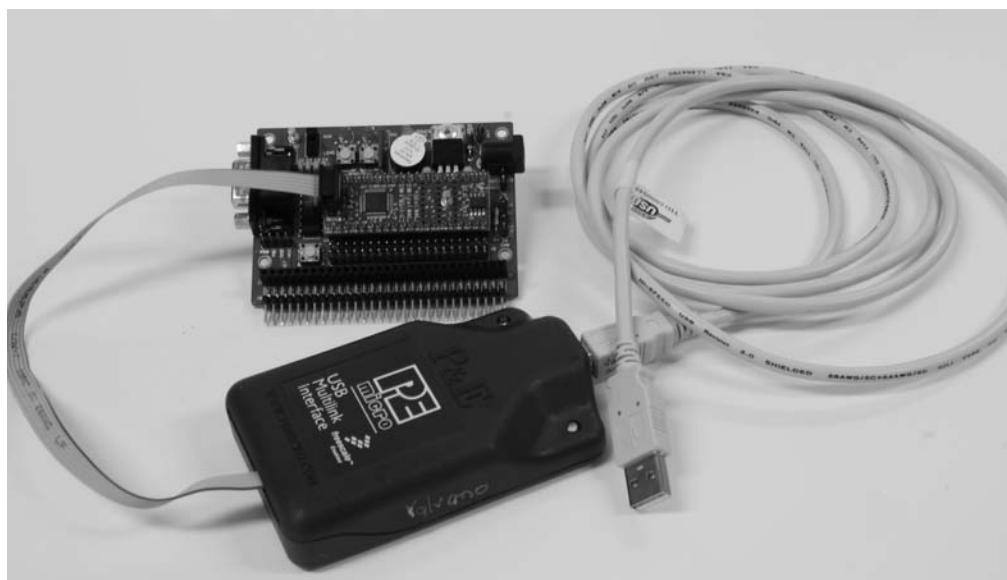
Figure 2.39 shows a logic analyzer output, where signals RxCAN and TxCAN are digital input/output, but signals PB3 PB1 and PB5 are debugging outputs to measuring timing relationships between software execution and digital I/O. The rising edge of PB3 is used to trigger the data collection.

**Figure 2.39**  
USBee SX logic analyzer output.



An emulator is a hardware debugging tool that recreates the input/output signals of the processor chip. To use an emulator, we remove the processor chip and insert the emulator cable into the chip socket. In most cases, the emulator/computer system operates at full speed. The emulator allows the programmer to observe and modify internal registers of the processor. Emulators are often integrated into a personal computer, so that its editor, hard drive, and printer are available for the debugging process.

The only disadvantage of the in-circuit emulator is its cost. To provide some of the benefits of this high-priced debugging equipment, the 9S12 has a *background debug module* (BDM). The BDM hardware exists on the microcomputer chip itself and communicates with the debugging computer via a dedicated serial interface, as shown in Figure 2.40. Although not as flexible as an ICE, the BDM can provide the ability to observe software execution in real-time, the ability to set breakpoints, the ability to stop the computer, and the ability to read and write registers, I/O ports, and memory. The registers only can be observed when the computer is halted, but the memory and I/O ports are accessible while the program is executing.



Courtesy of Jonathan Valvano

**Figure 2.40**

P&E Microcomputer Systems Multilink BDM connected to a DragonFly12 ([www.evbplus.com](http://www.evbplus.com))

**2.11.2 Debugging Theory** Research in the area of program monitoring and debugging has not kept pace with developments in other areas of computer programming. This area is comparatively deficient in the structure and unity now common to programming languages. The area is compartmentalized because of specialized tools. For example, run-time profile generators and execution monitors are tools that have many commands and functions in common with “debuggers,” but in practice, because of tool specialization, a user needs to use two tools: a monitor to examine time behavior and a debugger for functional behavior. This area is also fragmented because of a multiplicity of terms. Terms such as program testing, diagnostics, performance debugging, functional debugging, tracing, profiling, instrumentation, visualization, optimization, verification, performance measurement, and execution measurement have specialized meanings, but they are also used interchangeably, and they often describe overlapping functions. For example, the terms profiling, tracing, performance measurement, or execution measurement may be used to

describe the process of examining a program from a time viewpoint. But tracing is also a term that may be used to describe the process of monitoring a program state or history for functional errors or to describe the process of stepping through a program with a debugger. Usage of these terms among researchers and users vary.

Furthermore, the meaning and scope of the term debugging itself is not clear. We hold the view that the goal of debugging is to maintain and improve software, and the role of a debugger is to support this endeavor. We define the debugging process as testing, stabilizing, localizing, and correcting errors. And in our opinion, although testing, stabilizing, and localizing errors are important and essential to debugging, they are auxiliary processes: The primary goal of debugging is to remedy faults or to correct errors in a program.

Although a variety of program monitoring and debugging tools are available today, in practice it is found that an overwhelming majority of users either still prefer or rely mainly upon “rough and ready” manual methods for locating and correcting program errors. These methods include desk checking, dumps, and print statements, with print statements being one of the most popular manual methods. Manual methods are useful because they are readily available, and they are relatively simple to use. But the usefulness of manual methods is limited: They tend to be highly intrusive, and they do not provide adequate control over repeatability, event selection, or event isolation. A real-time system, where software execution timing is critical, usually cannot be debugged with simple print statements, because the print statement itself will require too much time to execute.

We define a *debugging instrument* as software code that is added to the program for the purpose of debugging. A print statement is a common example of an instrument. Using the editor, we add print statements to our code that either verify proper operation or illustrate the programming errors. A key to writing good debugging instruments is to provide for a mechanism to reliably and efficiently remove them all when the debugging is done. Consider the following mechanisms as you develop your own unique debugging style.

- Place all print statements in a unique column (e.g., first column) so that the only code that exists in this column must be a debugging instrument.
- Define all debugging instruments as functions that all have a specific pattern in their names. In this way, the find/replace mechanism of the editor can be used to find all the calls to the instruments.
- Define the instruments so that they test a run-time global flag. When this flag is turned off, the instruments perform no function. Notice that this method leaves a permanent copy of the debugging code in the final system, causing it to suffer a run-time overhead, but the debugging code can be activated dynamically without recompiling. Many commercial software applications utilize this method because it simplifies “on-site” customer support.
- Use conditional compilation (or conditional assembly) to turn on and off the instruments when the software is compiled. When the compiler supports this feature, it can provide both performance and effectiveness.

The emergence of concurrent languages and the increasing use of embedded real-time systems place further demands on debuggers. The complexities introduced by the interaction of multiple events or time-dependent processes are much more difficult to debug than errors associated with sequential programs. The behavior of non-real-time sequential programs is reproducible: For a given set of inputs their outputs remain the same. In the case of concurrent or real-time programs this does not hold true. Control over repeatability, event selection, and event isolation is even more important for concurrent or real-time environments. **Intrusiveness** is the extent to which the debugging process itself affects the software under test. **Nonintrusive** debugging has no effect, and **minimally intrusive** debugging has a small but irrelevant effect.

The first step of debugging is to *stabilize* the system with the bug. In the debugging context, we stabilize the problem by creating a test routine that fixes (or stabilizes) all the

inputs. In this way, we can reproduce the exact inputs over and over again. Once stabilized, if we modify the program, we are sure that the change in our outputs is a function of the modification we made in our software and not due to a change in the input parameters.

### 2.11.3 Functional Debugging

#### 2.11.3.1 Single Stepping or Trace

Functional debugging involves the verification of I/O parameters. It is a static process where inputs are supplied, the system is run, and the outputs are compared against the expected results. We will present seven methods of functional debugging.

#### 2.11.3.1 Single Stepping or Trace

Many debuggers allow you to set the program counter to a specific address then execute one instruction at a time. The Metrowerks debugger allows you to execute single assembly instructions or single C level instructions. The **TExaS** simulator provides Step, Few, StepOver, and StepOut commands. *Step* is the usual execute one assembly instruction. *Few* will execute some instructions and stop (you can set how many “some” is). *StepOver* will execute one assembly instruction, unless that instruction is a subroutine call, in which case the simulator will execute the entire subroutine and stop at the instruction following the subroutine call. *StepOut* assumes the execution has already entered a subroutine and will finish execution of the subroutine and stop at the instruction following the subroutine call.

#### 2.11.3.2 Breakpoints without Filtering

A **breakpoint** is a mechanism to tag places in our software, which when executed will cause the software to stop.

#### 2.11.3.3 Conditional Breakpoints

One of the problems with breakpoints is that sometimes we have to observe many breakpoints before the error occurs. One way to deal with this problem is the conditional breakpoint. Add a global variable called *count* and initialize it to zero in the ritual. Add the following conditional breakpoint to the appropriate location. And run the system again (you can change the 32 to match the situation that causes the error).

```
if(count==32)
    bkpt
```

Notice that the breakpoint occurs only on the 32nd time the break is encountered. Any appropriate condition can be substituted.

#### 2.11.3.4 Instrumentation: Print Statements

The use of print statements is a popular and effective means for functional debugging. The difficulty with print statements in embedded systems is that a standard “printer” may not be available. Another problem with printing is that most embedded systems involve time-dependent interactions with their external environment. The print statement itself may be so slow that the debugging process itself causes the system to fail. The print statement is very *intrusive*, meaning the debugger itself affects the system being tested. Therefore, this section will focus on debugging methods that do not rely on the availability of a printer.

#### 2.11.3.5 Instrumentation: Dump into Array without Filtering

One of the difficulties with print statements are that they can significantly slow down the execution speed in real-time systems. Many times the bandwidth of the print functions cannot keep pace with the existing system. For example, our system may wish to call a function 1000 times a second (or every 1 ms). If we add print statements to it that require 50 ms to perform, the presence of the print statements will significantly affect the system operation. In this situation, the print statements would be considered extremely intrusive. Another problem with print statements occurs when the system is using the same output hardware for its normal operation, as is required to perform the print function. In this situation, debugger output and normal system output are intertwined.

To solve both these situations, we can add a debugger instrument that dumps strategic information into an array at run time. We can then observe the contents of the array at a later time. One of the advantages of dumping is that the 9S12 BDM module allows you

to visualize memory even when the program is running. So this technique will be quite useful in systems with a BDM.

Assume happy and sad are strategic 8-bit variables. The first step when instrumenting a dump is to define a buffer in RAM to save the debugging measurements. The cnt will be used to index into the buffers. cnt must be initialized to zero, before the debugging begins. The debugging instrument, shown in Program 2.26, saves the strategic variables into the buffer.

### Program 2.26

Instrumentation dump without filtering.

<pre> ; global variables in RAM size    equ 20 buffer  rmb size*2 cnt     rmb 1 ; programs in EEPROM Save pshb     pshx    ; save     ldab cnt     cmpb #size*2 ; full?     beq done     ldx #buffer     abx    ; place to put next     ldab happy     stab 0,x ; save happy     ldab sad     stab 1,x ; save sad     inc   cnt     inc   cnt done pulx     pulb     rts </pre>	<pre> // global variables in RAM #define size 20 unsigned char buffer[size][2]; unsigned int cnt=0;  // dump happy and sad void Save(void){     if(cnt&lt;size){         buffer[cnt][0] = happy;         buffer[cnt][1] = sad;         cnt++;     } } </pre>
---	--

Next, you add jsr Save statements at strategic places within the system. You can either use the debugger to display the results or add software that prints the results after the program has run and stopped.

**Observation:** You should save registers at the beginning and restore them back at the end so that the debugging instrument itself won't cause the software to crash.

#### 2.11.3.6 Instrumentation: Dump into Array with Filtering

One problem with dumps is that they can generate a tremendous amount of information. If you suspect a certain situation is causing the error, you can add a filter to the instrument. A filter is a software/hardware condition that must be true to place data into the array. In this situation, if we suspect the error occurs when the pointer nears the end of the buffer, we could add a filter that saves in the array only when the pointer is above a certain value.

In the example shown in Program 2.27, the instrument saves the strategic variables into the buffer only when sad is greater than 100.

#### 2.11.3.7 Monitor Using Fast Displays

Another tool that works well for real-time applications is the *monitor*. A monitor is an independent output process, somewhat similar to the print statement, but one that executes much faster and thus is much less intrusive. The LCD display can be an effective monitor for small amounts of information. Small LCDs can display up to 16 characters, while the larger ones can hold four lines by 40 characters. The hardware/software interface for such a display will be presented in Chapter 8. You can place one or more LEDs on individual

**Program 2.27**

Instrumentation dump with filter.

<pre> Save pshb     pshx      ; save     ldab sad     cmpb #100     bls done ; only when sad &gt;100     ldab cnt     cmpb #size*2 ; full?     beq done     ldx #buffer     abx      ; place to put next     ldab happy     stab 0,x ; save happy     ldab sad     stab 1,x ; save sad     inc  cnt     inc  cnt done pulx     puls     rts </pre>	<pre> // dump happy and sad void Save(void){     if(sad&gt;100){         if(cnt&lt;size){             buffer[cnt][0] = happy;             buffer[cnt][1] = sad;             cnt++;         }     } } </pre>
--	---

otherwise unused output bits. Software toggles these LEDs to let you know what parts of the program are running. A LED is an example of a Boolean monitor.

Assume an LED is attached to Port B bit 6. Program 2.28 will toggle the LED.

**Program 2.28**

An LED monitor.

<pre> Toggle psha     ldaa PORTB     eora #\$40     staa PORTB     puls     rts </pre>	<pre> void Toggle(void){     PORTB ^= 0x40; // flip LED } </pre>
--	--

Next, you add `jsr Toggle` statements at strategic places within the system. On the 9S12, the DDRB must be initialized so that bit 6 is an output before the debugging begins. You can either observe the LED directly or look at the LED control signals with a high-speed oscilloscope. Toggling an LED is called a *heartbeat*, because it lets you know the software is running.

**Observation:** When using LED monitors it is better to modify just the one bit, leaving the other seven as is. In this way, you can implement additional LED monitors on one port.

**Checkpoint 2.26:** Write a debugging instrument that toggles Port A bit 3.

## 2.11.4 Performance Debugging

Performance debugging involves the verification of timing behavior of our system. It is a dynamic process where the system is run, and the dynamic behaviors of the I/Os are compared against the expected results. We will present two methods of performance debugging, then apply the techniques to measure execution speed.

### Instrumentation Measuring with an Independent Counter, TCNT

There is a 16-bit counter, called TCNT, which is incremented every E clock. There is a prescaler that can be placed between the E clock and the TCNT counter. It automatically rolls over when it gets to \$FFFF. If we are sure the execution speed of our function is less than 65,535 counts, we can use this timer to collect timing information with only a modest amount of intrusiveness.

#### 2.11.4.2 Instrumentation Output Port

Another method to measure real-time execution involves an output port and an oscilloscope. Assume an oscilloscope is attached to Port B bit 6. Program 2.29 can be used to set and clear the bit.

##### **Program 2.29**

Instrumentation output port.

<pre>Set bset PORTB,#\$40       rts Clr bclr PORTB,#\$40       rts</pre>	<pre>void Set(void){     PORTB  = 0x40; } // PB6=1 void Clr(void){     PORTB &amp;= ~0x40; } // PB6=0</pre>
--	---

Next, you add `jsr Set` and `jsr Clr` statements at strategic places within the system. On the 9S12, the DDRB must be initialized so that bit 6 is an output before the debugging begins. You can observe the signal with a high-speed oscilloscope. For example, to measure the execution time of a subroutine called `Calculate`, we stabilize the system by calling it over and over. Using the scope, we can measure the width of the pulse on PB6, which will be the execution time of the subroutine `Calculate`.

```
loop jsr Set
      jsr Calculate ; function under test
      jsr Clr
      bra loop
```

#### 2.11.4.3 Measurement of Dynamic Efficiency

There are three ways to measure dynamic efficiency of our software. To illustrate these three methods, we will consider measuring the execution time of the `sqrt` function presented earlier as Figures 2.31 and 2.32. The first method is to count bus cycles using the assembly listing (Program 2.28). This approach is appropriate only for very short programs and becomes difficult for long programs with many conditional branch instructions. Often this is a very tedious process, but luckily the **TEXaS** assembler will look up and keep a running count of the number of cycles. The assembly pseudo-operation `org *` will reset the cycle counter, shown between the parentheses. A portion of the assembly output is presented in Program 2.30. Notice that the total cycle count for a 9S12 implementation is 70 cycles. At 4 MHz, 70 cycles is 17.5 s. Because the loop (between `next` and `bne next`) is executed exactly three times, the actual time will be 140 cycles, or 35 s. For most programs it is actually very difficult to get an accurate time measurement using this technique.

The second method uses an internal timer called TCNT. Most Freescale microcomputers have this 16-bit internal register that is incremented at the bus frequency. If we are sure the function will complete in a time less than 65,535 bus cycles, then the internal timer can be used to measure execution speed empirically. The assembly language call to the function is modified so that TCNT is read before and after the subroutine call. The elapsed time is the difference. Since the execution speed may be dependent on the input data, it is often wise to measure the execution speed for a wide range of input parameters. There is a slight overhead in the measurement process itself. To be more accurate you could measure this overhead and subtract it from your measurements. Notice that in Program 2.31 the total time including parameter passing is measured. This same technique can also be used in C language programs (Program 2.32).

The third technique can be used in situations where TCNT is unavailable or where the execution time might be larger than 65,535 counts. In this empirical technique we attach an unused output pin to an oscilloscope or to a logic analyzer. We will set the pin high before the call to the function and set the pin low after the function call. In this way a pulse is created on the digital output with a duration equal to the execution time of the function. We

```

$F019          org * ;reset cycle counter
$F019 35      [ 2]( 0)sqrt pshy
$F01A B776    [ 1]( 2)    tsy
$F01C 1B9C    [ 2]( 3)    leas -4,sp      ;allocate t,oldt,s16
$F01E C7      [ 1]( 5)    clrb
$F01F A644    [ 3]( 6)    ldaa s8,y
$F021 2723    [ 3]( 9)    beq done
$F023 C610    [ 1]( 12)   ldab #16
$F025 12      [ 3]( 13)   mul           ;16*s
$F026 6C5C    [ 2]( 16)   std s16,y    ;s16=16*s
$F028 18085F20 [ 4]( 18)   movb #32,t,y  ;t=2.0, initial guess
$F02C 18085E03 [ 4]( 22)   movb #3,cnt,y
$F030 A65F    [ 3]( 26)next ldaa t,y    ;RegA=t
$F032 180E    [ 2]( 29)   tab           ;RegB=t
$F034 B705    [ 1]( 31)   tfr a,x     ;RegX=t
$F036 12      [ 3]( 32)   mul           ;RegD=t*t
$F037 E35C    [ 3]( 35)   addd s16,y  ;RegD=t*t+16*s
$F039 1810    [12]( 38)   idiv          ;RegX=(t*t+16*s)/t
$F03B B754    [ 1]( 50)   tfr x,d     ;RegB=((t*t+16*s)/t)/2
$F03D 49      [ 1]( 51)   lsrd          ;RegB=((t*t+16*s)/t)/2
$F03E C900    [ 1]( 52)   adcb #0
$F040 6B5F    [ 2]( 53)   stab t,y
$F042 635E    [ 3]( 55)   dec cnt,y
$F044 26EA    [ 3]( 58)   bne next
$F046 B767    [ 1]( 61)done tys
$F048 31      [ 3]( 62)   puly
$F049 3D      [ 5]( 65)   rts
$F04A 183E    [16]( 70)   stop

```

**Program 2.30**

Assembly listing from TExaS of the `sqrt` subroutine.

**Program 2.31**

Empirical measurement of dynamic efficiency in assembly language.

```

before rmb 2      ; TCNT value before the call
elapsed rmb 2      ; number of cycles required to execute sqrt
        movw TCNT,before
        movb ss,1,-sp ; push parameter on the stack (binary fixed point)
        jsr sqrt       ; subroutine call to the module "sqrt"
        ins
        stab tt       ; save result
        ldd TCNT      ; TCNT value after the call
        subd before
        std elapsed   ; execute time in cycles

```

**Program 2.32**

Empirical measurement of dynamic efficiency in C language.

```

unsigned short before,elapsed;
void main(void){
    ss=100;
    before=TCNT;
    tt=sqrt(ss);
    elapsed=TCNT-before;
}

```

assume Port B is available and that bit 7 is connected to the scope. By placing the function call in a loop, the scope can be triggered. With a storage scope or logic analyzer, the function need be called only once. Program 2.33 shows the assembly language measurement. Program 2.34 shows the same technique in C language.

**Program 2.33**

Another empirical measurement of dynamic efficiency in assembly language.

```
        movb #$FF,DDRB ; make Port B an output
loop bset PORTB,$$80 ; set PB7 high
        ldaa #100       ; typical input
        jsr sqrt        ; subroutine call to the module "sqrt"
        bclr PORTB,$$80 ; clear PB7 low
        bra loop
```

**Program 2.34**

Another empirical measurement of dynamic efficiency in C language.

```
void main(void){unsigned char ss,tt;
    DDRB = 0xFF; // PB7 is connected to a scope
    ss = 100;
    while(1){
        PORTB |= 0x80; // set PB7 high
        tt = sqrt(ss);
        PORTB &= ~0x80; // clear PB7 low
    }
}
```

## 2.11.5 Profiling

Profiling is similar to performance debugging because both involve dynamic behavior. Profiling is a debugging process that collects the time history of strategic variables. For example, if we could collect the time-dependent behavior of the program counter, then we could see the execute patterns of our software. We can profile the execution of a multiple-thread software system to detect reentrant activity.

### 2.11.5.1 Profiling Using a Software Dump to Study Execution Pattern

In this section, we will discuss software instruments that study the execution pattern of our software. To collect information concerning execution we will add a debugging instrument that saves the time and location in an array (like a dump). By observing this data we can determine both a time profile (when) and an execution profile (where) of the software execution (Program 2.35).

**Program 2.35**

A time/ position profile dumping into a data array.

```
unsigned short time[100];
unsigned short place[100];
unsigned short n;
void profile(unsigned short p){
    time[n]=TCNT; // record current time
    place[n]=p;
    n++;
}
unsigned short sqrt(unsigned short s){ unsigned short t,oldt;
profile(0);
    t=0;           // based on the secant method
    if(s>0) {
profile(1);
        t=32;      // initial guess 2.0
        do{
profile(2);
            oldt=t; // calculation from the last iteration
            t=((t*t+16*s)/t)/2; // t is closer to the answer
        while(t!=oldt); // converges in 4 or 5 iterations
profile(3);
        return t;
    }
}
```

Since the debugging instrument is implemented as a function, we could read the return address off the stack (the place from which it was called). This is a good approach when we are adding/subtracting many debugging instruments.

- 2.11.5.2 Profiling Using an Output Port** In this section, we will discuss a hardware/software combination to visualize program activity. Our debugging instrument will set output port bits. We will place these instruments at strategic places in the software. If we are using a regular oscilloscope, then we must stabilize the system so that the function is called over and over. We connect the output pins to a scope or logic analyzer and observe the program activity (Program 2.36).

### Program 2.36

A time/position profile using two output bits.

```
unsigned int sqrt(unsigned int s){ unsigned int t,oldt;
PTT=0;
    t=0;          // based on the secant method
    if(s>0) {
        PTT=1;
        t=32;      // initial guess 2.0
        do{
            PTT=2;
            oldt=t;  // calculation from the last iteration
            t=((t*t+16*s)/t)/2; // t is closer to the answer
            while(t!=oldt);}   // converges in 4 or 5 iterations
        PTT=3;
        return t;}
```

- 2.11.5.3 Thread Profile** When more than one program (multiple threads) is running, you could use the previous technique to visualize the thread that is currently active (the one running). For each thread, we assign an output pin. The debugging instrument would set the corresponding bit high when the thread starts and clear the bit when the thread stops. We would then connect the output pins to a multiple-channel scope to visualize in real time the thread that is currently running. For an example of this type of profile, run one of the thread.\* examples included with the **TExaS** simulator and observe the logic analyzer.

## 2.12 Exercises

- 2.1** In 16 words or less, describe the meaning of each of the following terms.

- a)** Z bit
- b)** N bit
- c)** V bit
- d)** C bit
- e)** Indexed addressing mode
- f)** Pseudo-op
- g)** Arithmetic right shift
- h)** Return address

- 2.2** Choose from the following possibilities the most accurate description of the V bit in the CCR that occurs after an addition or subtraction. Choose from the following possibilities the most accurate description of the C bit in the CCR that occurs after an addition or subtraction.
- a)** The bit is set on a signed overflow,
  - b)** The bit is set on an unsigned overflow,

- c) The bit is set on a signed underflow,
- d) The bit is set on an unsigned underflow,
- e) The bit is set on either a signed overflow or a signed underflow,
- f) The bit is set on either an unsigned overflow or an unsigned underflow.

**2.3** In 32 words or less, explain how the `bsr` instruction is used to call a subroutine. Then, in 32 words or less, explain how the `rts` instruction is used return back to the place from where the subroutine was invoked.

**2.4** Let N,M,P be three 8-bit unsigned locations. Write assembly code to implement  $P=2^2N+M$  without using the multiply instruction.

**2.5** Let N,M,P be three 8-bit unsigned locations. Write assembly code to implement  $P=5^2N+M$  without using the multiply instruction.

**2.6** Let N,M,P be three 8-bit locations. Write assembly code to implement  $P = (\$73|N)\&M$ .

**2.7** Let PTT be an 8-bit output port. Write assembly code to set bit 6 and clear bit 1.

**2.8** Let PTT be an 8-bit output port. Write assembly code to toggle bit 6 and set bit 2.

**2.9** Consider the result of executing the following three 9S12 assembly instructions.

```
ldaa #20
ldab #10
mul
```

What is the value in Register A after these three instructions are executed?

**2.10** When you add two 8-bit unsigned numbers, an overflow error can occur. Which of the following techniques can be used to handle the problem of overflow? If there is more than one answer, give all answers that could work.

- a) Write software that is friendly.
- b) Write software using structured programming.
- c) Implement a ceiling and floor.
- d) Write software so there is drop-out.
- e) Promote the numbers and perform the addition with this new precision.
- f) Write software to mask the two input data
- g) Demote the numbers and perform the addition with this new precision.
- h) Convert the numbers to signed and perform the addition with signed math.

**2.11** Consider the reasons why one chooses which technique to create a variable.

- a) List three reasons why one would implement a variable using a register.
- b) List three reasons why one would implement a variable on the stack and access it using RegX indexed mode addressing.
- c) List three reasons why one would implement a variable in RAM and access it using extended mode addressing.

**2.12** List three factors that we can use to evaluate the “goodness” of a program.

**2.13** In 32 words or less, explain the differences between a Mealy and Moore FSM. For which types of problems should you implement with Mealy? For which types of problems should you implement with Moore?

**2.14** Assuming the E clock is 8 MHz, write software to initialize TCNT to run at 1 MHz.

**2.15** Assuming the E clock is 24 MHz, write software to initialize TCNT to run at 375 kHz.

**2.16** The original values of some registers/memory locations are specified for the 9S12.

RegA=\$79	RegB=\$AC	RegX=\$2000	RegY=\$4000
[\$0030]=\$A5	[\$0031]=\$B6	[\$3000]=\$C7	[\$3001]=\$D8
[\$2001]=\$11	[\$2002]=\$22	[\$2003]=\$33	[\$2004]=\$44
[\$2006]=\$66	[\$4000]=\$40	[\$4001]=\$41	[\$4002]=\$42
			[\$4003]=\$43

- a) Specify the addressing mode, the effective address, the opcode, the operand, and the number of cycles it takes to execute the following instructions.
- b) Assume each instruction is at memory location \$C000. Start over with PC=\$C000 each time, and hand execute each instruction one at a time. That is, it is not a program that you execute one instruction after another. After each instruction is executed, indicate the registers and/or memory locations that get affected by the instructions. Give the new contents of the relevant registers and/or memory locations.

```

ldy    #$40
ldx    $30
ldy    $2002
ldy    2,X
ldy    2,Y
adca  #$30
adca  $30

```

- 2.17** Hand execute the following 9S12 program. After each instruction is executed, illustrate the condition of the stack and the values of the registers after each of the following instructions. The original condition is RegA=\$55, RegB=\$66, RegX=\$A5B6, and RegY=\$A1B2. Assume the stack is initially empty with the RegSP= \$4000.

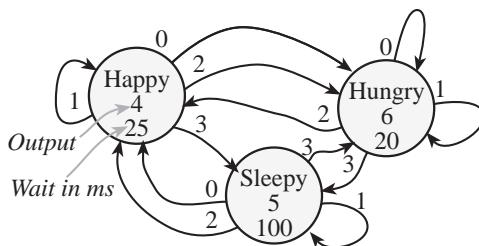
```

psha
pshx
pshy
pulb
pula
pulx
pula

```

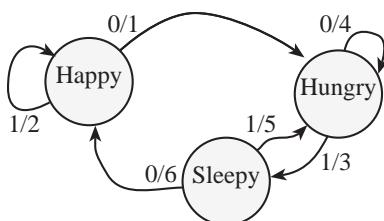
- 2.18** In 32 words or less, describe the meaning of each of the following terms.
  - a) Cohesion
  - b) Bandwidth
  - c) Call graph
  - d) Data flow graph
- 2.19** Give a quantitative measure of modularity (e.g., System A is more modular than system B) if...
- 2.20** In 32 words or less, describe the meaning of each of the following debugging terms.
  - a) Profile
  - b) Intrusive
  - c) Stabilize
  - d) Heartbeat
  - e) Monitor
  - f) Dump
  - g) Logic analyzer
  - h) Filter
- D2.21** Assume RegA contains an ASCII character. Write an assembly subroutine that converts any lowercase letters (a–z) to uppercase (A–Z). For example, if RegA is initially ‘g’, convert it to ‘G’. If the input is not a–z, leave it unchanged.
- D2.22** Assume RegA contains an ASCII character. Write an assembly subroutine that converts any uppercase letters (A–Z) to lowercase (a–z). For example, if RegA is initially ‘G’, convert it to ‘g’. If the input is not A–Z, leave it unchanged.
- D2.23** Write assembly or C software to implement the following Moore FSM (Figure 2.41). Include the FSM state machine, port initialization, timer initialization, and the FSM controller. The command sequence will be output, wait the specified time in ms, input, then branch to next state. The 2-bit input is on Port T (PT1 and PT0), and the 3-bit output is on Port T (PT7, PT6, PT5). Assume the E clock is 8 MHz.

**Figure 2.41**  
FSM for Exercise 2.23.



**D2.24** Write assembly or C software to implement the following Mealy FSM (Figure 2.42). Include the FSM state machine, port initialization, timer initialization, and the FSM controller. The command sequence will be input, output, wait 10 ms, input, then branch to next state. The 1-bit input is on Port P (P00), and the 3-bit output is on Port P (PP3, PP2, PP1). Assume the E clock is 8 MHz.

**Figure 2.42**  
FSM for Exercise 2.24.



**D2.25** Rewrite the Timer\_Wait function in Program 2.10 so that it continuously checks an alarm input on PA7. As long as PA7 is low (normal), it will wait the prescribed time. But, if PA7 goes high (alarm), the wait function returns.

**D2.26** Write a subroutine, called FUZZY, that performs the following input/output function. In and Th are inputs and Out is the result. All parameters are 8-bit unsigned integers. If In is greater than or equal to Th, then Out is 0. If In is less than Th, then Out is  $(255 * (Th - In)) / Th$ . A typical calling shows the two inputs, In and Th, are passed on the stack and the return parameter, Out, is returned in Reg B.

```

ldaa #150      value for Th
psha           Th pushed on the stack
ldaa #90       value for In
psha           In on the stack
jsr  FUZZY     your function
ins
ins           pop off Th and In
* Reg B = (255*(150-90))/150 = 102

```

You must use at least one local variable. COMMENTS will be graded.

## 2.13 Lab Assignments

**Lab 2.1** The overall objective is to create a **4-key digital lock**. The system has four digital inputs and one digital output. The LED will be initially on, signifying the door is locked. Define two separate key codes, one to lock and one to unlock the door. For example, if the keys are numbers 1, 2, 3, and 4, one possible key code is 23. This means if you push both the 2 and 3 keys (not pushing the

1, 4 keys) the door will unlock. Implement the design such that the unlock function occurs in the software of the 6811/6812.

**Lab 2.2** The overall objective is to create a **line tracking robot**. The system has two digital inputs and two digital outputs. You can simulate the system with two switches and two LEDs, or build a robot with two DC motors and two optical reflectance sensors. Both sensor inputs will be on if the machine is completely on the line. One sensor input will be on and the other off if the machine is just going off the track. If the machine is totally off the line, then both sensor inputs will be off. Implement the controller using a finite state machine. Choose a Moore or Mealy format as appropriate.

**Lab 2.3** The overall objective is to create an **enhanced traffic light controller**. The system has three digital inputs and seven digital outputs. You can simulate the system with three switches and seven LEDs. The inputs are North, East, and Walk. The outputs are six for the traffic light and one for a walk signal. Implement the controller using a finite state machine. Choose a Moore or Mealy format as appropriate.

**Lab 2.4** The overall objective is to create an **8-key digital lock**. The system has eight digital inputs and one digital output. The LED will be initially off, signifying the door is locked. Define a key sequence to unlock the door. For example, if the keys are numbers 1, 2, . . . and 8, one possible key code is 556. This means if you push the 5, release the 5, push the 5, release the 5 and push the 6, then the door will unlock. The unlock operation will be a two-second pulse on the LED.

**Lab 2.5** The overall objective is to design a **vending machine controller**. The system has five digital inputs and three digital outputs. You can simulate the system with five switches and three LEDs. The inputs are *quarter*, *dime*, *nickel*, *soda*, and *diet*. The *quarter* input will go high, then go low when a 25¢ coin is added to the machine. The *dime* and *nickel* inputs work in a similar manner for the 10¢ and 5¢ coins. The sodas cost 35¢ each. The user presses the *soda* button to select a regular soda and the *diet* button to select a diet soda. The *GiveSoda* output will release a regular soda if pulsed high, then low. Similarly, the *GiveDiet* output will release a diet soda if pulsed high, then low. The *Change* output will release a 5¢ coin if pulsed high, then low. Implement the controller using a finite state machine. Choose a Moore or Mealy format as appropriate. Because there are so many inputs and at most one is active at a time, you may wish to implement an FSM with a different format from the examples in the book.

**Lab 2.6** This lab is an example of tail recursion. Implement the following recursive greatest common divisor function in assembly language. Convert the operation to a nonrecursive algorithm, and implement it also in assembly language. Pass parameters in registers, and place local variables also in registers. Implement 16-bit unsigned arithmetic. Design a main program to test the functionality of your solution. Measure the execution speed and required stack space of both versions for five different input values. Generalize the results.

```
unsigned short gcd(unsigned short m, unsigned short n){
    unsigned short r;
    if(m < n){
        return gcd(n,m);
    }
    r = m%n;
    if(r == 0){
        return(n);
    }
    return(gcd(n,r));
}
```

**Lab 2.7** This lab is an example of binary recursion.  $nC_k$  is the number of combinations of choosing  $n$  elements out of a set of  $k$  elements. Implement the following recursive function in assembly language. Convert the operation to a nonrecursive algorithm, and implement it also in assembly language. Pass parameters in registers, and place local variables also in registers. Implement 16-bit

unsigned arithmetic. Design a main program to test the functionality of your solution. Measure the execution speed and required stack space of both versions for five different input values. Generalize the results.

```
unsigned short nCk(unsigned short n, unsigned short k){  
    if((k == 0) || (n == k)){  
        return(1);  
    }  
    return(nCk(n-1,k) + nCk(n-1,k-1));  
}
```

# 3 Interfacing Methods

---

## Chapter 3 objectives are to:

- ❖ Introduce basic performance measures for I/O interfacing
  - ❖ Outline various interfacing approaches
  - ❖ Interface simple I/O devices using blind cycle synchronization
  - ❖ Discuss the basic concepts of gadfly synchronization
  - ❖ Describe general approach to I/O interface design
  - ❖ Present the basic hardware/software for parallel port interfaces
  - ❖ Introduce the general concept of a handshake interface, then present many examples
  - ❖ Implement a serial port device driver
- 

**O**ne factor that makes an embedded system different from a regular computer is the special I/O devices we attach to our embedded system. While the entire book addresses the design and analysis of embedded systems, this chapter serves as an introduction to the critical task of I/O interfacing. Interfacing includes both the physical connections of the hardware devices and the software routines that affect information exchange. The chapter begins with performance measures to evaluate the effectiveness of our system (latency, bandwidth, priority). As engineers we are not asked simply to design and build devices, but we also are required to evaluate our products. Latency and bandwidth are two quantitative performance parameters we can measure on our real-time embedded system. Next, the basic approaches to I/O interfacing are presented (blind cycle, busy-wait, interrupts, periodic polling, and DMA). Although a complete understanding of interrupts and DMA won't come until you complete Chapters 4, 6, 7, and 10, the discussion in this chapter will point to situations that require these more powerful interfacing methods. The rest of the chapter presents simple examples to illustrate the blind cycle and gadfly approaches to interfacing.

---

## 3.1 Introduction

### 3.1.1 Performance Measures

*Latency* is the time between when the I/O device needs service and when service is initiated. Latency includes hardware delays in the digital gates plus computer hardware delays. Latency also includes software delays. For an input device, software latency (or software response time) is the time between new input data ready and the software reading the data. For an output device, latency is the delay from output device idle and the software giving the device new data to output. In this book, we will also have periodic events. For example, in our data acquisition systems, we wish to invoke the ADC at a fixed time interval. In this way we can collect a sequence of digital values that approximate the continuous analog signal. Software latency in this case is the time between when the ADC is supposed

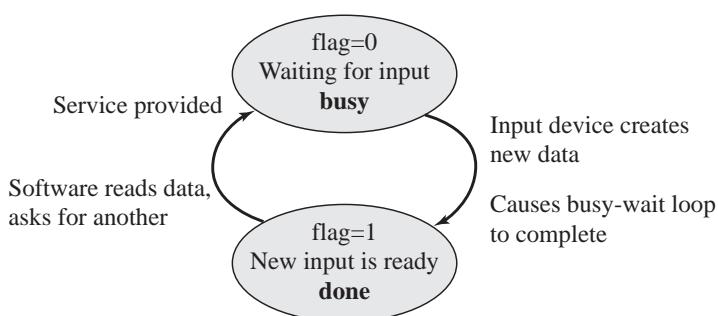
to be started and when it is actually started. The microcomputer-based control system also employs periodic software processing. Similar to the data acquisition system, the latency in a control system is the time between when the control software is supposed to be run and when it is actually run. A *real-time* system is one that can guarantee a worst-case latency. In other words, there is an upper bound on the software response time. *Throughput* or *bandwidth* is the maximum data flow (bytes per second) that can be processed by the system. Sometimes the bandwidth is limited by the I/O device, while other times it is limited by computer software. Bandwidth can be reported as an overall average or a short-term maximum. *Priority* determines the order of service when two or more requests are made simultaneously. Priority also determines if a high-priority request should be allowed to suspend a low-priority request that is currently being processed. We may also wish to implement equal priority so that no one device can monopolize the computer. In some computer literature, the term *soft real time* is used to describe a system that supports priority.

### 3.1.2 Synchronizing the Software with the State of the I/O

One can think of the hardware as being in one of three states. The *idle* state occurs when the device is disabled or inactive. No I/O occurs in the idle state. When active (not idle), the hardware toggles between the *busy* and *done* states. For an input device, a status flag is set when new input data are available (Figure 3.1). The busy-to-done state transition will cause a busy-wait loop (gadfly loop) to complete. Once the software recognizes that the input device has new data, it will read the data and ask the input device to create more data. These hardware state transitions are illustrated in Figures 3.1 and 3.3. It is the *busy-to-done* state transition that signals to the computer that service is required. When the hardware is in the *done* state, the I/O transaction is complete. Often the simple process of reading the data will clear the flag and request another input. Later in this chapter we will present examples of this type of interface.

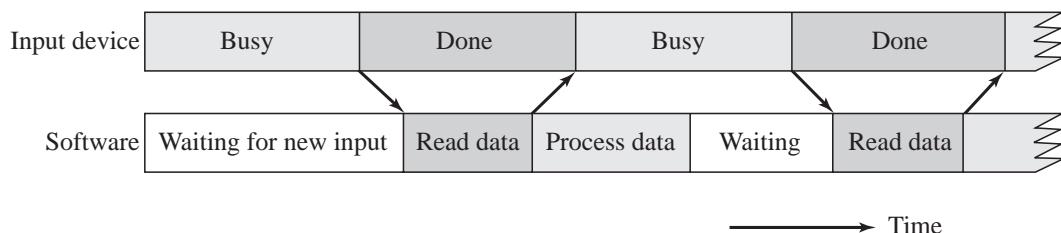
**Figure 3.1**

The input device sets a flag when it has new data.



The problem with I/O devices is that they are usually much slower than software execution. Therefore, we need synchronization, which is the process of the hardware and software waiting for each other in a manner such that data are properly transmitted. A way to visualize this synchronization is to draw a state versus time plot of the activities of the hardware and software (Figure 3.2). For an input device, the software begins by waiting for new input. When the input device is busy, it is in the process of creating new input. When the input device is done, new data are available. When the input device makes the transition from busy to done, it releases the software to go forward. In a similar way, when the software accepts the input, it can release the input device hardware. The arrows from one graph to the other represent the synchronizing events. In this example, the time for the software to read and process the data is less than the time for the input

device to create new input. This situation is called *I/O bound*. If the input device were faster than the software, a situation called *CPU bound*, then the software waiting time would be zero. From Figure 3.2 we can see that the bandwidth depends on both the hardware and the software.



**Figure 3.2**

The software must wait for the input device to be ready.

This configuration is also labeled as *unbuffered* because the hardware and software must wait for each other during the transmission of each piece of data. A buffered system allows the input device to run continuously, filling a buffer as fast as it can. In the same way, the software can empty the buffer whenever it is ready and whenever data are in the buffer. We will implement a buffered interface in Chapter 4 using interrupts.

For an output device, a status flag is set when the output is idle and ready to accept more data (Figure 3.3). The busy-to-done state transition causes a busy-wait loop (gadfly loop) to complete. Once the software recognizes that the output is idle, it gives the output device another piece of data to output. It will be important to make sure the software clears the flag each time new output is started.

**Figure 3.3**

The output device sets a flag when it has finished outputting the last data.

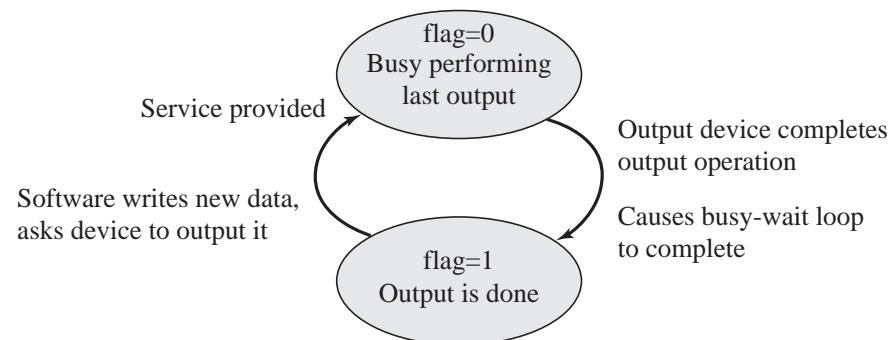
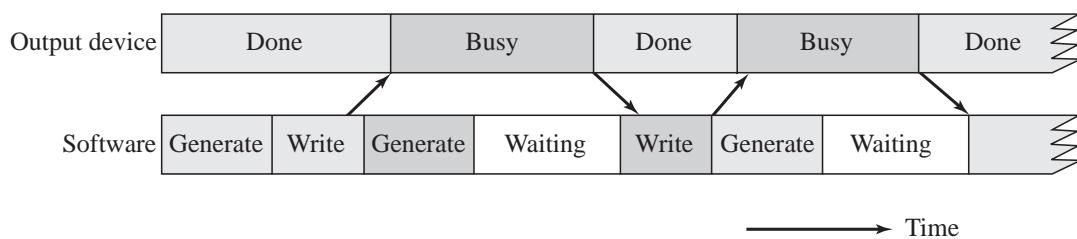


Figure 3.4 contains a state versus time plot of the activities of the output device hardware and software. For an output device, the software begins by generating data, then sending them to the output device. When the output device is busy, it is processing the data. Normally when the software writes data to an output port, that only starts the output process. The time it takes an output device to process data is usually longer than the software execution time. When the output device is done, it is ready for new data. When the output device makes the transition from busy to done, it releases the software to go forward. In a similar way, when the software writes data to the output, it releases the output device hardware. The output interface illustrated in Figure 3.4 is also I/O bound because the time

**Figure 3.4**

The software must wait for the output device to finish the previous operation.

for the output device to process data is longer than the time for the software to generate and write it.

This output interface is also unbuffered, because when the hardware is done, it will wait for the software, and after the software generates data, it waits for the hardware. A buffered system would allow the software to run continuously, filling a buffer as fast as it wishes. In the same way, the hardware can empty the buffer whenever it is ready and whenever there are data in the buffer. We will implement a buffered interface in Chapter 4 using interrupts.

The purpose of our interface is to allow the microprocessor to interact with its external I/O device. There are five mechanisms to synchronize the microprocessor with the I/O device. Each mechanism synchronizes the I/O data transfer to the busy-to-done transition. The methods are discussed in the following paragraphs.

*Blind cycle* is a method whereby the software simply waits a fixed amount of time and assumes the I/O will complete after that fixed delay. For an input device, the software triggers (starts) the external input hardware, waits a specified time, then reads data from device. For an output device, the software writes data to the output device, triggers (starts) the device, then waits a specified time. We call this method *blind* because there is no status information about the I/O device reported to the computer software. This method will be used in situations where the I/O speed is short and predictable.

*Gadfly* or *busy-wait* is a software loop that checks the I/O status waiting for the done state. For an input device, the software waits until the input device has new data, then reads them from the input device. For an output device, the software writes data, triggers the output device, then waits until the device is finished. Another approach to output device interfacing is for the software to wait until the output device has finished the previous output, write data, then trigger the device. We will discuss these two approaches to output device interfacing later in the chapter. Busy-wait synchronization will be used in situations where the software system is relatively simple and real-time response is not important.

An *interrupt* uses hardware to cause special software execution. With an input device, the hardware will request an interrupt when the input device has new data. The software interrupt service will read the data from the input device and save them in a global structure. With an output device, the hardware will request an interrupt when the output device is idle. The software interrupt service will get data from a global structure, then write them to the device. Sometimes we configure the hardware timer to request interrupts on a periodic basis. The software interrupt service will perform a special function. A data acquisition system needs to read the ADC at a regular rate. Details of data acquisition systems can be found in Chapters 11 and 12. The Freescale microcomputers will execute special software when it tries to execute an illegal instruction. Other computers can be configured to request an interrupt on an access to an illegal address or a divide by zero.<sup>1</sup> Interrupt

<sup>1</sup>The Freescale microcomputers do not provide for a divide-by-zero trap, but most computers do.

synchronization will be used in situations where the software system is fairly complicated or when real-time response is important.

*Periodic polling* uses a clock interrupt to periodically check the I/O status. With an input device, a ready flag is set when the input device has new data. At the next periodic interrupt, the software will read the data and save them in a global structure. With an output device, a ready flag is set when the output device is idle. At the next periodic interrupt, the software will get data from a global structure and write them. Periodic polling will be used in situations that require interrupts, but the I/O device does not support interrupt requests.

*Direct memory access* is an interfacing approach that transfers data directly to/from memory. With an input device, the hardware will request a DMA transfer when the input device has new data. Without the software's knowledge or permission, the DMA controller will read from the input device and save data in memory. With an output device, the hardware will request a DMA transfer when the output device is idle. The DMA controller will get data from memory, then write them to the device. Sometimes we configure the hardware timer to request DMA transfers on a periodic basis. DMA can be used to implement a high-speed data acquisition system. Details of DMA can be found in Chapter 10. DMA synchronization will be used in situations where bandwidth and latency are important.

### 3.1.3 Variety of Available I/O Ports

Microcomputers perform digital I/O using their ports. In this chapter, we will focus on the input and output of digital signals. Microcontrollers have a wide variety of configurations, only few of which are illustrated in Table 3.1. Each microcontroller manufacturer has multiple families consisting of a wide range of parts with varying numbers and types of I/O devices. It is typical for port pins to be programmed via software for alternative functions other than parallel I/O. Each microcontroller family comes with a mechanism to download code and debug. The 9S12 family includes a background debug module. The Texas Instruments MSP430 and Stellaris families include either a JTag or Spy-Bi-Wire debugging interface. The Microchip PIC family debugs in a variety of ways, including In-Circuit Serial Programming, In-Circuit Emulator, PICSTART, and PROMATE. The Atmel family can be developed with a JTag debugger. Basically, we first choose the processor type (e.g., PIC, MSP430, 9S12, or LM3S) depending on our software processing needs. Next, we choose the family depending on our I/O requirements. Lastly, we choose the particular part depending on our RAM and ROM memory requirements.

**Table 3.1**

The number of I/O ports and alternative function.

	Port Pins	Alternative Functions
PIC12F629	6	Very low cost, ADC, timer
MSP430F2013	10	Very low power, ADC, SCI, SPI, I2C, and timer
P87C52	32	Serial, timer, industry standard
MC9S12C32	60	Serial, timer, ADC, SPI, CAN
MC9S12NE64	70	Serial, timer, ADC, 10/100 Base-T
AT91RM	Up to 122	Arm Thumb, ADC, serial, DMA, USB, Ethernet, Smart card
LM3S	Up to 72	Arm Cortex M3, ADC, serial, DMA, USB, Ethernet, LCD, CAN

It is good practice to use the same technology of the microprocessor for the design of the I/O interface. When faced with the problem of designing an I/O interface to our microcomputer, we have the choice of using sophisticated devices made with large-scale integrated circuits [e.g., metal-oxide semiconductor (MOS), large-scale integration (LSI), very large scale integration (VLSI)] or simple devices made from small-scale integrated devices like standard TTL (e.g., 7400), low-power Schottky TTL (e.g., 74LS00) and high-speed CMOS (e.g., 74HC00).

The interfacing issues (speed, voltage levels, complexity) are matched when both the processor and I/O are designed from similar technologies. To produce a marketable LSI I/O device, one must:

- Increase the market by making the device flexible, able to perform many functions, and use only a fraction of its power
- Increase flexibility by making the device programmable and by decreasing pins while increasing function
- Increase yield by including redundancy, using a modular design, including on-chip diagnostics, making the device a standard size, and decreasing pins

Table 3.2 clearly shows us that LSI technology is appropriate for designing I/O devices.

**Table 3.2**

Advantages and disadvantages of using LSI technology to design I/O devices.

Advantages	Disadvantages
Shorter design time	Increased software complexity
Increased performance	Need to write a software ritual
Reduced size	Added LSI design costs
Fewer bugs	
Easier to maintain	
Easier to modify	
Increased flexibility	
Lower power	

To be successful each computer family needs a set of high-performance I/O devices. In single-chip microcomputer systems like the 9S12, most of these I/O devices are built-in. Similarly, Cyrix has an integrated microprocessor chip, Gx86, that implements the x86 microprocessor and the associated I/O ports (video controller, sound blaster, peripheral component interconnect (PCI) controller, etc.). We call the Gx86 an integrated microprocessor instead of a single-chip microcomputer because external memory is required to complete the system. Many manufacturers have a line of integrated microprocessors built around the Arm processor. Examples include the LM3S, STM32 and AT91.

In the early days of microcomputer interfacing, before single-chip microcomputers, design engineers would first evaluate the needs of their project. They would select a basic microprocessor (like the 6800, 6809, or 68000) that could handle the software tasks. Then they added external RAM, PROM, and I/O devices to build the microcomputer system. Adding devices is a very expensive and complex process but provides for a wide range of possibilities.

The current trend in the microcomputer industry is customer-specific integrated circuits (CSICs). A similar term for this development process is application-specific integrated circuits (ASICs). With these approaches, the design engineers (customer) first evaluate the needs of their project. In many ways this new development process is similar to the “older way” of design, but now the design engineers work more closely with the microcomputer manufacturer. The design engineers together with the microcomputer manufacturer make a list of features the microcomputer requires. For example:

CPU type	CISC, RISC, multiple cores
Coprocessors	Floating point, DMA, graphics
Memory	RAM, EEPROM, Flash, OTP ROM, ROM
Power	PLL, sleep states, variable supply voltage, reduced output drive
Analog	8 to 16-bit ADC, 8 to 12-bit DAC, analog comparators
Timer	Pulse-width modulation, Input capture, Output compare
Parallel Ports	Key wakeup, pull up, pull down, open collector
Serial	Asynchronous (SCI), synchronous (SPI), peripheral (I2C)
Networks	USB, CAN, Ethernet, wireless

The designer engineers either choose a microcontroller from existing products that meets their needs, or the engineers contract with the manufacturer to produce a microcontroller with the exact specifications for that project. Many manufacturers distribute starter code, reference designs, or white papers showing complete implementations using that particular microcontroller to solve actual problems. The availability of such solutions will be extremely helpful, even if the applications are just remotely similar to your problem.

### 3.1.4 Timing Equations

When interfacing devices, it is important to manage when events occur. Typical events include the rise or fall of control signals, when data pins need to be correct, and when data pins actually contain the proper values. In this book, we will use two mechanisms to describe the timing of events. In this section, we present a formal syntax called *timing equations*, which are algebraic mechanisms to describe time. In the next section, we will present graphical mechanisms called timing diagrams.

When using a timing equation, we need to define a *zero-time reference*. For synchronous systems, which are systems based on a global clock, we can define one edge of the clock as time=0. The 9S12 is a synchronous system based on the E clock. Thus, when describing timing of a memory bus cycle on the 9S12, we will define zero-time at the fall of the E clock, because the fall of E to the next fall of E is one bus cycle. Timing equations can contain numbers typically given in ns, variables, and edges. For example,  $\downarrow A$  means the time when signal A falls, and  $\uparrow A$  means the time when it rises. Some timing variables we see frequently in data sheets include:

$t_{pd}$	propagation delay from a change in input to a change in output
$t_{pHl}$	propagation delay from input to output, as the output goes from high to low
$t_{pLH}$	propagation delay from input to output, as the output goes from low to high
$t_{pZL}$	propagation delay from control to output, as the output goes from floating to low
$t_{pZH}$	propagation delay from control to output, as the output goes from floating to high
$t_{pLZ}$	propagation delay from control to output, as the output goes from low to floating
$t_{pHZ}$	propagation delay from control to output, as the output goes from high to floating
$t_{en}$	propagation delay from floating to driven either high or low, same as $t_{pZL}$ and $t_{pZH}$
$t_{dis}$	propagation delay from driven high/low to floating, same as $t_{pLZ}$ and $t_{pHZ}$
$t_{su}$	setup time, the time before a clock input data must be valid
$t_h$	hold time, the time after a clock input data must continue to be valid

To specify an interval of time, we give its start and stop times between parentheses separated by a comma. For example, (400, 520) means the time interval begins at 400 ns and ends at 520 ns. These two numbers are relative to the zero-time reference. We can use algebraic variables, edges, and expressions to describe complex behaviors. Some timing intervals are not dependent on the zero-time reference. For example, ( $\uparrow A - 10, \uparrow A + t$ ) means the time interval begins 10 ns before the rising edge of signal A and ends at time t after that same rising edge.

Sometimes we are not quite sure exactly when an event starts or stops, but we can give upper and lower bounds. We will use brackets to specify this timing uncertainty. For example, assume we know the interval starts somewhere between 400 and 430 ns, and stops somewhere between 520 and 530 ns, we would then write ([400, 430], [520, 530]).

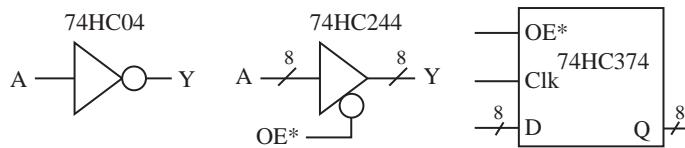
As examples, we will consider the timing of a not gate, a tristate driver and an octal D flip-flop, as shown in Figure 3.5. If the input to the 74HC04 is low, its output will be high. Conversely, if the input to the 74HC04 is high, its output will be low. There are eight

data inputs to the 74HC244, labeled as **A**. Its eight data outputs are labeled **Y**. The 74HC244 tristate driver has two modes. When the output enable, **OE\***, is low, the output **Y** equals the input **A**. When **OE\*** is high, the output **Y** floats, meaning it is not driven high or low. The slash with an 8 above it top means there are eight signals that all operate in a similar or combined fashion. The 74HC374 octal D flip-flop has 8 data inputs (**D**) and eight data outputs (**Q**). A D flip-flop will store or latch its **D** inputs on the rising edge of its **Clk**. The **OE\*** signal on the 74HC374 works in a manner similar to the 74HC244. When **OE\*** is low, the stored values in the flip-flop are available at its **Q** outputs. When **OE\*** is high, the **Q** outputs float. The making **OE\*** go high or low does not change the internal stored values. **OE\*** only affects whether or not the stored values are driven on the **Q** outputs.

*Positive logic* means the true or asserted state is a higher voltage than the false or not asserted state. *Negative logic* means the true or asserted state is a lower voltage than the false or not asserted state. The \* in the name **OE\*** means negative logic. Other writing styles that mean negative logic include a slash before the symbol (e.g.,  $\text{OE}$ ) or a line over the top (e.g.,  $\overline{\text{OE}}$ ).

**Figure 3.5**

A NOT gate, a tristate driver, and an octal D flip-flop.



We will begin with the timing of the 74HC04 not gate. The typical propagation delay time ( $t_{pd}$ ) for this not gate is 8 ns. Considering just the typical delay, we specify time when **Y** rises in terms of the time when **A** falls. That is,

$$\uparrow Y = \downarrow A + t_{pd} = \downarrow A + 8$$

From the 74HC04 data sheet, we see the maximum propagation delay is 15 ns, and no minimum is given. Since the delay cannot be negative, we set the minimum to zero and write

$$\uparrow Y = [\downarrow A, \downarrow A + 15] = \downarrow A + [0, 15]$$

We specify the time interval when **Y** is high as

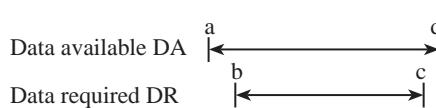
$$(\uparrow Y, \downarrow Y) = ([\downarrow A, \downarrow A + 15], [\uparrow A, \uparrow A + 15]) = (\downarrow A + [0, 15], \uparrow A + [0, 15])$$

When data is transferred from one location (the source) and stored into another (the destination), there are two time intervals that will determine if the transfer will be successful. The *data available* interval specifies when the data driven by the source is valid. The data available interval is when the output will be valid. The *data required* interval specifies when the data to be stored into the destination must be valid. The data required interval is when the input needs to be valid. For a successful transfer, the data available interval must overlap (start before and end after) the data required interval. Let **a**, **b**, **c**, **d** be times relative to the same zero-time reference, let the data available interval be **(a, d)**, and let the data required interval be **(b, c)**, as shown in Figure 3.6. The data will be successfully transferred if

$$a \leq b \text{ and } c \leq d$$

**Figure 3.6**

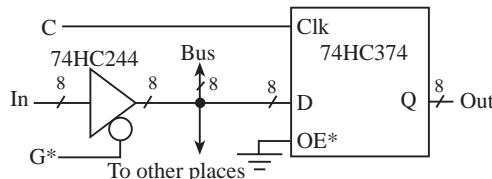
The data available interval should overlap the data required interval.



The example shown in Figure 3.7 illustrates the fundamental concept of timing for a digital interface. The object is to transfer the data from the input, **In**, to the output, **Out**. First, we assume the signal at the **In** input of the 74HC244 is always valid. When the tristate control, **G\***, is low then the **In** is copied to the **Bus**. On the rising edge of **C**, the 74HC374 D flip-flop will copy this data to the output **Out**.

**Figure 3.7**

Simple circuit to illustrate that the data available interval should overlap the data required interval.



The data available interval defines when the signal **Y** contains valid data and is determined by the timing of the 74HC244. From its data sheet, the output of the 74HC244 is valid between 0 and 38 ns after the fall of **G\***. It will remain valid until 0 to 38 ns after the rise of **G\***. The data available interval is

$$DA = (\downarrow G^* + t_{en}, \uparrow G^* + t_{dis}) = (\downarrow G^* + [0, 38], \uparrow G^* + [0, 38])$$

The data required interval is determined by the timing of the 74HC374. The 74HC374 input, **D**, must be valid from 25 ns before the rise of **C** and remain valid until 5 ns after that same rise of **C**. The time before the clock the data must be valid is called the *setup time*. The setup time for the 74HC374 is 25 ns. The time after the clock the data must continue to be valid is called the *hold time*. The hold time for the 74HC374 is 5 ns. The data required interval is

$$DR = (\uparrow C - t_{su}, \uparrow C + t_h) = (\uparrow C - 25, \uparrow C + 5)$$

Since the objective is to make the data available interval overlap the data required window, the worst case situation will be the shortest data available and the longest data required intervals. Without loss of information, we can write the shortest data available interval as

$$DA = (\downarrow G^* + 38, \uparrow G^*)$$

Thus, the data will be properly transferred if the following are true:

$$\downarrow G^* + 38 \leq \uparrow C - 25 \quad \text{and} \quad \uparrow C + 5 \leq \uparrow G^*$$

Notice in Figure 3.7, the signal between the 74HC244 and 74HC374 is labeled **Bus**. A *bus* is a collection of signals that facilitate the transfer of information from one part of the circuit to another. Consider a system with multiple 74HC244's and multiple 74HC374's. The **Y** outputs of all the 74HC244's and the **D** inputs of all the 74HC374's are connected to this bus. If the system wished to transfer from input 6 to output 5, it would clear **G6\*** low, make **C5** rise and then set **G6\*** high. One of the problems with shared bus will be *bus arbitration*, which is a mechanism to handle simultaneous requests. We will see in Chapter 9 that the 9S12 CPU is the bus master, arbitrating all activity on the bus.

### 3.1.5 Timing Diagrams

An alternative mechanism for describing when events occur uses voltage versus time graphs, called *timing diagrams*. It is very intuitive to describe timing events using graphs, because it is easy to visually sort events into their proper time sequence. Figure 3.8 defines the symbols we will use to draw timing diagrams in this book. Arrows will be added to describe the causal relations in our interface. Numbers or variables can be included that define how far apart events will be or should be. It is important to have it clear in our minds whether we are drawing an input or an output signal, because what a

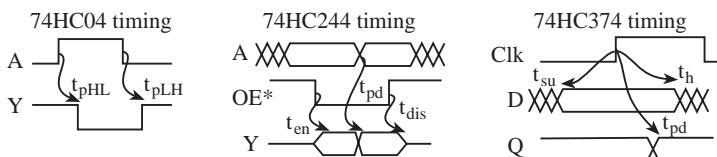
symbol means depends on whether we are drawing the timing of an input or an output signal. Many datasheets use the tristate symbol when drawing an input signal to mean “don’t care.”

**Figure 3.8**  
Nomenclature for drawing timing diagrams.

Symbol	Input	Output
X	The input must be valid	The output will be valid
—	If the input were to fall	Then the output will fall
—	If the input were to rise	Then the output were to rise
XXXXXXXX	Don’t care, will work regardless	Don’t know, output value is indeterminate
Y	Not defined	High impedance, tristate, HiZ, Not driven, floating

To illustrate the graphical relationship of dynamic digital signals, we will draw timing diagrams for the three devices presented in the last section (see Figure 3.9). The arrows in the 74HC04 timing diagram describe the causal behavior. If the input were to rise, then the output will fall  $t_{pHL}$  time later. The subscript HL refers to the output changing from high to low. Similarly, if the input were to fall, then the output will rise  $t_{pLH}$  time later.

**Figure 3.9**  
Timing diagrams for the circuits in Figure 3.5.



The arrows in the 74HC244 timing diagram also describe the causal behavior. If the input A is valid and if the OE\* were to fall, then the output will go from floating to properly driven  $t_{en}$  time later. If the OE\* is low and if the input A were to change, then the output will change  $t_{pd}$  time later. If the OE\* were to rise, then the output will go from driven to floating  $t_{dis}$  time later.

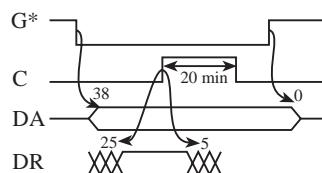
The parallel lines on the D timing of the 74HC374 mean the input must be valid. “Must be valid” means the D input could be high or low, but it must be correct and not changing. In general, arrows represent causal relationships (i.e., “this” causes “that”). Hence, arrows should be drawn pointing to the right towards increasing time. The setup time arrow is an exception to the “arrows point to the right” rule. The setup arrow (labeled with  $t_{su}$ ) defines how long before an edge the input must be stable. The hold arrow (labeled with  $t_h$ ) defines how long after that same edge the input must continue to be stable.

When we get to Chapter 9, we will see the timing of the 74HC244 mimics the behavior of devices on the computer bus during a read cycle, and the timing of the 74HC374 clock mimics the behavior of devices during a write cycle. Figure 3.10 shows the timing diagram for the interface problem presented in Figure 3.7. Again we assume the input **In** is valid at all times. The data available (DA) and data required (DR) intervals refer to data on the **Bus**. In this timing diagram, we see graphically the same design constraint developed with timing equations.  $\downarrow G^* + 38$  must be less than or equal to  $\uparrow C - 25$  and  $\uparrow C + 5$  must be less than or equal to  $\uparrow G^*$ . One of the confusing parts about a timing diagram is that it contains more information than actually matters. For example, notice that the fall of

**C** is drawn before the rise of **G\***. In this interface the relative timing of  $\uparrow G^*$  and  $\downarrow C$  does not matter. However, we draw  $\downarrow C$  so that we can specify the width of the **C** pulse must be at least 20 ns.

**Figure 3.10**

Timing diagram of the interface shown in Figure 3.7.



## 3.2 Key Wake-Up

The basic idea of key wake-up is to connect an input to the 9S12 and configure the interface so an interrupt is requested on either the rising or falling edge of the input. Using key wake-up allows software to respond quickly to changes in the external world. The 9S12C32 has ten possible key wake-up interrupt sources, which are available on Ports J and P. The 9S12DP512 has twenty key wake-up interrupt sources, which are available on Ports H, J, and P. See Table 3.3. Any or all of these pins can be configured as a key wake-up interrupt. Each of the wake-up lines has a separate I/O pin (PTH, PTJ, PTP), a direction register bit (DDRH, DDRJ, DDRP), a trigger flag bit (PIFH, PIFJ, PIFP), an arm bit (PIEH, PIEJ, PIEP), and a polarity bit (PPSH, PPSJ, PPSP). First we identify external digital signals containing strategic edges (rising or falling). In particular, strategic means we wish to execute software whenever one of these edges occur. We connect these digital signals to individual key

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0260	PH7	PH6	PH5	PH4	PH3	PH2	PH1	PH0	PTH
\$0261	PH7	PH6	PH5	PH4	PH3	PH2	PH1	PH0	PTIH
\$0262	DDRH7	DDRH6	DDRH5	DDRH4	DDRH3	DDRH2	DDRH1	DDRH0	DDRH
\$0263	RDRH7	RDRH6	RDRH5	RDRH4	RDRH3	RDRH2	RDRH1	RDRH0	RDRH
\$0264	PERH7	PERH6	PERH5	PERH4	PERH3	PERH2	PERH1	PERH0	PERH
\$0265	PPSH7	PPSH6	PPSH5	PPSH4	PPSH3	PPSH2	PPSH1	PPSH0	PPSH
\$0266	PIEH7	PIEH6	PIEH5	PIEH4	PIEH3	PIEH2	PIEH1	PIEH0	PIEH
\$0267	PIFH7	PIFH6	PIFH5	PIFH4	PIFH3	PIFH2	PIFH1	PIFH0	PIFH
\$0268	PJ7	PJ6	-	-	-	-	PJ1	PJ0	PTJ
\$0269	PJ7	PJ6	-	-	-	-	PJ1	PJ0	PTIJ
\$026A	DDRJ7	DDRJ6	-	-	-	-	DDRJ1	DDRJ0	DDRJ
\$026B	RDRJ7	RDRJ6	-	-	-	-	RDRJ1	RDRJ0	RDRJ
\$026C	PERJ7	PERJ6	-	-	-	-	PERJ1	PERJ0	PERJ
\$026D	PPSJ7	PPSJ6	-	-	-	-	PPSJ1	PPSJ0	PPSJ
\$026E	PIEJ7	PIEJ6	-	-	-	-	PIEJ1	PIEJ0	PIEJ
\$026F	PIFJ7	PIFJ6	-	-	-	-	PIFJ1	PIFJ0	PIFJ
\$0258	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0	PTP
\$0259	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0	PTIP
\$025A	DDRP7	DDRP6	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0	DDRP
\$025B	RDRP7	RDRP6	RDRP5	RDRP4	RDRP3	RDRP2	RDRP1	RDRP0	RDRP
\$025C	PERP7	PERP6	PERP5	PERP4	PERP3	PERP2	PERP1	PERP0	PERP
\$025D	PPSP7	PPSP6	PPSP5	PPSP4	PPSP3	PPSP2	PPSP1	PPSP0	PPSP
\$025E	PIEP7	PIEP6	PIEP5	PIEP4	PIEP3	PIEP2	PIEP1	PIEP0	PIEP
\$025F	PIFP7	PIFP6	PIFP5	PIFP4	PIFP3	PIFP2	PIFP1	PIFP0	PIFP

**Table 3.3**

9S12 key wake-up ports (all twenty pins are available on the 9S12DP512, while just the ten shaded pins are available on the 9S12C32).

wake-up pins. To use key wake-up, we must make these lines an input and configure the strategic edge to be active. Key wake-up interrupts can be configured to be active on either the rising or falling edge. If the corresponding bit in the PPSH/PPSJ/PPSP is 0, then a falling edge will set the trigger flag. Conversely, if the bit in the PPSH/PPSJ/PPSP register is 1, then a rising edge will set the trigger flag. A key wake-up interrupt will be generated if the trigger flag bit is set, the arm bit is set, and the interrupts are enabled ( $I=0$ ). The RDRH/RDRJ/RDRP register determines the drive strength of an output signal. If the bit is 1, then the corresponding output will have one third of the drive current. This decision is a tradeoff between power and speed. Running with reduced current will reduce the power requirements of the system. Running with full current will increase the slew rate of the digital signals, allowing for faster I/O operations.

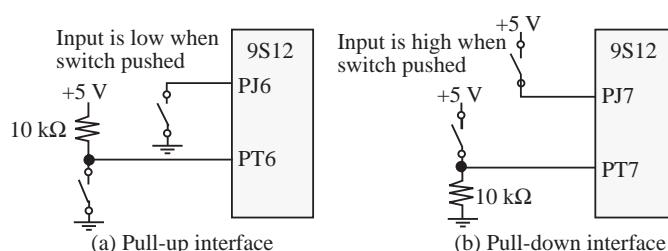
If a pin is configured as an input, then reads to PTH/PTJ/PTP return the same value as reads to PTIH/PTIJ/PTIP, which will be the digital value at the input. Conversely, if a pin is configured as an output, then reads to PTH/PTJ/PTP return the most recent value written to the output port, while reads to PTIH/PTIJ/PTIP will return the digital value at the input. Another convenience of Ports H, J, and P is the available pull-up or pull-down resistors, as shown in Table 3.4. Each of the pins of Ports H, J, and P can be configured separately.

**Table 3.4**  
Pull up/down modes of Port H, J, and P.

DDRH/ DDRJ/DDRP	PPSH/ PPSJ/PPSP	PERH/ PERJ/PERP	Port mode
1	-	-	Regular output
0	0	0	Regular input, falling edge
0	1	0	Regular input, rising edge
0	0	1	Input with passive pull-up, falling edge
0	1	1	Input with passive pull-down, rising edge

A typical application of pull-up is the interface of simple switches. Using pull-up or pull-down mode eliminates the need for an external resistor when interfacing a switch in Figure 3.11. Compare the interfaces on Port J to the interfaces on Port T. The Port P interfaces employ software-configured internal resistors, while the Port T interfaces require actual resistors. The two bit:6 interfaces in Figure 3.11(a) implement negative logic switch inputs, and the two bit:7 interfaces in Figure 3.11(b) implement positive logic switch inputs.

**Figure 3.11**  
Key wake-up or input capture can generate interrupts on a switch touch.



**Checkpoint 3.1:** What do negative logic and positive logic mean in this context?

**Checkpoint 3.2:** What values do you write into DDRJ, PPSJ, and PERJ to configure the switch interfaces of PJ6 and PJ7 in Figure 3.11?

Three conditions must be simultaneously true for a key wake-up interrupt to be requested:

1. The trigger flag bit is set.
2. The arm bit is set.
3. The I bit in the 9S12 CCR is 0.

Even though there are twenty key wake-up lines, there are only three interrupt vectors; one for Port H, one for Port J, and the other for Port P. So, if two or more wake-up interrupts are used on the same port, it will be necessary to poll. Interrupt polling is the software function to look and see which of the potential sources requested the interrupt. The flag bits are cleared by writing a one to it. For example, to clear Port P trigger flag 7 in C we can execute:

```
PIFP = 0x80; // clears flag bit 7 of Port P
```

In assembly, to clear Port P trigger flag 7:

```
movb #$80,PIFP ; clears flag bit 7 of Port P
```

**Checkpoint 3.3:** Which 9S12 pins could we use if we needed to recognize the occurrence a falling edge on an input signal?

**Checkpoint 3.4:** How could you use PTIP and PTP registers to detect whether a Port P output signal is broken or overloaded?

**Checkpoint 3.5:** How do we clear a PIFJ bit 7?

**Checkpoint 3.6:** What bad thing happens if we use this code to clear bit 7  
 $\text{PIFP} |= 0x80;$  ?

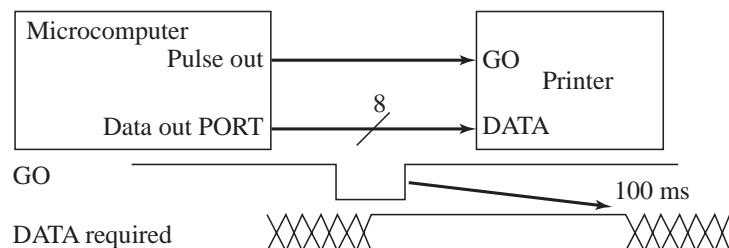
### 3.3 Blind Cycle Counting Synchronization

Blind cycle counting is appropriate when the I/O delay is fixed and known. This type of synchronization is blind because it provides no feedback from the I/O back to the computer.

#### 3.3.1 Blind Cycle Printer Interface

For example, consider a printer that can print 10 characters every second. With blind cycle counting synchronization, there is no printer status signal from the printer telling the computer when the last character output is complete. A simple software interface would be to output the character, then wait 100 ms for it to finish (Figure 3.12).

**Figure 3.12**  
A simple printer  
interface



The subroutine that outputs one character follows these steps: (1) The software places the character to be printed on the output part, (2) the software issues a GO pulse (set GO to high, clear GO to low), and (3) the software waits the 100 ms for the character to be printed (Program 3.1).

The advantage of blind cycle counting is that it is simple and predictable. It does not have the chance of hanging up (i.e., never returning). Unfortunately, there are several

**Program 3.1**

A software function that outputs to a simple printer.

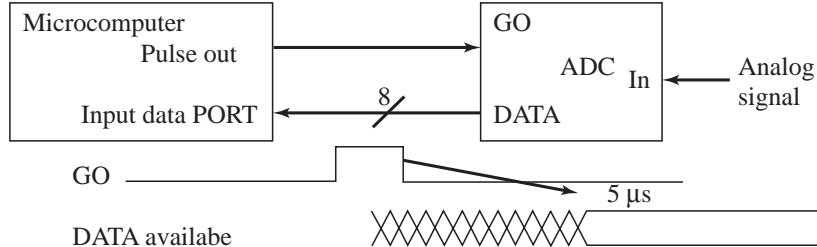
```
void Output (unsigned char LETTER) {
    PORT=LETTER;           /* sets Port outputs */
    Pulse();                /* pulses GO */
    Timer_MsWait(100);      /* Wait for 100 ms */
}
```

disadvantages of the blind cycle counting technique. If the output rate is variable (like a “carriage return,” “tab,” “graphics,” or “formfeed”), then this technique is awkward. If the input rate is unknown (like a keyboard), this technique is inappropriate. The time delay is wasted. If the delay time is long (as it is in the above example), then this technique is dynamically inefficient. This wait time could be used to perform other useful functions. It does not allow for error checking or special conditions.

### 3.3.2 Blind Cycle ADC Interface

Nevertheless, blind cycle counting can be appropriate for simple high-speed interfaces. An ADC has an analog input (e.g.,  $0 \leq \text{In} \leq +5\text{V}$ ) and converts this analog signal into digital form (e.g.,  $0 \leq \text{data} \leq 255$ ). Consider the following example of a high-speed 8-bit ADC interface (Figure 3.13). A positive logic pulse, GO, starts the ADC conversion. The result, DATA, is available 5  $\mu\text{s}$  later. There are no error conditions to consider in this problem.

**Figure 3.13**  
A simple ADC interface.



To perform one ADC conversion the subroutine does the following steps (Program 3.2). First, the software starts the ADC conversion by sending a `GO` pulse (set `GO` to high, clear `GO` to low). Second, the software waits for the ADC to convert the analog input signal into digital form (only takes 5  $\mu\text{s}$ ). Last, the software inputs the 8-bit result. It is a “blind” interface because there is no ADC status signal telling the software when the conversion is complete.

**Program 3.2**

A software function that inputs from an ADC.

```
unsigned char Input(void) {
    Pulse();                  /* pulses GO          */
    Timer_Wait(5);            /* Wait for 5us      */
    return(PORT);              /* Read ADC result */
}
```

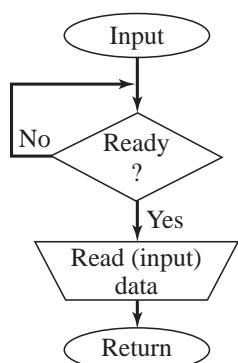
## 3.4 Gadfly or Busy-Wait Synchronization

To synchronize the software with the I/O device, the microcomputer usually must be able to recognize the busy-to-done transition. With *gadfly* or *busy-wait* synchronization, the software checks a status bit in the I/O device and loops back until the device is ready. The busy-wait loop must precede the data transfer for an input device (Figure 3.14).

Two steps are involved when the software interfaces with hardware to perform an output function. One step is for the software to output the new data to an output port. This step usually executes in a short amount of time because it involves just a few instructions, with no backward jumps. The other software step is a busy-wait loop that executes until the

**Figure 3.14**

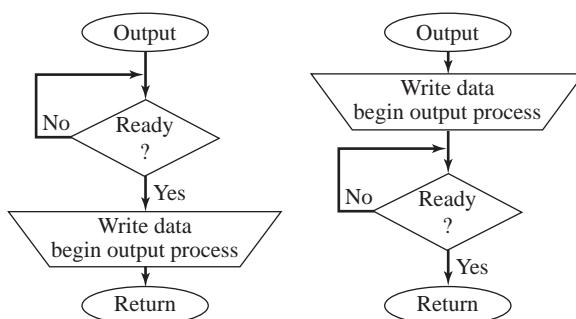
A software flowchart for gadfly input.



output device is ready. The time in this step is usually long compared to the other operations (I/O-bound situation). These two steps can be performed in either order as long as that order is consistently maintained, and we assume the device is initially ready. Polling before the output allows the computer to perform additional tasks while the output is occurring. Therefore, polling before the output will have a higher bandwidth than polling after the output. On the other hand, polling after the output allows the computer to know exactly when the output has been completed (Figure 3.15).

**Figure 3.15**

Two software flowcharts for busy-wait output.

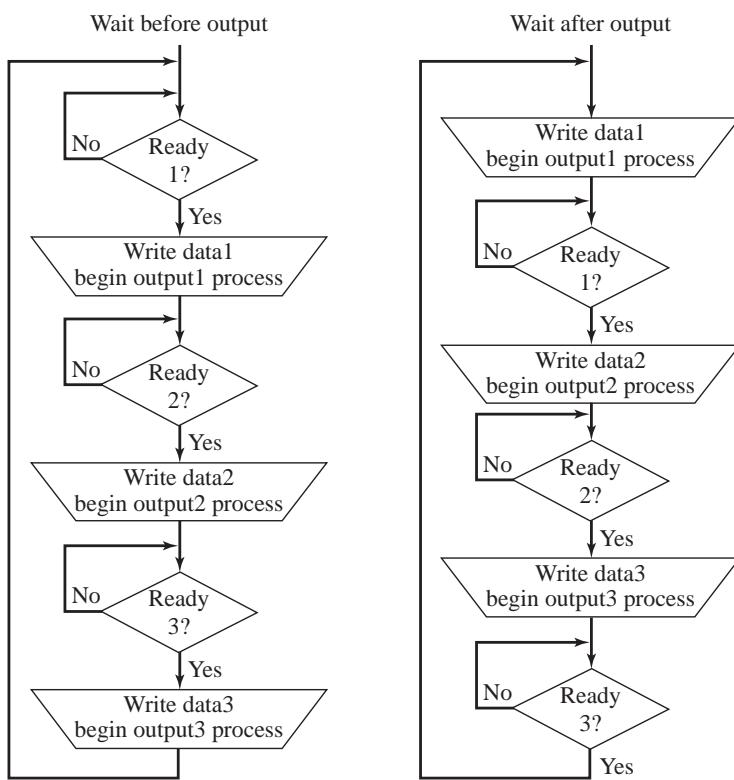


To illustrate the differences between polling before and after the write-data operation, consider a system with three printers. Each printer can print a character in 1 ms. In other words, a printer will be ready 1 ms after the write-data operation. We will also assume all three printers are initially ready. Since the execution speed of the microcomputer is fast compared to the 1 ms it takes to print a character, we will neglect the software execution time (I/O bound). In the gadfly-before-output system, all three outputs are started together and will operate concurrently. In the gadfly-after-output system, the software waits for the output on printer 1 to finish before starting the output on printer 2. In this system, the three outputs are performed sequentially—that is, about three times slower than the first case (Figure 3.16).

Time(ms)	Wait before output	Wait after output
0	Start 1,2,3	Start 1
From 0 to 1	Wait for 1	Wait for 1
1	Start 1,2,3	Start 2
From 1 to 2	Wait for 1	Wait for 2
2	Start 1,2,3	Start 3
From 2 to 3	Wait for 1	Wait for 3
3	Start 1,2,3	Start 1
From 3 to 4	Wait for 1	Wait for 1
4	Start 1,2,3	Start 2
From 4 to 5	Wait for 1	Wait for 2

**Figure 3.16**

Two software flowcharts for multiple busy-wait outputs.

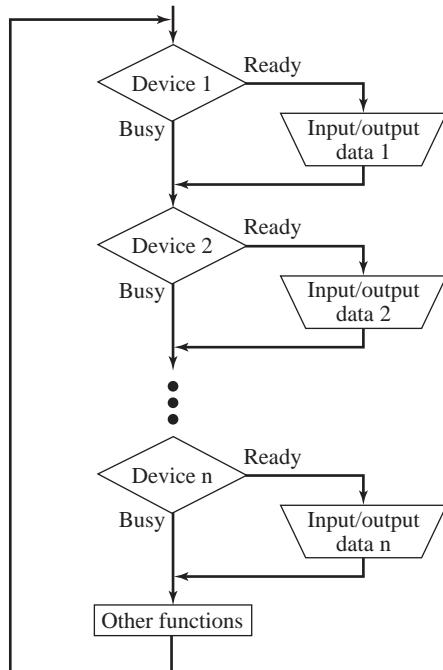


**Performance tip:** Whenever we can establish concurrent I/O operations, we can expect an improvement in the overall system bandwidth.

To implement busy-wait synchronization with multiple I/O devices, simply poll them in sequence and perform service as required. Figure 3.17 implements a fixed priority and

**Figure 3.17**

A software flowchart for multiple busy-wait inputs and outputs.



does not allow high-priority devices to suspend the service of lower-priority devices. Therefore the software response time (latency) to high-priority devices will be poor.

The term gadfly was chosen as an alternate to busy-wait, because it has a negative context outside the computer field. Plato in his book *Apology* describes Socrates as a gadfly, meaning he is a constant and annoying pest to the Athenian political scene. In biology, a gadfly is a large bug in the horse-fly family. A social gadfly is an irritating person who constantly upsets the status quo. As we will learn in this book, interrupt synchronization will provide more elegant and effective solutions over gadfly synchronization when considering issues such as real-time, priority, low-power, and complexity.

## 3.5 Parallel I/O Interface Examples

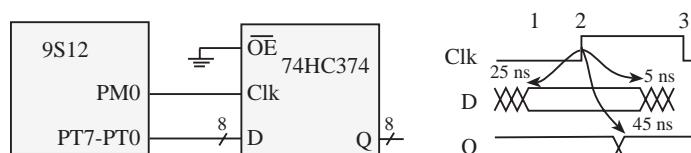
A parallel interface encodes the information as separate binary bits on individual port pins, and the information is available simultaneously. For example, if we are transmitting ASCII characters, then the ASCII code is represented by digital voltages on seven (or eight) digital signals. We begin with parallel I/O devices because they are simple to understand.

**Example 3.1** Design an interface to provide additional output pins. For each additional eight output pins, one 74HC374 octal D flip-flop will be used.

**Solution** A port expander is used to create additional I/O pins. In this example, we will create output pins. Although we will use nine actual pins to create eight outputs, the process can be expanded to provide 16, 24, 32, 40, 48, 56, or 64 outputs using 10, 11, 12, 13, 14, 15, or 16, pins respectively. The right side of Figure 3.18 shows the timing of the D flip-flop. The first step is to put new data on the **D** lines. This is done by outputting the data to **PTT**. The second step is to make a rising edge on the clock **Clk**. This is done by making **PM0** go from a low to a high. The *setup time* is the minimum time between steps 1 and 2. Since the setup time for the 74HC374 is only 25 ns, the interface will work as long as steps 1 and 2 are performed by separate instructions. 45 ns after the rising edge of **Clk**, the new data is available on **Q**. The **Q** data output remains valid until the next rising edge of **Clk**. The *hold time* is the time after the clock the **D** input must remain valid. The hold time for the 74HC374 is 5 ns. The third step will be the falling edge of **Clk**. Nothing special happens on the fall, but it must obviously fall in order to have a subsequent rising edge. In summary, the pulse output on **Clk** will latch **D** input into the **Q** output. When the **OE\*** pin is low, the **Q** outputs are driven high or low with the value previously stored. If the **OE\*** pin were to go high, the **Q** output signals would float (i.e., not driven high or driven low). The **OE\*** pin does not affect the internal stored value, just whether or not the stored value is driven to the **Q** output.

**Figure 3.18**

Hardware interface between the microcomputer and a D flip-flop.



The initialization function configures the pins as outputs, as shown in Program 3.3. The output function stores the parameter into 74HC374 by executing steps 1, 2, and 3. The assembly language function passes the data call-by-value using Register B.

The **bset** and **bclr** instructions require four cycles to execute. If the E clock is 24 MHz, then there will be 167 ns between when the **staa** sets **D** input and when **bset** makes a rising edge of **Clk**. Furthermore, the width of the **Clk** pulse also will be 167 ns. This design illustrates a blind synchronization, because the software blindly outputs, without feedback concerning the status of the external device.

**Program 3.3**

Software to initialize and output.

<pre>; PortT is D, PM0=Clk Init movb #\$FF,DDRT ;outputs       bclr PTM,#\$01 ;Clk=0       bset DDRM,#\$01 ;output       rts ; Data in Register B Out  stab PTT ;1) set data       bset PTM,#\$01 ;2) Clk=1       bclr PTM,#\$01 ;3) Clk=0       rts</pre>	<pre>void Init(void){     DDRT = 0xFF; // outputs     PTM &amp;= ~0x01; // Clk=1     DDRM  = 0x01; // output } void Out(unsigned char value){     PTT = value; // 1) set data     PTM  = 0x01; // 2) Clk=1     PTM &amp;= ~0x01; // 3) Clk=0 }</pre>
--	--

**Checkpoint 3.7:** In Example 3.1, how wide would the **Clk** pulse be if the E clock were changed to 4 MHz? Would the interface still work?

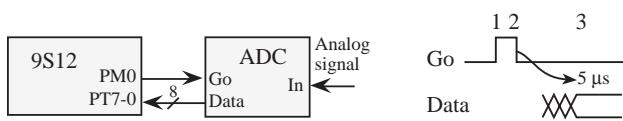
**Performance tip:** It is important to document the relationship between software timing delays and the E clock, so that if the E clock rate were to change, you could anticipate the effects on the performance of the interface.

**Example 3.2** Design an interface to a high-speed ADC. A positive logic pulse, **Go**, starts the analog-to-digital conversion. The 8-bit result, **Data**, is available in parallel form 5  $\mu$ s later.

**Solution** The interface is “blind” because there is no feedback from the ADC back to the software telling the computer whether the ADC is busy or done. The right side of Figure 3.19 shows the ADC timing. To perform an ADC conversion, the software issues a **Go** pulse, waits 5  $\mu$ s, then reads the 8-bit **Data**.

**Figure 3.19**

Hardware interface between an ADC converter and the microcomputer.



The **Data** pins are outputs of the ADC, so the initialization sets the **Data** pins as inputs to the microcontroller, as shown in Program 3.4. Conversely, the **Go** pin is an input to the ADC. Therefore, the **Go** pin is configured as an output of the microcontroller. The subroutine, **In**, triggers an ADC conversion and returns the digital result. The first step is to set **Go** to 1, and the second step will set **Go** to 0. This width of the pulse will be 4 bus cycles. The blind interface waits for ADC to convert the analog input signal into digital form by simply waiting 5  $\mu$ s. The third step is to input the 8-bit ADC result.

**Program 3.4**

Software to initialize and read from an ADC.

<pre>Init  clr DDRT ;data input       bclr PTM,#\$01 ;Go=0       bset DDRM,#\$01       bsr Timer_Init ;Prog 2.6       rts ;return ADC result in Reg B In   bset PTM,#\$01 ;1) Go=1       bclr PTM,#\$01 ;2) Go=0       ldd #5       bsr Timer_Wait ;5us       ldab PTT ;3) read       rts</pre>	<pre>void Init(void){     DDRT = 0x00; // input DATA     PTM &amp;= ~0x01; // Go=0     DDRM  = 0x01; // PM0 Go     Timer_Init(); // Program 2.6 } unsigned char In(void){     PTM  = 0x01; // 1) Go=1     PTM &amp;= ~0x01; // 2) Go=0     Timer_Wait(5); // 5us     return(PTT); // 3) read }</pre>
---	--

**Observation:** When interfacing devices to the microcontroller, the device output signals are connected to microcontroller pins configured as inputs, and the device input signals are connected to microcontroller pins configured as outputs.

**Observation:** When there is just one input parameter, the Metrowerks and ICC12 compilers pass it in RegD or RegB depending on whether it is 16 bits or 8 bits respectively.

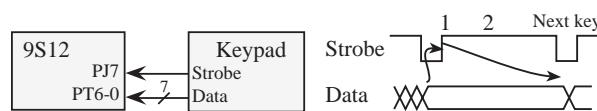
**Observation:** If there is an output parameter, the Metrowerks and ICC12 compilers return it in RegD or RegB depending on whether it is 16 bits or 8 bits respectively.

**Example 3.3.** Design an interface to a parallel keypad. When the operator types a key, the ASCII data becomes available and there is rising edge on a **Strobe** signal.

**Solution** The right side of Figure 3.20 shows the timing for this interface. When the user types a key on this keypad, the 7-bit ASCII code becomes available on the **Data**, followed by a rise in the signal **Strobe**. The data remains available until the next key is typed. The software interface is a two-step sequence. First the software waits for the rising edge using busy-wait synchronization, and then the software reads the **Data**.

**Figure 3.20**

Hardware interface between a simple keyboard and the microcomputer.



The input data are not buffered in hardware for this interface. Therefore, this system requires a real-time solution. The latency for this interface is the time between when the operator types a key (signified by the rising edge of **Strobe**) and the time the software reads the **Data**. For example, if the operator is typing at 10 characters per second, the software must read the **Data** within 100 ms of the rising edge of **Strobe**, or else data will be lost. In particular, the interface latency must be less than 100 ms. We will later show how to interface devices using interrupts, so the latency requirement can be guaranteed. However, in this solution, we will employ busy-wait synchronization, and present no evidence one way or another whether this solution is in real time.

The solution uses the key wake-up feature, as shown in Program 3.5. The **PPSJ** register determines if the rise (1) or fall (0) edge on the Port J inputs will set the corresponding bit in the **PIFJ** register. The initialization routine sets **PPSJ** bit 7 to 1, signifying the rise of **Strobe** will set bit 7 in the **PIFJ** register. The input routine first waits for **PIFJ** bit 7 to be set, and then clears the **PIFJ** bit 7 (writing a 1 to this register clears the bit). This interface

```
; PT6-0 is Data, PJ7=Strobe
Init movb #$80,DDRT ;PT7 unused
bcclr DDRJ,#$80
bset PPSJ,$#80 ;rise on PJ7
movb #$80,PIFJ ;clear flag7
clr PTT ;PT7=0
rts
;Return ASCII key in Reg B
In brclr PIFJ,$#80,In ;1)wait
movb #$80,PIFJ ;clear flag7
ldab PTT ;2) read Data
rts
```

```
void Init(void){ // PJ7= Strobe
DDRJ &= ~0x80; // PT6-0 Data
DDRT = 0x80; // PT7 unused
PPSJ |= 0x80; // rise on PJ7
PIFJ= 0x80; // clear flag7
}
unsigned char In(void){
while((PIFJ&0x80)==0); // 1)wait
PIFJ = 0x80; // clear flag7
return(PTT); // 2) read Data
}
```

### Program 3.5

Assembly language routines to initialize and read from a keypad.

is not blind, because the busy/done status of the keyboard is available to the microcontroller. The advantage of key wake-up is the operator can first type a key, which sets `PIFJ` bit 7, and then the software could call `In`. In this scenario, the data would be properly captured (as long as the software called the input function before the user typed a second character.)

Without using key wake-up, the input function would be three steps. First, it would wait for **Strobe** to be 0. Next, it would wait for **Strobe** to be one. Lastly, it would read the **Data**. Without key wakeup, data would be lost if the user types a key, then the software calls `In`.

**Common error:** CMOS inputs have a very large input impedance and a very small input current. If a CMOS input port is left unconnected, then it may begin to toggle, causing a significant power drain.

**Observation:** A gadfly loop like the one implemented in Programs 3.5 will hang (crash) the computer if the input device is broken and no strobe pulse ever comes.

**Performance tip:** Unused CMOS inputs should be connected to +5 or to ground to prevent power loss.

**Performance tip:** Unused CMOS port pins could be programmed as outputs to prevent power loss.

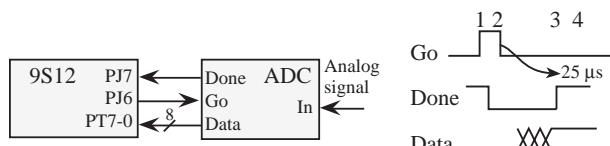
**Observation:** Many pins on the 9S12 can be configured with pull-ups to +5 or pull-downs to ground.

When interfacing a device to a microcontroller, a handshaked interface provides a mechanism for the device to wait for the microcontroller to be ready and for the microcontroller to wait for the device to be ready.

**Example 3.4** Design a handshaked interface to a parallel ADC. A positive logic pulse, **Go**, starts the analog-to-digital conversion. When the ADC conversion is complete, there is a rising edge on **Done**. The 8-bit result, **Data**, then can be read in parallel form.

**Solution:** The right side of Figure 3.21 shows the timing of this handshaked interface. To perform an ADC conversion, the software issues a **Go** pulse. The ADC **Go** pulse is generated first by writing a 1 then writing a 0 to PJ6. The third step is to wait for the ADC to finish, using busy-wait synchronization. The rising edge of **Done** signifies the ADC conversion is complete. The last step is to read the **Data** from PTT. **Done** will remain high, and the **Data** will remain valid until the next ADC conversion operation. Notice that in this handshaked interface, the ADC will wait for the microcontroller to be ready. That is, the ADC does nothing until it receives a **Go** pulse. Furthermore, the microcontroller will wait for the ADC. That is, the software will wait for the rising edge of **Done** before reading the data.

**Figure 3.21**  
Hardware interface  
between an ADC con-  
verter and the micro-  
computer.



The solution, shown in Program 3.6, uses the key wake-up feature. The initialization routine sets `PPSJ` bit 7 to 1, signifying the rise of **Done** will set bit 7 in the `PIFJ` register. The input routine first clears the `PIFJ` bit 7 (writing a 1 to this register clears the bit), issues the pulse on **Go**, waits for `PIFJ` bit 7 to be set, and then reads the **Data**.

**Program 3.6**

Software routines to initialize and read from an ADC. The numbers in the comments refer to Figure 3.21.

```
; PTT=Data, PJ7=Done, PJ6=Go
Init bclr PTJ,#$40 ;Go=0
    bset DDRJ,#$40 ;PJ6 output
    bclr DDRJ,#$80 ;PJ7 input
    bset PPSJ,#$80 ;rise on PJ7
    clr DDRT      ;PT7-0 inputs
    rts
;Return result in Reg B
In   movb #$80,PIFJ ;clear flag7
    bset PTJ,#$40 ;1) Go=1
    bclr PTJ,#$40 ;2) Go=0
loop brclr PIFJ,#$02,loop ;3)wait
    lddab PTT       ;4) Data
    rts

void Init(void){
    PTJ &= ~0x40; // Go=0
    DDRJ |= 0x40; // PJ6=Go out
    DDRJ &= ~0x80; // PJ7=Done in
    PPSJ |= 0x80; // rise on PJ7
    DDRT = 0x00; // PT7-0 DATA in
}
unsigned char In(void){
    PIFJ = 0x80; // clear flag7
    PTJ |= 0x40; // 1) Go=1
    PTJ &=~0x40; // 2) Go=0
    while((PIFJ&0x80)==0); // 3) wait
    return(PTT); // 4) read Data
}
```

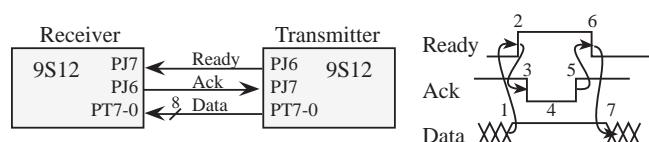
**Observation:** Handshaking is a robust interface. The handshaked interface in Example 3.4 would function properly if either the microcontroller or the ADC were changed to run faster or slower.

**Performance tip:** When initializing output pins, it is better to first write the desired initial output value, then set the direction register to output. This way the pin goes from input to output of the correct value, rather than from input to output of the wrong value, and then output with the correct value.

**Example 3.5** Design a one-directional communication channel between two microcontrollers using a handshaked protocol.

**Solution** Recall that handshaking allows each device to wait for the other. This is particularly important when communicating between two computers. The right side of Figure 3.22 shows the timing as one byte is transferred from transmitter to receiver. The key to designing a handshaked protocol is to require each side to wait until the other side is ready. The transmitter will begin communication by putting new data on the **Data** lines and issuing a rising edge on **Ready**. Furthermore, notice the data is properly driven whenever **Ready** is high. Next, the receiver signals it is about to read the data by making its acknowledge, called **Ack**, low. The receiver then reads the result while **Ack** is low. Finally, the receiver acknowledges acceptance of the data by outputting a rising edge on **Ack**, signaling it is ready to accept another. The handshake ends with a falling edge on **Ready**.

**Figure 3.22**  
Handshaked interface  
between two  
microcontrollers.



This interface is called handshaked or interlocked because each event (1 through 7) follows in sequence one event after the other. The arrows in Figure 3.22 represent causal events. In a handshaked interface, one event causes the next to occur, and the arrows create a “head to tail” sequence. There is no specific minimum or maximum time delay for these causal events, except they must occur in sequence.

1. Transmitter outputs new **Data**.
2. Transmitter makes a rising edge of **Ready**, signifying new **Data** available.

3. Receiver makes a falling edge of **Ack**, signifying the receiver is starting to process the data.
4. The receiver reads the **Data**.
5. Receiver makes a rising edge on **Ack**, signifying the receiver has captured the **Data**.
6. Transmitter makes a falling edge on **Ready**, meaning **Data** is not valid.
7. Transmitter no longer needs to maintain **Data** on its outputs.

One of the issues involved in handshaked interfaces is whether the receiver should wait for both Steps 2 and 6 or just Step 2. Similarly, should the transmitter wait for both Steps 3 and 5? It is a more robust design to affect all four of these waits. If the receiver did not wait for Step 6 (**Ready** = 0), then a subsequent call to **In** may find **Ready** still high from the last call and return the same **Data** a second time. We could have employed key wake-up, but since the interface is interlocked, there is no need to capture the edges. Key wake-up would be required if interrupt synchronization were desired. Program 3.7 shows the assembly solution using busy-wait synchronization.

### Program 3.7

Handshaking assembly language routines to initialize and transfer data. The numbers in the comments refer to Figure 3.22.

<pre>; Transmitter Init bclr PTJ,#\$40 ;Ready=0         bset DDRJ,#\$40 ;PJ6 is Ready out         bclr DDRJ,#\$80 ;PJ7 is Ack in         movb #\$FF,DDRT ;PTT Data out         rts  ;Reg B has data to send Out  stab PTT      ;1)Data out         bset PTJ,#\$40   ;2)Ready=1 out3 brset PTJ,#\$80,out3 ;wait for 3) out5 brclr PTJ,#\$80,out5 ;wait for 5)         bclr PTJ,#\$40   ;6)Ready=0         rts</pre>	<pre>; Receiver Init bset PTJ,#\$40    ;Ack=1         bset DDRJ,#\$40  ;PJ6 is Ack out         bclr DDRJ,#\$80  ;PJ7 is Ready in         clr  DDRT       ;PTT Data in         rts  ;Reg B returned with data In   brclr PTJ,#\$80,In ;wait for 2)         bclr PTJ,#\$40   ;3)Ack=0         ldaa PTT        ;4)read Data         bset PTJ,#\$40   ;5)Ack=1 in6 brset PTJ,#\$80,in6 ;wait for 6)         rts</pre>
--	---

These same algorithms can be implemented in C, as shown in Program 3.8. Assuming both initializations have run, the transfer of data will occur properly regardless of which computer executes its function first.

**Checkpoint 3.8:** Assume both computers in Program 3.8 have executed the initialization. Assume **Out** is called before **In**. List the time-sequenced execution in both computers. Repeat for the case that **In** is called before **Out**.

### Program 3.8

Handshaking C language routines to initialize and transfer data. The numbers in the comments refer to Figure 3.22.

<pre>// Transmitter void Init(void){     PTJ &amp;= ~0x40; // Ready=0     DDRJ  = 0x40; // PJ6=Ready out     DDRJ &amp;= ~0x80; // PJ7 Ack in     DDRT = 0xFF; // PTT Data out }  void Out(unsigned char data){     PTT = data; // 1)Data out     PTJ  = 0x40; // 2)Ready=1     while(PTJ&amp;0x80); // wait 3)     while((PTJ&amp;0x80)==0); // wait 5)     PTJ &amp;= ~0x40; // 6)Ready=0 }</pre>	<pre>// Receiver void Init(void){     PTJ  = 0x40; // Ack=1     DDRJ  = 0x40; // PJ6=Ack out     DDRJ &amp;= ~0x80; // PJ7 Ready in     DDRT = 0x00; // PTT Data in }  unsigned char In(void){     unsigned char data;     while((PTJ&amp;0x80)==0); // wait 2)     PTJ &amp;= ~0x40; // 3)Ack=0     data = PTT; // 4)read data     PTJ  =0x40; // 5)Ack=1     while(PTJ&amp;0x80); // wait 6)     return(data);}</pre>
---	---

**Observation:** Programs written for embedded computers are tightly coupled (depend highly) on the hardware; therefore, it is good programming practice to document the hardware configuration in the software comments.

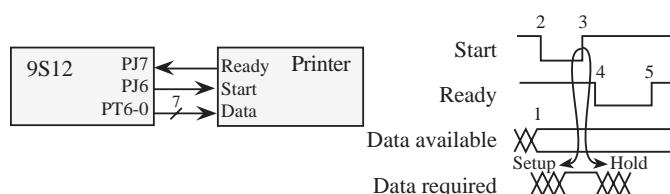
Handshaking is a very reliable synchronization method when connecting devices from different manufacturers and at different speeds. It also allows you to upgrade one device (e.g., get a newer and faster sensor) without redesigning both sides of the interface. Handshaking is used for the SCSI and the IEEE488 instrumentation bus.

**Example 3.6:** Interface a parallel printer using a handshaked protocol.

**Solution** The right side of Figure 3.23 shows the timing of this interface. To output a character on this printer, the software first outputs the 7-bit ASCII code to the **Data**, shown as Step 1 in Figure 3.23. Next the software followed by a negative logic pulse on the signal **Start**, shown as Steps 2 and 3. The printer will drop its **Ready** line signifying the print operation is in progress (Step 4). The completion of the output operation is signified by the rise of **Ready** (Step 5). The printer uses the rise of **Start** to latch the data. The time between Step 1 and Step 3 must be longer than the setup time of the printer. A typical setup time is less than 100 ns, so as long as Steps 1, 2, and 3 are separate instructions, this solution should work. In this dedicated interface, the hold time should not be a problem because the data will be available until the next output. The key wake-up feature to wait for the rise of **Ready**.

**Figure 3.23**

Handshaking hardware interface between a printer and the microcomputer.



The output routine in Program 3.9 follows the sequence: (1) output new **Data**, (2) set **Start** = 0, (3) set **START**=1, then (5) wait for the rising edge of **Ready**. Since key wake-up triggers on the 0 to 1 edge of PJ7, waiting for **PIFJ** bit 7 actually waits for both Step 4 and Step 5. In this example, we will use all eight bits of **PTT** as outputs, even though only seven are required. The assembly **out** functions uses call-by-value parameter passing in Register B.

### Program 3.9

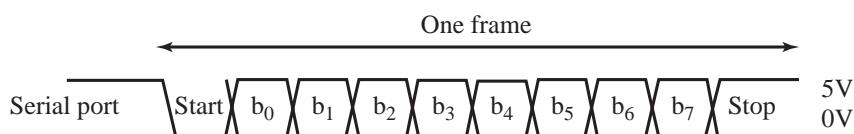
Handshaking assembly language routines to initialize and write to a printer. The numbers in the comments refer to Figure 3.23.

<pre> Init bset PTJ,#\$40      ;Start=1         bset DDRJ,#\$40    ;PJ6 Start out         bclr DDRJ,#\$80    ;PJ7 Ready in         bset PPSJ,#\$80    ;rise on PJ7         movb #\$FF,DDRT   ;PT7-0 outputs         rts         ;RegB is data to output Out   movb #\$80,PIFJ   ;clear flag7         stab PTT          ;1)write Data         bclr PTJ,#\$40    ;2)Start=0         bset PTJ,#\$40    ;3)Start=1         wait brclr PIFJ,#\$80,wait ;wait 5)         rts </pre>	<pre> void Init(void){     PTJ  = 0x40; // Start=1     DDRJ  = 0x40; // PJ6 Start out     DDRJ &amp;= ~0x80; // PJ7 Ready in     PPSJ  = 0x80; // rise on PJ7     DDRT = 0xFF; // PT7-0 Data out } void Out(unsigned char data){     PIFJ = 0x80; // clear flag     PTT = data; // write data     PTJ &amp;=~0x40; // Start =0     PTJ  = 0x40; // Start =1     while((PIFJ&amp;0x80)==0);} </pre>
---	--

## 3.6 Serial Communications Interface (SCI) Device Driver

In this section, we will develop a simple device driver using the Serial Communications Interface (SCI). This serial port allows the microcomputer to communicate with devices such as other computers, printers, input sensors, and LCD displays. Serial transmission involves sending one bit of a time, where the data is spread out over time. The total number of bits transmitted per second is called the *baud rate*. Most of the Freescale embedded microcomputers support at least one *Serial Communications Interface* or SCI. Before discussing the detailed operation of particular devices, we will begin with general features common to all devices. Each SCI module has a *baud rate control register*, which we use to select the transmission rate. There is a mode bit, **M**, which selects 8-bit ( $M=0$ ) or 9-bit ( $M=1$ ) data frames. Each device is capable of creating its own serial port clock with a period that is an integer multiple of the E clock period. The programmer will select the baud rate by specifying the integer divide-by used to convert the E clock into the serial port clock. A *frame* is the smallest complete unit of serial transmission. Figure 3.24 plots the signal versus time on a serial port, showing a single frame, which includes a start bit (0), 8 bits of data (least significant bit first), and a stop bit (1). This protocol is used for both transmitting and receiving. The information rate, or *bandwidth*, is defined as the amount of data or useful information transmitted per second. From Figure 3.24, we see that 10 bits are sent for every byte of usual data. Therefore, the bandwidth of the serial channel (in bytes/second) is the baud rate (in bits/sec) divided by 10.

**Figure 3.24**  
A serial data frame with  
 $M=0$ .



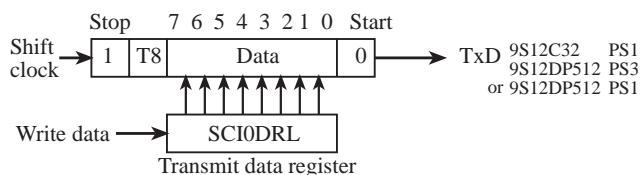
**Common error:** If you change the E clock frequency without changing the baud rate register, the SCI will operate at an incorrect baud rate.

**Checkpoint 3.9:** Assuming  $M=0$  and a baud rate of 9600 bits/sec, what is the bandwidth in bytes/sec?

### 3.6.1 Transmitting in Asynchronous Mode

We will begin with transmission, because it is simpler than reception. The transmitter portion of the SCI includes a TxD data output pin, with TTL voltage levels (see Figure 3.25). The transmitter has a 10- or 11-bit shift register, which cannot be directly accessed by the programmer, and this shift register is separate from the receive shift register. To output data using the SCI, the software will write to the Serial Communications Data Register. The 9S12C32 has just one SCI port, so its data register is called SCIDRL. The 9S12DP512 has two SCI ports, and the data registers are called SCI0DRL and SCI1DRL. The transmit data register is write only, which means the software can write to it (to start a new transmission), but cannot read from it. Even though the transmit data register is at the same address as the receive data register, the transmit and receive data registers are two separate registers. When using 9-bit data mode ( $M=1$ ), we first set the T8 bit, then we write to the transmit data register to start transmission.

**Figure 3.25**  
Data and shift registers implement the serial transmission.



Four control bits affect transmission. We initialize the Transmit Enable control bit, TE, to 1 to enable the transmitter. Interrupt-driven software will be developed in Chapter 7. There are two status bits generated by transmitter activity. The Transmit Data Register Empty flag, TDRE, is set when the transmit SCIDRL is empty. The TDRE bit is cleared by reading the TDRE flag (with it set), then writing to the SCIDRL. The Transmit Complete flag, TC, is set when the transmit shift register is done shifting. The TC is cleared by reading the TC flag (with it set), then writing to the SCIDRL.

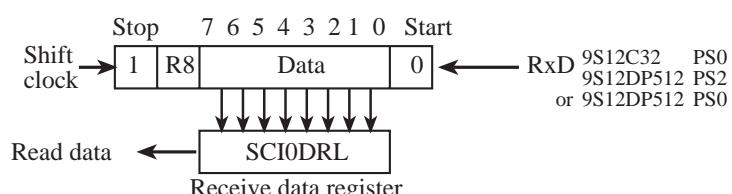
When new data (8 bits) is loaded in the SCIDRL, it is copied into the 10- or 11-bit transmit shift register. Next, the start bit, T8 (if M=1), and stop bits are added. Then the frame is shifted out one bit at a time at a rate specified by the baud rate register. If there is already data in the shift register when the SCIDRL is written, it will wait until the previous frame is transmitted before it too is transferred. The serial port hardware is actually controlled by a clock that is 16 times faster than the baud rate. The digital hardware in the SCI counts 16 times between changes to the TxD output line.

In essence, the SCIDRL and transmit shift register behave together like a two-element first-in-first-out queue (FIFO). In other words, the software can actually write two bytes to the SCIDRL, and the hardware will send them both, one at a time. In fact, the serial port interface chip used in most PC computers has a 16-byte hardware FIFO between the data register and the shift register. A PC that has a 16C550-compatible UART supports this hardware FIFO function. This FIFO reduces the software response time requirements of the operating system to service the serial port hardware.

### 3.6.2 Receiving in Asynchronous Mode

Receiving data frames is a little trickier than transmission, because we have to synchronize the receive shift register with the incoming data. The receiver portion of the SCI includes an RxD data input pin, with TTL voltage levels (see Figure 3.26). There is also a 10- or 11-bit shift register, which cannot be directly accessed by the programmer. Again, the receive shift register is separate from the transmit shift register. The receiver has a Serial Communications Data Register. The receive data register is read only, which means write operations to this address have no effect on this register. When operating in 9-bit mode (M=1), the ninth data bit is saved in the R8 bit.

**Figure 3.26**  
Data register shift  
registers implement the  
receive serial interface.



We will set the Receiver Enable control bit, RE, to 1 to enable the receiver. The Receive Data Register Full flag, RDRF, is set when new input data is available. The RDRF bit is cleared by reading the RDRF flag (with it set), then reading the SCDR. The Overrun flag, OR, is set when input data is lost because previous data frames had not been read. The OR bit is cleared by reading the OR flag (with it set), then reading the SCIDRL. The Noise flag, NF, is set when the input is noisy. The NF bit is cleared by reading the NF flag (with it set), then reading the SCIDRL. Each bit is sampled three times by the receiver. The NF bit is set when any of the groups of three samples do not all agree. NF errors can occur if there is indeed noise on the line, but more likely it is caused by a mismatch between the transmitter and receiver baud rates. The Framing Error, FE, is set when the stop bit is incorrect. The FE bit is cleared by reading the FE flag (with it set), then reading the SCIDRL. Framing errors are also probably caused by a mismatch in baud rate.

The receiver waits for the 1 to 0 edge signifying a start bit, then shifts in 10 or 11 bits of data one at a time from the RxD line. The start and stop bits are removed (checked for noise and framing errors), the 8 bits of data are loaded into the SCIDRL, the 9th data bit is put in R8 (if M=1), and the RDRF flag is set. If there is already data in the SCIDRL when the shift register is finished, it will wait until the previous frame is read by the software before it is transferred.

**Observation:** If the receiving SCI device has a baud rate mismatch of more than 5%, then a framing error can occur when the stop bit is incorrectly captured.

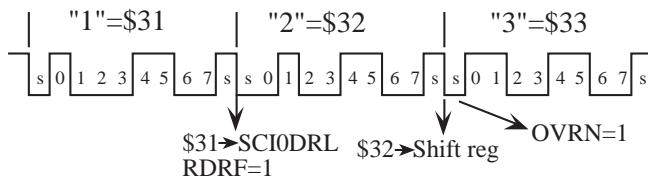
An overrun occurs when there is one receive frame in the SCIDRL, one receive frame in the receive shift register, and a third frame comes into RxD. In order to avoid overrun, we can design a real-time system (i.e., one with a maximum latency). The latency of a SCI receiver is the delay between the time when new data arrives in the receiver SCIDRL and the time the software reads the SCIDRL. If the latency is always less than 10 (11 if M=1) bit times, then overrun will never occur.

**Observation:** With a serial port that has a shift register and one data register (no additional FIFO buffering), the latency requirement of the input interface is the time it takes to transmit one data frame.

In the example illustrated in Figure 3.27, assume the SCI receive shift register and receive data register are initially empty. Three incoming serial frames occur one right after another, but the software does not respond. At the end of the first frame, the \$31 goes into the receive SCIDRL and the RDRF flag is set. In this scenario, the software is busy doing other things and does not respond to the setting of RDRF. Next, the second frame is entered into the receive shift register. At the end of the second frame, there is the \$31 in the SCIDRL and the \$32 in the shift register. If the software were to respond at this point, then both characters would be properly received. If the third frame begins before the first is read by the software, then an overrun error occurs and a frame is lost. We can see from this worst-case scenario that the software must read the data from SCIDRL within 10 bit times of the setting of RDRF.

**Figure 3.27**

Three receive data frames result in an overrun (OR) error.



Next we will overview the specific SCI functions on particular Freescale microcomputers. This section is intended to supplement rather than replace the Freescale manuals. When designing systems with a SCI, please also refer to the reference manual of your specific Freescale microcomputer.

### 3.6.3 9S12 SCI Details

The MC9S12DP512 has two asynchronous serial ports using Port S bits 3,2 and bits 1,0. On the other hand, most 9S12 versions have only one serial port using Port S bits 1,0. Except for the specific addresses listed in Table 3.5, the SCI details in the section will apply to most chips. Each serial port has a 16-bit baud rate register. The full details can be found in the Freescale data sheet. We will assume the SCI is running in the default 8-bit data mode (SCICR1 = 0).

Address	msb													lsb	Name		
\$00C8	-	-	-	12	11	10	9	8	7	6	5	4	3	2	1	0	SCIBD

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$00CA	LOOPS	SWAI	RSRC	M	WAKE	ILT	PE	PT	SCICR1
\$00CB	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCICR2
\$00CC	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCISR1
\$00CF	R7T7	R6T6	R5T5	R4T4	R3T3	R2T2	R1T1	R0T0	SCIDRL

**Table 3.5**

MC9S12C32 SCI ports.

The **SCIBD** register is 16 bits, occupying two bytes. The least significant 13 bits of SCIBD determine the baud rate for the SCI port. If BR is the value written to bits 12:0 and MCLK is the module clock (typically this is the same as the E clock), then the baud rate is

$$\text{SCI Baud Rate} = \frac{\text{MCLK}}{(16 - \text{BR})}$$

**Checkpoint 3.10:** Assume MCLK is 4 MHz. What is the baud rate if SCIBD equals 13?

**Checkpoint 3.11:** Assume MCLK is 25 MHz. What value should SCIBD be for a baud rate of 38400 bits/sec?

The **SCICR2** control register contains the bits that turn on the SCI and the interrupt arm bits. Interrupts will be covered in Chapter 7. **TE** is the Transmitter Enable bit, and **RE** is the Receiver Enable bit. We set both TE and RE equal to 1 in order to activate the SCI device.

The flags in the **SCISR1** register can be read by the software, but cannot be modified by writing to this register. **TDRE** is the Transmit Data Register Empty Flag. It is set by the SCI hardware if transmit data can be written to SCIDRL. If TDRE is zero, the transmit data register contains previous data that has not yet been moved to the transmit shift register. Writing into the SCIDRL when TDRE is zero will result in a loss of data. On the other hand, when this bit is set, the software can begin another output transmission by writing to SCIDRL. This flag is cleared by first reading SCISR1 with TDRE set, followed by a SCIDRL write. **RDRF** is the Receive Data Register Full bit. RDRF is set if a received character is ready to be read from SCIDR. We clear the RDRF flag by reading SCISR1 with RDRF set followed by reading SCIDRL.

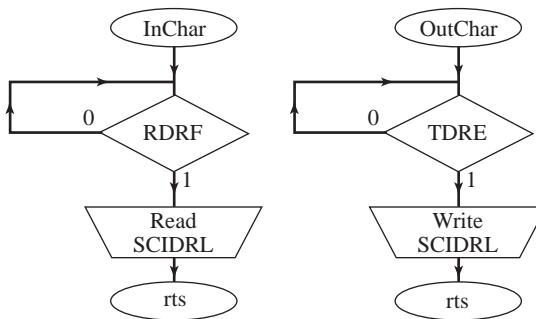
The **SCIDRL** register contains the data transmitted out and received in by the SCI device. Even though there are separate transmit and receive data registers, these two registers exist at the same I/O port address. Reads to SCIDRL access the eight bits of the read-only SCI receive data register. Writes to SCIDRL access the eight bits of the write-only SCI transmit data register.

### 3.6.4 SCI Device Driver

Software that sends and receives data must implement a mechanism to synchronize the software with the hardware. In particular, the software should read data from the input device only when data is indeed ready. Similarly, software should write data to an output device only when the device is ready to accept new data. *Busy-waiting*, or *gadfly* are two equivalent names for the same synchronization method. In this scheme, the software continuously checks the hardware status waiting for it to be ready. In this section, we

will use busy-waiting to write I/O programs that send and receive data using the SCI port. When a new 8-bit character is received into the serial port, RDRF is set and the 8-bit data is available in the serial data register, SCDR. To get new data from the serial port, the software first waits for the flag to be set, then reads the result. Recall that when the software reads SCIDRL, it accesses the receive data register. This operation is illustrated in Figure 3.28 and shown in Program 3.10. In a similar fashion, when the software wishes to output via the serial port, it first waits for TDRE to be set, then performs the output. When the software writes SCIDRL, it sets the transmit data register. An interrupt synchronization method will be presented in Chapter 7.

**Figure 3.28**  
Flowcharts of InChar and OutChar using busy-waiting.



**Program 3.10**  
Device driver functions that implement serial I/O.

<pre> ; Inputs: none      Outputs: none SCI_Init     movb #\$0c,SCICR2 ;enable SCI     movw #13,SCIBD   ;19200 bps     rts  ; Read 8 bits from serial port ; Return 8-bit byte in RegA RDRF equ \$20 SCI_InChar     ldaa SCISR1 ; serial status     anda #RDRF ; available?     beq SCI_InChar ; wait RDRF     ldaa SCIDRL ; read ASCII     rts  ; Write 8 bits to serial port ; Input 8-bit byte in RegA TDRE equ \$80 SCI_OutChar     ldab SCISR1 ; serial status     andb #TDRE ; output ready?     beq SCI_OutChar ; wait TDRE     staa SCIDRL ; write ASCII     rts   </pre>	<pre> void SCI_Init(void){     SCIBD = 13; // 19200 bits/sec     SCICR2 = 0x0C; // enable }  #define RDRF 0x20 // Wait for new input, // then return ASCII code char SCI_InChar(void){     while((SCISR1&amp;RDRF) == 0){};     return(SCIDRL); }  #define TDRE 0x80 // Wait for buffer to be empty, // then output void SCI_OutChar(char data){     while((SCISR1&amp;TDRE) == 0){};     SCIDRL = data; }   </pre>
---	---

The initialization program, `SCI_Init`, enables the SCI device and selects the baud rate. The input routine waits in a loop until RDRF (receive data register full) is set, then reads the data register, SCDR. The function returns the new ASCII character by value in RegA. The output routine first waits in a loop until TDRE (transmit data register empty) is set, then writes data to the data register, SCDR. The function accepts the new ASCII character as a call by value in RegA. As we saw previously in Section 3.3, this is an efficient way to implement an output busy-wait loop.

**Checkpoint 3.12:** Rewrite the SCI\_OutChar, removing one instruction.

**Checkpoint 3.13:** How does the software clear RDRF?

**Checkpoint 3.14:** How does the software clear TDRE?

The **TExaS** simulator has example serial port functions for the 9S12 written in both assembly and C. Once **TExaS** is installed, the busy-wait SCI functions can be found in the TUT2.\* files. The TUT4.\* files implement similar serial port operations using interrupt synchronization.

**Checkpoint 3.15:** Describe what happens if the receiving computer is operating on a baud rate that is twice as fast as the transmitting computer?

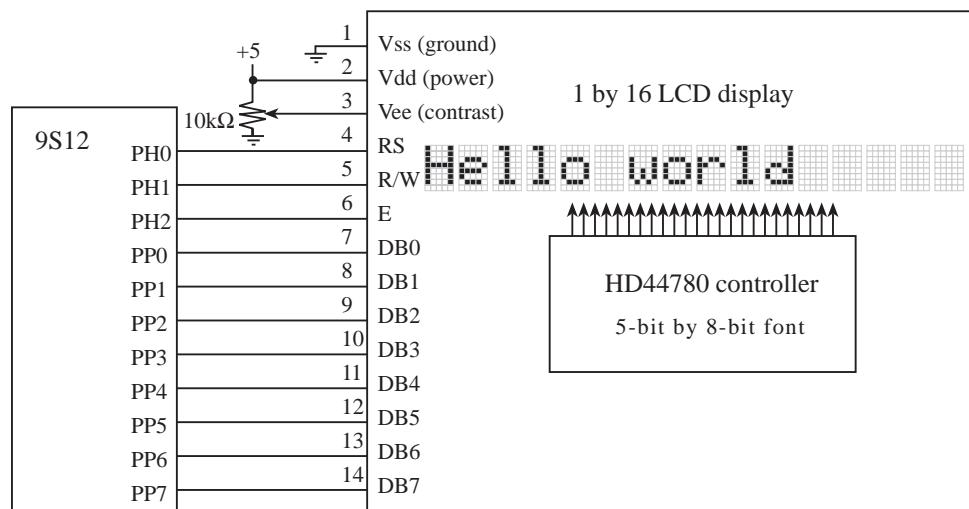
**Checkpoint 3.16:** Describe what happens if the transmitting computer is operating on a baud rate that is twice as fast as the receiving computer?

## 3.7 Parallel Port LCD Interface with the HD44780 Controller

Microprocessor-controlled LCD displays are widely used, having replaced most of their LED counterparts, because of their low power and flexible display graphics. This section will illustrate how a handshaked parallel port of the microcomputer will be used to output to the LCD display. The hardware for the display uses an industry standard HD44780 controller, as shown in Figure 3.29. The low-level software initializes and outputs to the HD44780 controller. The 9S12 simply writes ASCII characters to the HD44780 controller. Each ASCII character is mapped into a 5-bit by 8-bit pixel image, called a font. A 1 by 16 LCD display is 80 pixels wide by 8 pixels, and the HD44780 is responsible for refreshing the pixels in a raster-scanned manner similar to maintaining an image on a TV screen or computer monitor.

**Figure 3.29**

Interface of a HD44780 LCD controller.



There are four types of access cycles to the HD44780 depending on RS and R/W, as shown in Table 3.6.

Normally, you write ASCII characters into the data buffer (called DDRAM in the data sheets) to have them displayed on the screen. However, you can create up to eight new characters the LCD by writing to the character generator RAM (CGRAM). These new characters exist as ASCII data 0 to 7.

**Table 3.6**

Two control signals specify the type of access to the HD44780.

RS	R/W	Cycle
0	0	Write to Instruction Register
0	1	Read Busy Flag (bit 7)
1	0	Write data from the microcontroller to the HD44780
1	1	Read data from the HD44780 to the microcontroller

Two types of synchronization can be used: blind-cycle and busy-waiting. Most operations require 40 µs to complete, while some require 1.64 ms. Programs 3.11 and 3.12 use the timer to create the blind-cycle wait. A busy-wait interface would have provided feedback to detect a faulty interface, but it has the problem of creating a software crash if the LCD never finishes. A better interface would have utilized both busy-wait and blind-cycle, so that the software can return with an error code if a display operation does not finish on time (due to a broken wire or damaged display.) First we present a low-level private helper function (see Program 3.11). This function would not have a prototype in the LCD.H file.

### Program 3.11

Private functions for an HD44780-controlled LCD display.

<pre> E      equ 4    ;PH2 RW     equ 2    ;PH1 RS      equ 1   ;PH0 ; Output command to LCD ; Input: RegA is command ; Output: none OutCmd staa PTP         movb #0,PTH ;E=0, RS=0, R/W=0         movb #E,PTH ;E=1, RS=0, R/W=0         movb #0,PTH ;E=0, RS=0, R/W=0         ldd #40         jsr Timer_Wait ;wait 40us         rts </pre>	<pre> #define E 4 // on PH2 #define RW 2 // on PH1 #define RS 1 // on PH0 void OutCmd(unsigned char command){     PTP = command;     PTH = 0; // E=0, R/W=0, RS=0     PTH = E; // E=1, R/W=0, RS=0     PTH = 0; // E=0, R/W=0, RS=0     Timer_Wait(40); // wait 40us } </pre>
---	---

Next, we show the high-level public functions (see Program 3.12). These functions would have prototypes in the LCD.H file. The initialization sequence is copied from the data sheet of the HD44780.

<pre> ; Initialize HD44780 LCD display ; Inputs: none,   Outputs: none LCD_Init           ;PH3=R/W,PH1=E         movb #\$FF,DDRP ;LCD data         movb #\$FF,DDRH ;PH2=RS         jsr Timer_Init ;lus TCNT         ldy #15         jsr Timer_Wait1ms ;15ms         ldaa #\$38       ;first time         jsr OutCmd         ldy #4         jsr Timer_Wait1ms ;4ms         ldaa #\$38       ;second time         jsr OutCmd         ldd #100         jsr Timer_Wait ;100us         ldaa #\$38       ;third time         jsr OutCmd         ldaa #\$38 ;N=1 two line, F=0 5by7         jsr OutCmd ;DL=1 8-bit data         ldaa #\$08       ;display off </pre>	<pre> // Initialize LCD // Inputs: none // Outputs: none void LCD_init(void){     DDRH = 0xFF;     DDPY = 0xFF;     Timer_Init(); // lus TCNT     Timer_Wait1ms(15); // 15 ms     OutCmd(0x38); // function set     Timer_Wait1ms(4); // 4 ms     OutCmd(0x38); // second time     Timer_Wait(100); // 100us     OutCmd(0x38); // third time     // now the busy flag could be read     OutCmd(0x38);     // 8bit, N=1 2line, F=0 5by7     OutCmd(0x08); // D=0 displayoff     LCD_Clear();     OutCmd(0x0E); // D=1 displayon,     // C=1 cursoron, B=0 blinkoff     OutCmd(0x06); // Entry mode } </pre>
---	--

```

jsr OutCmd
jsr LCD_Clear
ldaa #$0E ;set D=1, C=1, B=0
jsr OutCmd ;cursor on,no blink
ldaa #$06 ;set I/D, S
jsr OutCmd ;inc, no shift
ldaa #$14 ;cursor move
jsr OutCmd ;left
rts
; Output one character to LCD
; Inputs: RegA is ASCII, Outputs: none
LCD_OutChar
    staa PTP
    movb #RS,PTH ;E=0, R/W=0, RS=1
    movb #E+RS,PTH ;E=1, R/W=0, RS=1
    movb #RS,PTH ;E=0, R/W=0, RS=1
    ldd #40
    jsr Timer_Wait ;wait 40us
    rts
LCD_Clear
    ldaa #$01
    jsr OutCmd ;Clear Display
    ldd #1600
    jsr Timer_Wait ;wait 1.6ms
    ldaa #$02
    jsr OutCmd ;Cursor to home
    ldd #1600
    jsr Timer_Wait ;wait 1.6ms
    rts

```

```

// I/D=1 Increment, S=0 nodisplayshift
OutCmd(0x14);
// S/C=0 cursormove, R/L=0 shiftleft
}

// Output a character to the LCD
// Inputs: ASCII character, 0 to 0x7F
// Outputs: none
void LCD_OutChar(unsigned char letter){
// letter is ASCII code
    PTP = letter;
    PTH = RS; // E=0, R/W=0, RS=1
    PTH = E+RS; // E=1, R/W=0, RS=1
    PTH = RS; // E=0, R/W=0, RS=1
    Timer_Wait(40); // 40 us wait
}

// Clear the LCD
// Inputs: none
// Outputs: none
void LCD_Clear(void){
    OutCmd(0x01); // Clear Display
    Timer_Wait(1600); // 1.6 ms wait
    OutCmd(0x02); // Cursor to home
    Timer_Wait(1600); // 1.6 ms wait
}

```

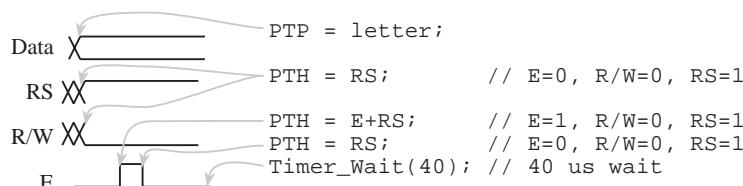
**Program 3.12**

Public functions for an HD44780-controlled LCD display.

Figure 3.30 shows a rough sketch of the E, RS, R/W and data signals as the `LCD_OutChar` function is executed.

**Figure 3.30**

Timing diagram of the LCD signals as data is sent to the HD44780 display.



## 3.8 Exercises

**3.1** In 32 words or less, describe the meaning of each of the following terms.

- Latency
- Real time
- Bandwidth
- I/O bound
- CPU bound
- Blind cycle synchronization
- Busy-waiting synchronization
- Interrupt synchronization

- 3.2** List five different methods for I/O synchronization. In 16 words or less, describe each method.
- 3.3** What does DMA stand for? When do we use DMA?
- 3.4** List one advantage of blind cycle synchronization. List one disadvantage.
- 3.5** What is difference between busy-wait and gadfly synchronization?
- 3.6** What is the difference between regular interrupts and periodic polling?
- 3.7** Simplify  $\downarrow A = 5 + [10, 20] - [5, 15]$
- 3.8** Consider a circuit like Figure 3.7, except using 74LS244 and 74LS374. Rework the timing equations to find the appropriate timing relationship between  $\downarrow G$ ,  $\uparrow C$ ,  $\uparrow C$ , and  $\uparrow G^*$  so the interface still works.
- 3.9** Consider a circuit in Figure 3.7 running at voltage supply VCC of 2 V. Rework the timing equations to find the appropriate timing relationship between  $\downarrow G$ ,  $\uparrow C$ ,  $\uparrow C$ , and  $\uparrow G^*$  so the interface still works.
- 3.10** High-speed CMOS logic will run at a wide variety of supply voltages. Look in the data sheets for the 74HC04, 74HC244 and 74HC374. Make a general observation about the relationship between how fast the chip operates and the power-supply voltage for HC logic.
- 3.11** What is the effect on PIFP for the following C code fragments. In each case, something inappropriate happens (i.e., these are not good examples of code). Answer each separately—not as one long program executed from top to bottom.
- a)** `PIFP = 0;`
  - b)** `PIFP |= 0x01;`
  - c)** `PIFP &= ~0x01;`
- 3.12** Write code to configure Port P bit 5 as a rising-edge, key wake-up input. Write code that waits for a rising edge on PP5.
- 3.13** Write code to configure Port H bit 0 as a falling-edge, key wake-up input. Write code that waits for a falling edge on PH0.
- 3.14** Write code to initialize all of Port P as outputs in power saving mode.
- 3.15** Interface a positive logic switch to PP0 without an external resistor. Write code to configure Port P bit 0 as needed. Write code that waits for the switch to be touched and then released. You may assume the switch does not bounce.
- 3.16** Interface a negative logic switch to PH3 without an external resistor. Write code to configure Port H bit 3 as needed. Write code that waits for the switch to be touched and then released. You may assume the switch does not bounce.
- 3.17** Give a reason why you might want to set bits in the RDRH/RDRJ/RDRP registers for pins that are output. Give a reason why you might want to clear bits in the RDRH/RDRJ/RDRP registers for pins that are output.
- 3.18** Assuming the E clock to be 8 MHz, what value goes into SCIBD to make the baud rate 38400 bits/sec?
- 3.19** Assuming the E clock to be 24 MHz, what value goes into SCIBD to make the baud rate 9600 bits/sec?
- 3.20** Look up the SCI module in the 9S12 data sheet, and describe the similarities and differences between TC and TDRE.
- 3.21** In 32 words or less, describe the difference between TE and TIE in the SCI module.
- 3.22** In 32 words or less, describe the difference between RE and RIE in the SCI module.
- 3.23** Assume you are given a working 9S12 system with a SCI connected to an external device. To save power, you decide to slow down the E clock by a factor of 4. Briefly explain the changes required for the SCI software driver.

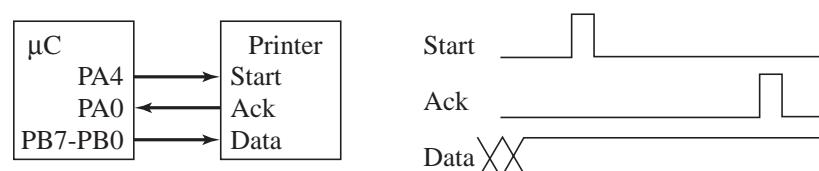
- 3.24** In all your programs, you included some assembler directives depositing the program start address to locations \$FFFE & \$FFFF. For example:

```
org $FFFE
fdb Start
```

where Start is the starting address of your program. Explain the purpose of doing this.

- D 3.25 a)** In this problem you will write a function that outputs data to the printer in Figure 3.31 using a gadfly handshake protocol. You may write the software in assembly or C. The following sequence will print one ASCII character:

**Figure 3.31**  
Simple printer interface.



1. The microcomputer puts the 8-bit ASCII on the Data lines
2. The microcomputer issues a Start pulse (does not matter how wide)
3. The microcomputer waits for the Ack pulse (printer is done)

You do not have to save/restore registers. You may assume the Ack pulse is larger than 10  $\mu$ s. The 8-bit ASCII data to print is *passed by value on the stack*. An example calling sequence is

```
ldab #V
pshb           The ASCII V is pushed on the stack
jsr Output
ins            The V is discarded
```

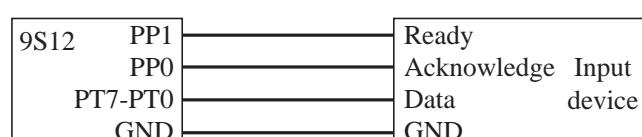
Full credit will be given to the solution that includes the appropriate use of the bset bclr brset and brclr instructions. If you write in C, pass by value with the following prototype:

```
void Output(unsigned char);
```

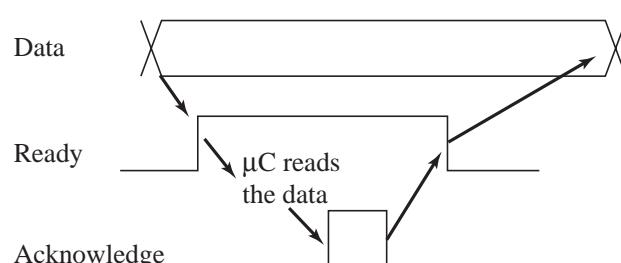
- b)** How long is your Start pulse? Explain your calculation.

- D 3.26** The objective of this problem is to interface an input device to a 9S12 (Figure 3.32) and implement a binary tree command interpreter. You may write the software in assembly or C. The sequence to input an ASCII character from the input device is as follows. When a new character is available, the input device puts its ASCII code on the 8-bit Data, then the input device makes Ready=1. Next your software should read the 8-bit ASCII code, then acknowledge receipt of the data by pulsing Acknowledge. After the pulse, the input device will make Ready=0 again (Figure 3.33).

**Figure 3.32**  
An input device  
interfaced to a 9S12.



**Figure 3.33**  
Timing diagram of the  
interface between an  
input device and a 9S12.



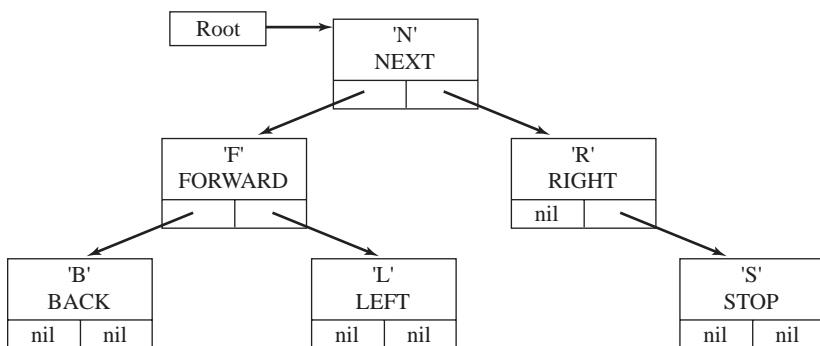
- a) Show the ritual that initializes the microcomputer.  
 b) Show the subroutine that inputs one data byte. The sequence of steps is described above. The input data are returned by value using RegB. Use gadfly synchronization. If you write in C, implement call by reference with the following prototype.

```
unsigned char Input(void);
```

To improve search time, your command interpreter will use a binary tree (Figure 3.34) instead of a table or single-linked list. The six subroutines NEXT, FORWARD, BACK, LEFT, RIGHT, STOP are given (you do not write these six subroutines.) Each node of the tree has one ASCII character (e.g., 'B', 'L', . . .), a 16-bit subroutine address (e.g., BACK, LEFT, . . .), and two pointers to its children. Notice that the tree is defined in alphabetical order. The command interpreter will input one character from the input device, search the tree, and if a match is found, the interpreter will execute the appropriate subroutine.

**Figure 3.34**

A linked binary tree data structure.



- c) Show the pseudo-operations (e.g., FCC, FCB, FDB) that define the above binary tree. You may define a 'nil' pointer as any address that can not be a valid node pointer (e.g., 0, \$1000, \$FFFF)  
 d) Show the command interpreter that initializes (calls the subroutine you wrote in part b), then repeats over and over the following sequence:
1. It inputs one ASCII character (calls the subroutine you wrote in part c)
  2. It looks up that letter in the binary tree
  3. If found then it executes the appropriate subroutine.

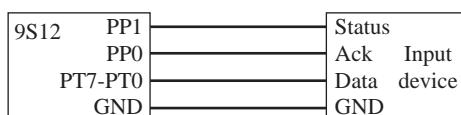
In this first example assume the input letter is 'L'. Your program initializes a pointer with the ROOT (it points to node N.) Since 'L' < 'N', your program updates the pointer in the direction of nodes that are alphabetically before 'N' (now it points to node F). Since 'L' > 'F', your program updates the pointer in the direction of nodes that are alphabetically after 'F' (now it points to node L). Since 'L' = 'L', your program executes the subroutine LEFT.

In this second example assume the input letter is 'P'. Your program initializes a pointer with the ROOT (it points to node N.) Since 'P' > 'N', your program updates the pointer in the direction of nodes that are alphabetically after 'N' (now it points to node R). Since 'P' < 'R', your program updates the pointer in the direction of nodes that are alphabetically before 'R' (now it points to nil). Since the pointer is nil, there is no match.

**D 3.27** The objective of this problem is to interface an input device to a 9S12 (Figure 3.35). You may write the software in assembly or C.

**Figure 3.35**

An input device interfaced to a 9S12.



The timing shown in Figure 3.36 occurs when 1 byte is transferred.

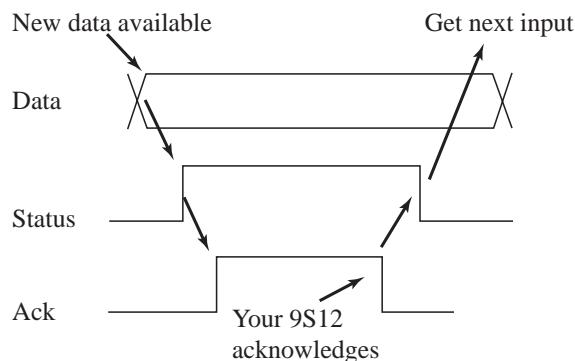
- a) Show the ritual that initializes the microcomputer.  
 b) Show the subroutine that inputs one data byte. Use gadfly synchronization. If you write in assembly, return the new data by reference using RegY. RegY points to the place to

store the next data byte. If you write in C, implement call by reference with the following prototype:

```
void Input(unsigned char *);
```

**Figure 3.36**

Timing diagram of the interface between an input device and a 9S12.



**D 3.28** The objective of this problem is to interface an output device to a 9S12 (Figure 3.37). You may write the software in assembly or C.

**Figure 3.37**

An output device interfaced to a 9S12.



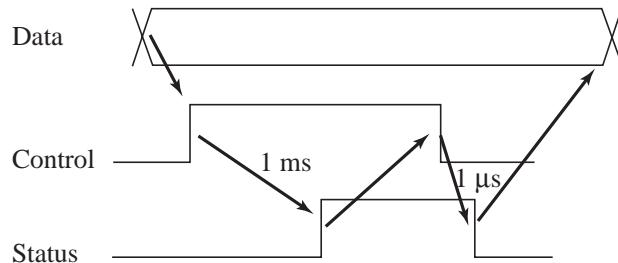
First, the microcomputer outputs data on PT7-PT0. Then, the microcomputer makes Control = 1; 1 ms later the output device will make Status = 1 (the microcomputer waits for Status = 1). Then, the microcomputer makes Control = 0; 1  $\mu$ s later the output device will make Status = 0 (the microcomputer need not check for Status = 0) (Figure 3.38).

- Show the ritual that initializes the microcomputer. Use assembly or C.
- Show the subroutine that outputs one data byte. The sequence of steps is described above. If you write in assembly, the output data are passed by value using RegB. Use gadfly synchronization. If you write in C, implement call-by-value with the following prototype:

```
void OUTPUT(unsigned char);
```

**Figure 3.38**

Timing diagram of the interface between an output device and a 9S12.



**D3.29** The objective of this problem is to interface a position transducer array to a 9S12 (Figure 3.39). You may use any available I/O port. Use busy-wait synchronization. You may write the software in assembly or C. The sequence to read a 5-bit sensor position from the input device is as follows:

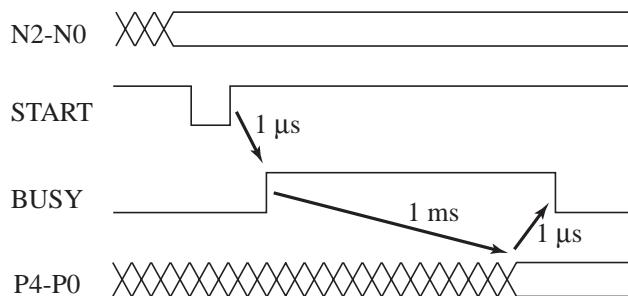
The microcomputer specifies which of eight sensors is to be read (sets N2, N1, N0)

The microcomputer tells the sensor array to read the position by a negative logic pulse on START

The sensor signals it is working by setting BUSY=1

After about 1 ms the 5-bit position is available on the lines P4, P3, P2, P1, P0

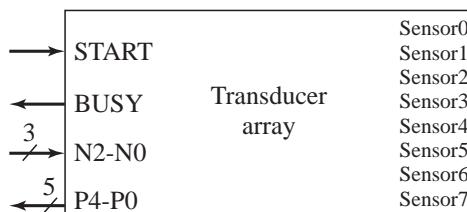
**Figure 3.39**  
Timing diagram for a position sensor.



The sensor signals it is done by setting BUSY=0  
The microcomputer reads the 5-bit position

- a) Show the connections between the position transducer array and the microcomputer.  
Label chip numbers but not pin numbers of any required digital logic (Figure 3.40).

**Figure 3.40**  
An eight-channel sensor array.



```
unsigned char data[8];           /* Room for all 8 results      */
void main(void) { unsigned char N; /* Sensor position 0,1,2..., 7 */
    RITUAL();                   /* Initialize uC Part c          */
    for (N=0;N<8;N++)
        data[N]=SENSOR(N);     /* Read sensor N Part d          */
}
```

- b) Show the RITAL() procedure that initializes the microcomputer.  
c) Show the SENSOR(N) function that reads one data byte. The sequence of steps is described above. Use gadfly synchronization.

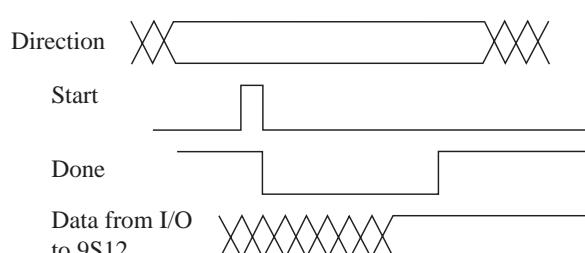
- D 3.30** The objective of this problem is to interface an I/O device to a 9S12 (Figure 3.41). The same interface must be capable of both input and output. You may write the software in assembly or C.

**Figure 3.41**  
An I/O device interfaced to a 9S12.



The sequence to receive an 8-bit data value from the input device to the microcomputer is as follows. First the microcomputer sets Direction=1 and issues a Start pulse. When the input device has the data ready, it will drive its data lines and signal with Done=1. The microcomputer should then read the data (Figure 3.42).

**Figure 3.42**  
Input timing diagram of the interface between an I/O device and a 9S12.



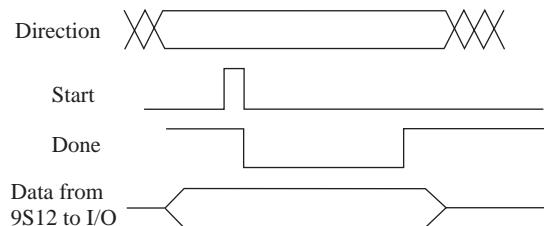
- a) Show the C (or assembly language) procedure that initializes the microcomputer.  
 b) Show the C (or assembly language) procedure that inputs one data byte. Assume the ritual in part a has been called. The sequence of steps is described above. The input data is returned by value. Use busy-wait synchronization. If you write in C, implement call by reference with the following prototype:

```
void Input(unsigned char *);
```

The sequence to transmit an 8-bit data value from the microcomputer to the output device is as follows. First the microcomputer sets `Direction=0`, places the desired 8-bit information on the data lines, and issues a `Start` pulse. When the output device is complete, it signals with `Done=1`. The microcomputer should then make its data output hiZ (tristate) (Figure 3.43).

**Figure 3.43**

Output timing diagram of the interface between an I/O device and a 9S12.



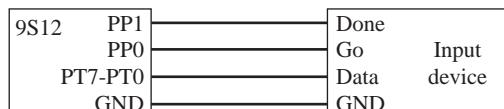
- c) Show the C (or assembly language) procedure that outputs one data byte. Assume the ritual in part a has been called. The sequence of steps is described above. The output data is passed by value. Use busy-wait synchronization. If you write in C pass by value with the following prototype:

```
void Output(unsigned char);
```

**D 3.31** The objective of this problem is to interface an input device to a 9S12 (Figure 3.44). You may write the software in assembly or C.

**Figure 3.44**

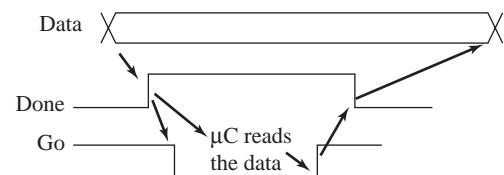
An input device interfaced to a 9S12.



The sequence to input an ASCII character from the input device is as follows. When a new character is available, the input device puts its ASCII code on the 8-bit `Data`, then the input device makes `Done=1`. Make `Go=0` immediately. Recognizing that `Done=1`, your software should read the 8-bit ASCII code, then acknowledge receipt of the data by making `Go=1` again. After the `Go=1`, the input device will make `Done=0` again (Figure 3.45).

**Figure 3.45**

Timing diagram of the interface between an input device and a 9S12.



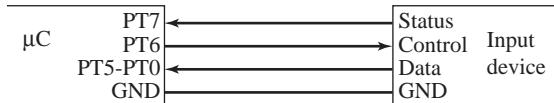
- a) Show bit by bit your choice (what and why) for the parallel I/O control register. Specify that 1 for this bit must be 1, 0 for this bit must be 0, and X for this bit can be written with either 0 or 1.  
 b) Show the C program (ritual) that initializes the microcomputer.  
 c) Show the C procedure that inputs one data byte. The sequence of steps is described above. The input data is returned by value. Use gadfly synchronization. If you write in C, implement a function return value with the following prototype:

```
unsigned char Input(void);
```

**D 3.32** The objective of this problem is to interface the following input device to a 9S12 single-chip computer using any available I/O port (Figure 3.46). The figure shows the connections to Port T. You may write the software in assembly or C.

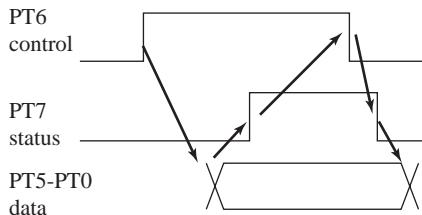
**Figure 3.46**

An input device interfaced to a microcomputer.



**Figure 3.47**

Timing diagram of the interface between an input device and a microcomputer.



- a) Show your choice in hexadecimal for data direction register, e.g., DDRT.
- b) Show the assembly language subroutine that inputs one 6-bit data. The sequence of steps is described above. The input data is returned by value in RegA. Use busy-wait synchronization. Full credit will be given to the subroutine that appropriately uses the bset bclr brset and/or brclr assembly language instructions. If you write in C, implement a function return value with the following prototype:

```
unsigned char Input(void);
```

## 3.9 Lab Assignments

**Lab 3.1** The overall objective is to create a serial port device driver that supports fixed-point input/output. The format will be 16-bit unsigned decimal fixed-point with a resolution of 0.001. You will be able to find on the book website implementations of a serial port driver that supports character, string, and integer I/O, which will be similar to those presented in Section 3.6. In particular, you will design, implement, and test two routines called SCI\_FixIn and SCI\_FixOut. SCI\_FixIn will accept input from the SCI similar to the function SCI\_InUDec. SCI\_FixOut will transmit output to the SCI similar to the function SCI\_OutUDec. During the design phase of this lab, you should define the range of features available. You should design, implement, and test a main program that illustrates the range of capabilities.

**Lab 3.2** The same as Lab 3.1, except the format will be 16-bit signed binary fixed-point with a resolution of  $2^{-8}$ .

**Lab 3.3** The same as Lab 3.1, except the format will be 32-bit unsigned integer.

**Lab 3.4** Design a four-function (add, subtract, multiply, divide) calculator using fixed-point math and the SCI for input/output. The format should be 16-bit signed decimal fixed-point with a resolution of 0.01. The syntax of the calculator should be Reverse Polish with a data stack.

**Lab 3.5** The overall objective of this lab is to design, implement, and test a parallel port expander. Using less than 16 I/O pins of your microcontroller, you will design hardware and software that supports two **8-bit latched input** ports and two strobed output ports. Each input port has 8 data lines and

one **latch** signal. On the rising of each **latch**, your system should capture (latch) the data lines. Each output port has 8 data lines and one **strobe** signal. The hardware/software system should generate a pulse out on **strobe** whenever new output is sent. The output ports do not need to be readable. For example, consider using four 74HC374 registers.

**Lab 3.6** The overall objective of this lab is to design, implement, and test a parallel output port expander. Using just three I/O pins of your microcontroller, you will design hardware and software that supports four 8-bit output ports. The output ports do not need to be readable. For example, consider using four 74HC595 registers.

**Lab 3.7** The overall objective of this lab is to design, implement, and test a parallel input port expander. Using just three I/O pins of your microcontroller, you will design hardware and software that supports four 8-bit input ports. The input ports do not need to be latched. For example, consider using four 74HC597 registers.

**Lab 3.8** The overall objective is to create a HD44780-controlled LCD device driver that supports fixed-point output. The format will be 16-bit unsigned decimal fixed-point with resolutions of 0.1, 0.01, and 0.001. You will be able to find (on the book website) implementations of a HD44780-controlled LCD device driver that supports character, string, and integer I/O. In particular, you will design, implement, and test three routines called `LCD_FixOut1`, `LCD_FixOut2` and `LCD_FixOut3`, implementing fixed-point resolutions 0.1, 0.01, and 0.001 respectively. During the design phase of this lab, you should define the range of features available. You should design, implement, and test a main program that illustrates the range of capabilities.

**Lab 3.9** The same as Lab 3.8, except the format will be 16-bit signed decimal fixed-point with resolutions of 0.1, 0.01 and 0.001. You will design, implement, and test three routines.

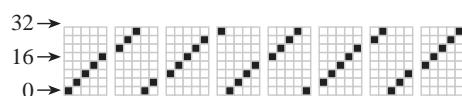
**Lab 3.10** The same as Lab 3.8, except the format will be 16-bit signed binary fixed-point with resolutions of  $2^{-4}$ ,  $2^{-8}$ , and  $2^{-12}$ . You will design, implement, and test three routines.

**Lab 3.11** The same as Lab 3.8, except the formats will be 12-, 24, and 32-bit unsigned integers. You will design, implement, and test three routines.

**Lab 3.12** The overall objective is to create a HD44780-controlled LCD device driver that supports voltage versus time graphical output. The display will be 8 pixels high by 40 pixels wide, using eight character positions on the LCD. You will need to continuously write to the CGRAM creating new fonts, and then output the new font as data to create the images. The initialization routine will clear the graph and set the minimum and maximum range scale on the voltage axis (y-axis). The plot routine takes a voltage data point between minimum and maximum and draws one pixel on the 8 by 40 display. It takes 40 calls to plot to complete one image on the display illustrated in Figure 3.48. Subsequent calls to plot should remove the point from 40 calls ago and draw the new point. If the plot is called at a fixed period, the display should show a continuous sweep of the voltage versus time data, like an untriggered oscilloscope. During the design phase of this lab, you should define the range of features available. You should design, implement, and test a main program that illustrates the range of capabilities.

**Figure 3.48**

Image after 40 calls to plot with the LCD is initialized to Max=32, Min=0. The software outputs this repeating pattern 0,4,8,12,16, 20,24,28,0,4,8,12,16...



# 4 Interrupt Synchronization

## Chapter 4 objectives are to:

- ❖ Introduce the concept of interrupt synchronization
- ❖ Discuss the issues involved in reentrant programming
- ❖ Implement and apply the FIFO circular queue
- ❖ Discuss the specific details of using interrupts on the 9S12
- ❖ Interface a keyboard and printer using key wake-up interrupts
- ❖ Show a high-priority/low-latency XIRQ power failure interface
- ❖ Define mechanisms to establish priority, including round robin
- ❖ Design and implement background I/O for simple devices using periodic polling

There are many reasons to consider interrupt synchronization. The first consideration is that the software in a real-time system must respond to hardware events within a prescribed time. To illustrate the need for interrupts, consider a keyboard interface where the time between new keyboard inputs might be as small as 10 ms. In this situation, the software latency is the time from when the new keyboard input is ready until the time the software reads the new data. To prevent loss of data in this case, the software latency must be less than 10 ms. We can implement real-time software using busy-wait synchronization only when the size and complexity of the system are very small. Interrupts are important for these real-time systems because they provide a mechanism to guarantee an upper bound on the software response time. Interrupts also give us a way to respond to infrequent but important events. Alarm conditions like low battery power and error conditions can be handled with interrupts. Periodic interrupts, generated by the timer at a regular rate, will be necessary to implement data acquisition and control systems. In the unbuffered interfaces of Chapter 3, the hardware and software took turns waiting for each other. Interrupts provide a way to buffer the data so that the hardware and software spend less time waiting. In particular, the buffer we will use is a FIFO queue placed between the interrupt routine and the main program to increase the overall bandwidth. We will begin our discussion with general issues, then present the specific details about the 9S12 microcomputer. After that, a number of simple interrupt examples will be presented, and at the end of the chapter we will discuss some advanced concepts like priority, round-robin polling, and periodic polling.

## 4.1 What Are Interrupts?

### 4.1.1 Interrupt Definition

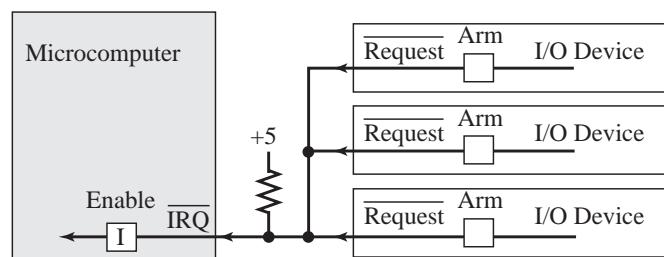
An *interrupt* is the automatic transfer of software execution in response to hardware that is asynchronous with the current software execution. The hardware can be either an external I/O device (like a keyboard or printer) or an internal event (like an opcode fault or a periodic timer). When the hardware needs service (busy-to-done state transition), it will request an interrupt. Recall from Chapter 2 that a *thread* is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a *background* thread. This thread is created by the hardware interrupt request and is killed when the interrupt service routine executes the `rti` instruction. A new thread is created for each interrupt request. Each time the interrupt service routine runs, we consider it as a separate thread, because local variables and registers used in the interrupt service routine are unique, and separate from one interrupt event to the next. In a multithreaded system we consider the threads as cooperating to perform an overall task. Consequently we will develop ways for the threads to communicate (see Section 4.3 on FIFO queues) and synchronize (see the discussion of semaphores in Section 5.2) with each other. Most embedded systems have a single common overall goal. On the other hand, general-purpose computers can have multiple unrelated functions to perform. A *process* is also defined as the action of software as it executes. The difference is processes do not necessarily cooperate toward a common shared goal. Threads share access to global variables, while processes have separate globals.

The software has dynamic control over aspects of the interrupt request sequence. First, each potential interrupt source has a separate arm bit that the software can activate or deactivate. The software will set the arm bits for those devices it wishes to accept interrupts from, and it will deactivate the arm bits within those devices from which interrupts are not to be allowed. In other words, it uses the arm bits to individually select which devices will and which devices will not request interrupts. The second aspect that the software controls is the interrupt enable bit, I, which is in the condition code register. The software can enable all armed interrupts by setting I=0, or it can disable all interrupts by setting I=1. The disabled interrupt state (I=1) does not dismiss the interrupt requests; rather it postpones them until a later time, when the software deems it convenient to handle the requests. We will pay special attention to these enable/disable software actions. In particular we will need to disable interrupts when executing nonreentrant code, but disabling interrupts will have the effect of increasing the response time of software.

There are two general methods with which we configure external hardware so that it can request an interrupt. The first method is a shared negative-logic-level-active request like  $\overline{\text{IRQ}}$ . All the devices that need to request interrupts have an open-collector negative-logic interrupt request line. The hardware requests service by pulling the interrupt request  $\overline{\text{IRQ}}$  line low. The line over the IRQ signifies negative logic. In other words, an interrupt is requested when  $\overline{\text{IRQ}}$  is zero. Because the request lines are open-collector, a pull-up resistor is needed to make  $\overline{\text{IRQ}}$  high when no devices need service (Figure 4.1).

**Figure 4.1**

Wire- or negative-logic interrupt request line.

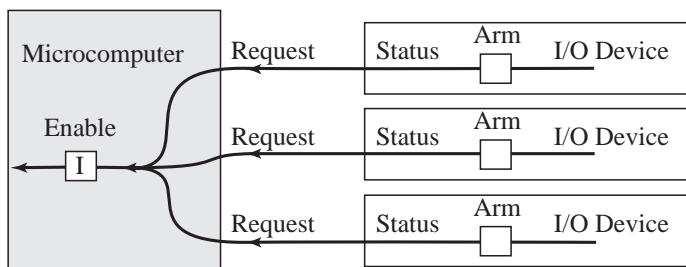


Normally these interrupt requests share the same interrupt vector. This means that whichever device requests an interrupt, the same interrupt service routine is executed. Therefore the interrupt service routine must first determine which device requested the interrupt. The 9S12 has two of these shared negative-logic-level-active interrupts,  $\overline{IRQ}$  and  $XIRQ$ . XIRQ interrupts on the 9S12 are nonmaskable. In other words, once they are enabled (the X bit in the CCR is 0), they cannot be disabled.

The second method uses multiple dedicated *edge-triggered* requests (Figure 4.2). In this method, each device has its own interrupt request line that is usually connected to a status signal in the I/O device. In this way, an interrupt is requested on the *busy-to-done* state transition. Edge-triggered means the interrupt is requested on the rise, the fall, or both the rise and the fall of the *Status* signal. With *vectored* interrupts these individual requests have a unique interrupt vector. This means there can be a separate interrupt service routine for each device. In this way the microcomputer will automatically execute the appropriate software interrupt handler when an interrupt is requested. Therefore the interrupt service routine does not need to determine which device requested the interrupt.

**Figure 4.2**

Dedicated edge-triggered interrupt request lines.



The original IBM PC had only eight dedicated edge-triggered interrupt lines, and the second generation IBM PC I/O bus has only 15. This small number can be a serious limitation in a computer system with many I/O devices.

**Observation:** Microcomputer systems running in expanded mode often use shared negative-logic-level-active interrupts for their external I/O devices.

**Observation:** Microcomputer systems running in single-chip mode often use dedicated edge-triggered interrupts for their I/O devices.

**Observation:** The number of interrupting devices on a system using dedicated edge-triggered interrupts is limited when compared to a system using shared negative-logic-level-active interrupts.

**Observation:** Most Freescale microcomputers support both shared negative-logic and dedicated edge-triggered interrupts.

The advantages of a wire- or negative-logic interrupt request are (1) additional I/O devices can be added without redesigning the hardware, (2) there is no fundamental limit to the number of interrupting I/O devices you can have, and (3) the microcomputer hardware is simple. The advantages of the dedicated edge-triggered interrupt request are (1) the software is simpler, therefore it will be easier to debug and it will run faster, (2) there will be less coupling between software modules, making it easier to debug and to reuse code, and (3) it will be easier to implement priority such that higher-priority requests are handled quickly, while lower-priority interrupt requests can be postponed. Because the Freescale microcomputers support both types of interrupt, the designer can and must address these considerations.

#### 4.1.2 Interrupt Service Routines

The interrupt service routine (ISR) is the software module that is executed when the hardware requests an interrupt. From Section 4.1.1, we see that there may be one large ISR that handles all requests (polled interrupts) or many small ISRs specific for each potential

source of interrupt (vectored interrupts). The design of the ISR requires careful consideration of many factors that will be discussed in this chapter. When an interrupt is requested (and the device is armed and the I bit is zero), the microcomputer will service an interrupt:

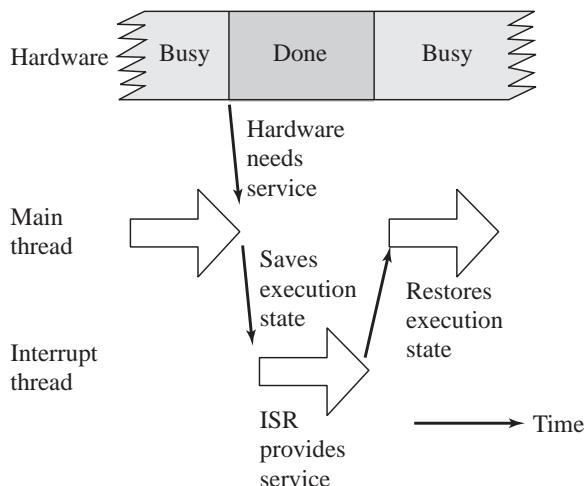
1. The execution of the main program is suspended (the current instruction is finished)<sup>1</sup>
2. The ISR, or background thread, is executed
3. The main program is resumed when the ISR executes `rti`

When the microcomputer accepts an interrupt request, it will automatically save the execution state of the main thread by pushing its registers on the stack (i.e., PC, Y, X, A, B, CCR). After the ISR provides the necessary service, it will execute a `rti` instruction. This instruction pulls the registers from the stack, which returns control to the main program. Execution of the main program will then continue with the exact stack and register values that existed before the interrupt. Although interrupt handlers can allocate, access, then deallocate local variables, parameters passing between threads must be implemented using global memory variables. Global variables are also required if an interrupt thread wishes to pass information to itself, (e.g., from one interrupt instance to another). The execution of the main program is called the *foreground thread*, and the executions of ISRs are called *background threads* (Figure 4.3).

**Checkpoint 4.1:** What value of the I bit enables interrupts to occur?

**Figure 4.3**

An interrupt causes the main thread to be suspended, and the interrupt thread is run.



### 4.1.3 When to Use Interrupts

The factors listed in Table 4.1 should be considered when deciding the most appropriate mechanism to synchronize hardware and software. One should not always use busy-waiting just because one is too lazy to implement the complexities of interrupts. On the other hand, one should not always use interrupts just because they are fun and exciting. Direct Memory Access (DMA) will be covered in Chapter 10.

### 4.1.4 Interthread Communication

For regular function calls we use the registers and stack to pass parameters, but interrupt threads have logically separate registers and stack. In particular, all registers are automatically saved by the microcomputer as it switches from the main program (foreground thread) to the ISR (background thread). The `rti` instruction will restore the registers (including the interrupt enable bits and the PC) back to their previous values.

<sup>1</sup>The 9S12 instructions `rev`, `revw`, and `wav` can be interrupted in the middle of their execution.

**Table 4.1**

Each synchronization method has motivations for its use.

Busy-Wait	Interrupts	DMA
Predictable	Variable arrival times	Low latency
Simple I/O	Complex I/O, different speeds	High bandwidth
Fixed load	Variable load	
Dedicated, single thread	Other functions to do	
Single process	Multithread or multiprocess	
Nothing else to do	Infrequent but important alarms	
	Program errors	
	Overflow, invalid opcode	
	Illegal stack or memory access	
	Machine errors	
	Power failure, memory fault	
	Breakpoints for debugging	
	Real-time clocks	
	Data acquisition and control	

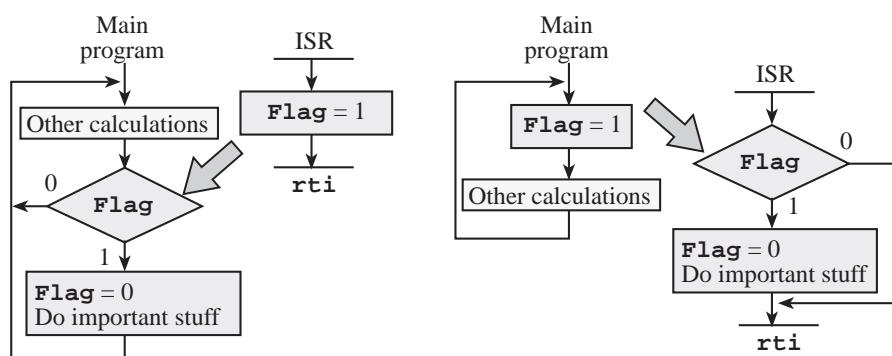
Thus, all parameter passing must occur through global memory. One cannot pass data from the main program to the ISR using registers or the stack.

In this chapter, multithreading means one main program (foreground thread) and multiple ISRs (background threads). In Chapter 5, we present techniques that allow multiple foreground threads. Synchronizing threads is a critical task affecting the efficiency and effectiveness of systems using interrupts. In this section, we will present (in general form) three constructs to synchronize threads: binary semaphore, mailbox, and FIFO queue.

A *binary semaphore* is simply a shared flag. There are two operations one can perform on a semaphore. *Signal* is the action that sets the flag. *Wait* is the action that checks the flag, and if the flag is set, the flag is cleared, and important stuff is performed. As mentioned in the previous paragraph, this flag must exist as a shared global variable. In order to reduce complexity of the system, it will be important to limit the access to this flag to as few modules as possible. A flag of course has two states: 0 and 1. However, it is good design to assign a meaning to this flag. For example, 0 means the switch has not been pressed, and 1 means the switch has been pressed. Figure 4.4 shows two examples of the binary semaphore. The big arrow in this figure signifies the synchronization link between the threads. In the example on the left, the ISR signals the semaphore, and the main program waits on the semaphore. Notice the “important stuff” is run in the foreground once per execution of the ISR. In the example on the right, the main program signals the semaphore, and the ISR waits. It is good design to have NO backwards jumps in an ISR. In this particular application, if the ISR is running and the semaphore is 0, the action is just skipped and the computer returns from the interrupt.

**Figure 4.4**

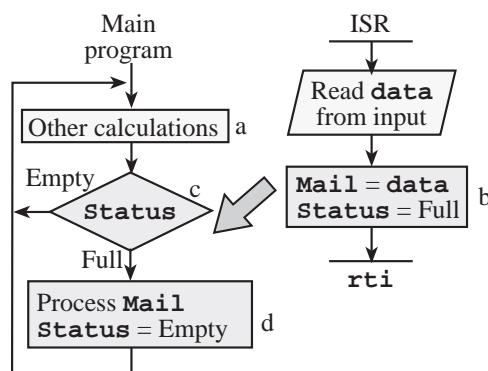
A semaphore can be used to synchronize threads.



The second interthread synchronization scheme is the *mailbox*. The mailbox is a binary semaphore with an associated data variable. Figure 4.5 illustrates an input device interfaced using interrupt synchronization. The big arrow in this figure signifies the communication/synchronization link between the background and foreground. The mailbox structure is implemented with two shared global variables. `Mail` contains data, and `Status` is a semaphore flag specifying whether the mailbox is full or empty. The interrupt is requested when its trigger flag is set, signifying new data are ready from the input device. The ISR will read the data from the input device, store it in the shared global variable `Mail`, and then update its status to full. The main program will perform other calculations while occasionally checking the status of the mailbox. When the mailbox has data, the main program will process it. This approach is adequate for situations where the input bandwidth is slow compared to the software processing speed. It is also possible to process the data within the ISR itself and just report the results of the processing to the main program using the mailbox.

**Figure 4.5**

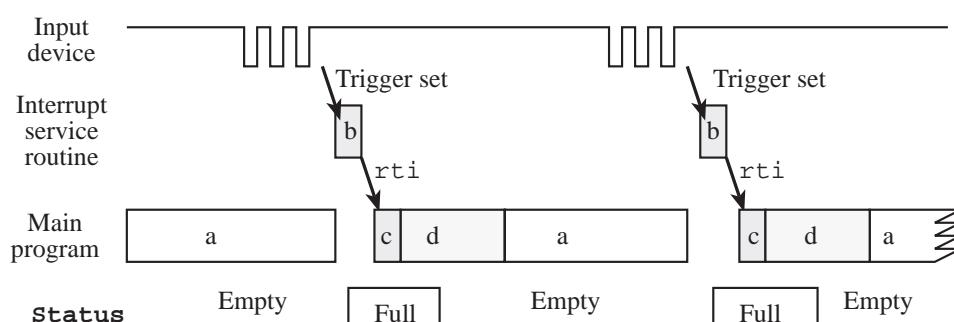
A mailbox can be used to pass data between threads.



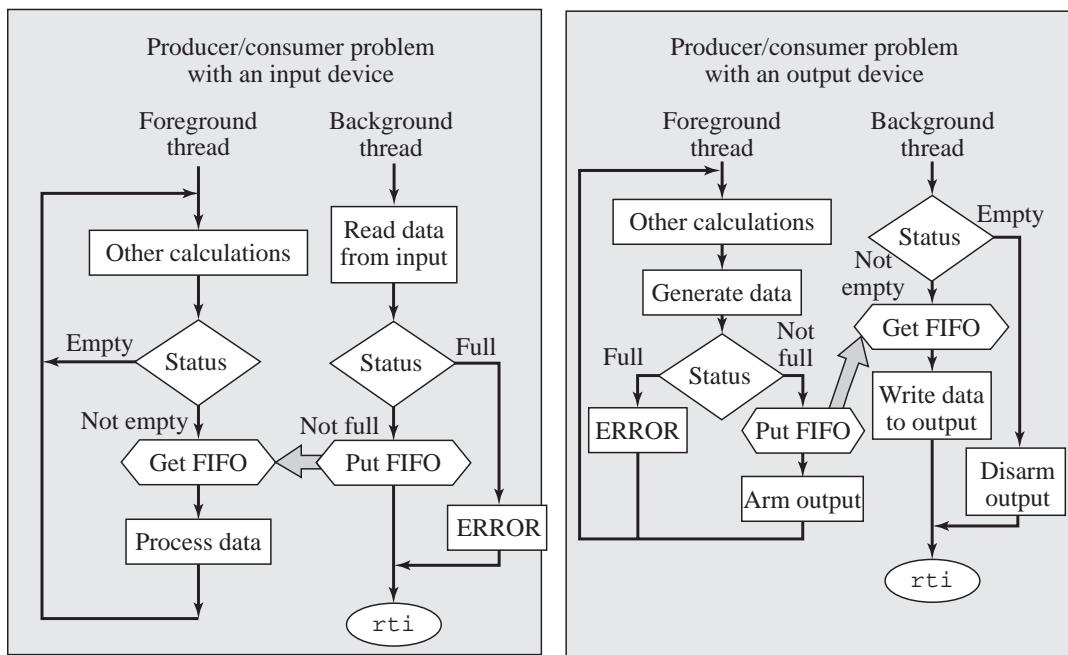
One way to visualize the interrupt synchronization is to draw a state-versus-time plot of the activities of the hardware, the mailbox, and the two software threads. Figure 4.6 shows that at time (a) the mailbox is empty, the input device is idle, and the main program is performing other tasks because the mailbox is empty. When new input data are ready, the trigger flag will be set, and an interrupt will be requested. At time (b), the ISR reads data from the input device, saves it in `Mail`, and then sets `Status` to full. At time (c), the main program recognizes `Status` is full. At time (d), the main program processes data from `Mail` and sets `Status` to empty. Notice that even though there are two threads, only one is active at a time. The interrupt hardware switches the processor from the main program to the ISR, and the `rti` instruction switches the processor back.

**Figure 4.6**

Hardware/software timing of an input interface using a mailbox.



The third synchronization technique is the FIFO queue, shown in Figure 4.7. With mailbox synchronization, the threads execute in lock-step: one, the other, one, the other, . . . However, with the FIFO queue, execution of the threads is more loosely coupled.



**Figure 4.7**

FIFO queues can be used to pass data between threads.

The classic producer/consumer problem has two threads. One thread produces data, and the other consumes data. For an input device, the background thread is the producer because it generates new data, and the foreground thread is the consumer because it uses the data up. For an output device, the data flows in the other direction so that the producer/consumer roles are reversed. It is appropriate to pass data from the producer thread to the consumer thread using a FIFO queue (Figure 4.7).

**Observation:** For systems with interrupt-driven I/O on multiple devices, there will be a separate FIFO for each device.

An input device needs service (busy-to-done state transition) when new data are available. The ISR (background) will accept the data and process it. If more data are desired, the ISR will restart the input hardware (done-to-busy state transition).

An output device needs service (busy-to-done state transition) when the device is idle, ready to output more data. The ISR (background) will get more data and output it. The output function will restart the hardware (done-to-busy state transition). For this output device example, it would be appropriate to GET more data from a FIFO queue and send it to the device. Two particular problems with output device interrupts are:

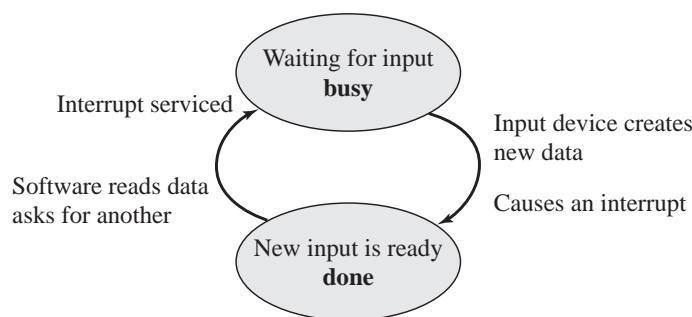
1. How does one generate the first interrupt? In other words, how does one start the output thread?
2. What does one do if an output interrupt occurs (device is idle) but there are no more data currently available (e.g., FIFO is empty)?

These problems can be solved by arming and disarming when needed.

The foreground thread (main program) executes a loop and accesses the appropriate FIFO when it needs to input or output data. The background threads (interrupts) are executed when the hardware needs service. For an input device, an interrupt is requested (causing the ISR to be executed) when new input data are available. The busy-to-done state transition causes an interrupt. These hardware state transitions are illustrated in Figures 4.8 and 4.11.

**Figure 4.8**

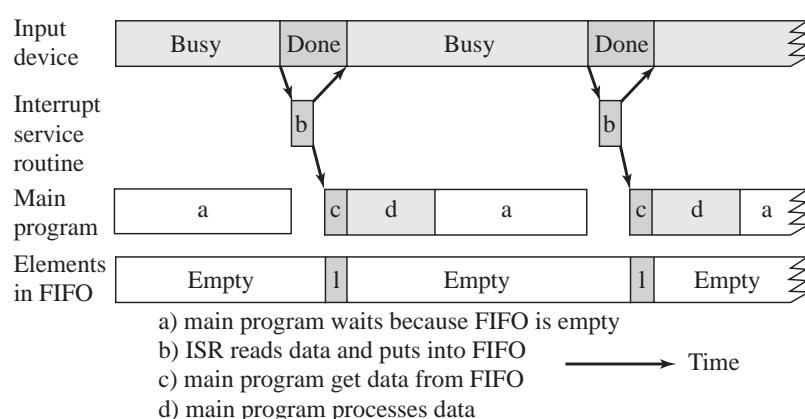
The input device interrupts the computer when it has new data.



One way to visualize the interrupt synchronization is to draw a state versus time plot of the activities of the hardware and the two software modules (Figure 4.9). For an input device, the main thread begins by waiting for new input. When the input device is busy, it is in the process of creating new input. When the input device is done, new data are available and an interrupt is requested. The ISR will read the data and put them into the FIFO. Once data are in the FIFO, the main program is released to go on. The arrows from one graph to the other represent the synchronizing events. In this first example, the time for the software to read and process the data is less than the time for the input device to create new input. This situation is called *I/O-bound*. In this situation, the FIFO has either 0 or 1 entry, and the use of interrupts does not enhance the bandwidth over the busy-wait implementations presented in Chapter 3. Even with an I/O-bound device it may be more efficient to utilize interrupts because it provides a straightforward approach to servicing multiple devices.

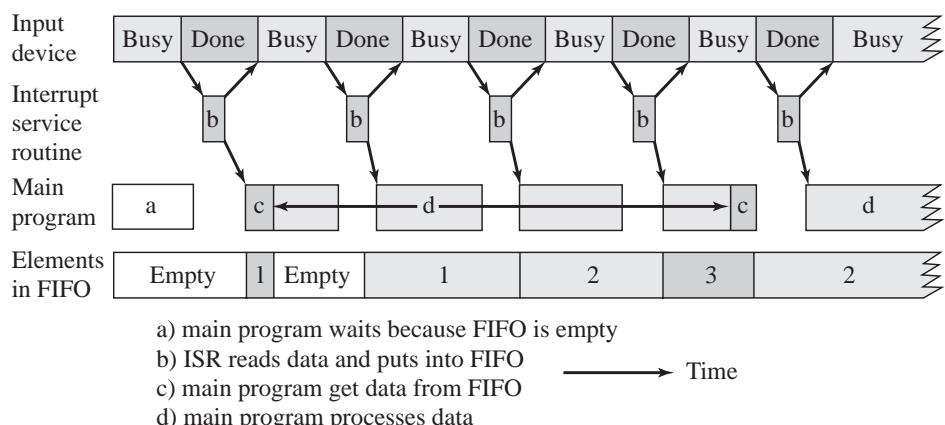
**Figure 4.9**

Hardware/software timing of an I/O-bound input interface.



In this second example, the input device starts with a burst of high-bandwidth activity (Figure 4.10). As long as the ISR is fast enough to keep up with the input device and as long as the FIFO does not become full, no data are lost. In this situation, the overall

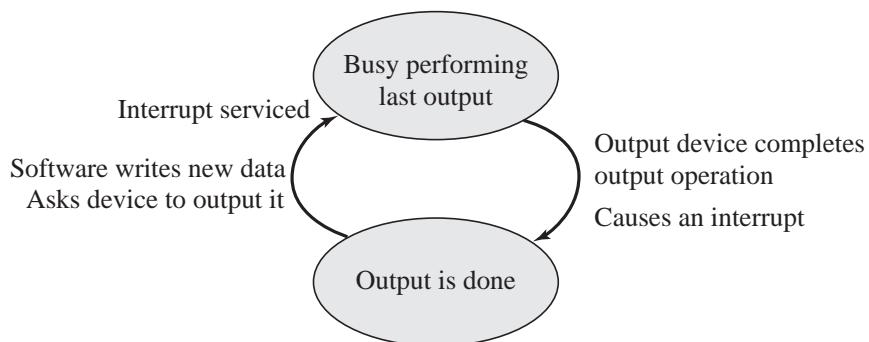
**Figure 4.10**  
Hardware/software timing of an input interface during a high-bandwidth burst.



bandwidth is higher than it would be with a busy-wait implementation, because the input device does not have to wait for each data byte to be processed. This is the classic example of a *buffered* input, because data enter the system (via the interrupts), are temporarily stored in a buffer (put into the FIFO), and are processed later (by the main program, get from the FIFO). When the I/O device is faster than the software, the system is called *CPU-bound*. As we will see later, this system can work if the producer rate temporarily exceeds the consumer rate (a short burst of high-bandwidth input). If the external device sustained the high-bandwidth input rate, then the FIFO would become full and data would be lost.

For an output device, the interrupt is requested when the output is idle and ready to accept more data (Figure 4.11). The busy-to-done state transition causes an interrupt. The ISR gives the output device another piece of data to output.

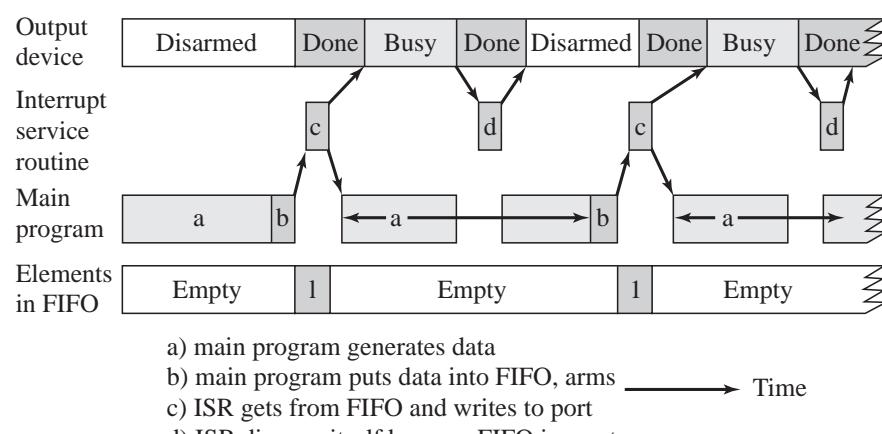
**Figure 4.11**  
The output device  
interrupts the computer  
when it is idle and needs  
new data.



Again, we can visualize the interrupt synchronization by drawing a state versus time plot of the activities of the hardware and the two software modules. For an output device interface, the output device is initially disarmed and the FIFO is empty. The main thread begins by generating new data. After the main program puts the data into the FIFO, it arms the output interrupts. This first interrupt occurs immediately, and the ISR gets some data from the FIFO and outputs it to the external device. The output device becomes busy because it is in the process of outputting data. It is important to realize that it only takes the software on the order of 1  $\mu$ s to write data to one of its output ports, but usually it takes the output device much longer to fully process the data. When the output device is done, it is ready to accept more data and an interrupt is requested. If the FIFO is empty at this point, the ISR

**Figure 4.12**

Hardware/software timing of a CPU-bound output interface.

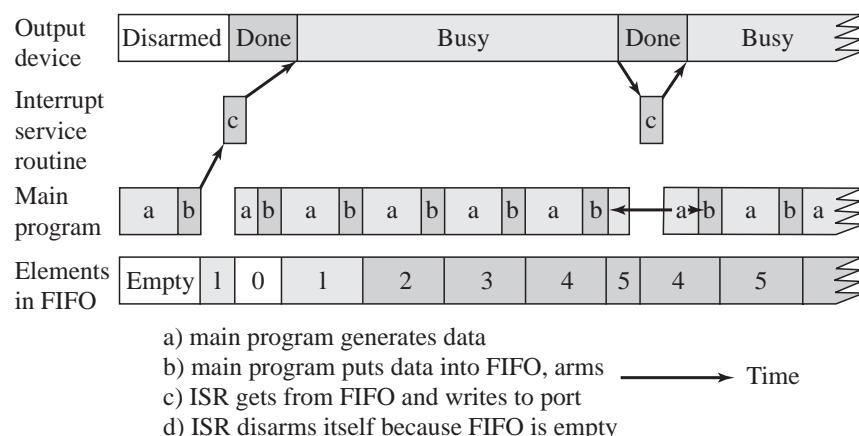


will disarm the output device. If the FIFO is not empty, the ISR will get data from the FIFO and write them out to the output port. Once data are written to the output port, the output device is released to go on. In this first example, the time for the software to generate data is longer than the time for the external device to output it. This is an example of a CPU-bound system (Figure 4.12). In this situation, the FIFO has either 0 or 1 entry, and the use of interrupts does not enhance the bandwidth over the busy-wait implementations presented in Chapter 3. Nevertheless, interrupts provide a well-defined mechanism for dealing with complex systems.

In this second example, the software starts with a burst of high-bandwidth activity (Figure 4.13). As long as the FIFO does not become full, no data are lost. In this situation, the overall bandwidth is higher than it would be with a busy-wait implementation, because the software does not have to wait for each data byte to be processed by the hardware. This is the classic example of a *buffered* output, because data enter the system (via the main program), are temporarily stored in a buffer (put into the FIFO), and are processed later (by the ISR, get from the FIFO, write to external device). When the I/O device is slower than the software, the system is called I/O bound. Just like the input scenario, the FIFO might become full if the producer rate is too high for too long.

**Figure 4.13**

Hardware/software timing of an I/O-bound output interface.



There are other types of interrupts that are not an input or an output. For example, we will configure the computer to request an interrupt on a periodic basis. This means an

interrupt handler will be executed at fixed time intervals. This periodic interrupt will be essential for the implementation of real-time data acquisition and real-time control systems. For example, if we are implementing a digital controller that executes a control algorithm 100 times a second, then we will set up the timer hardware to request an interrupt every 10 ms. The ISR will execute the digital control algorithm and return to the main thread. We will learn how to create the period interrupt in Section 4.15 and use it in Chapters 12 and 13.

An axiom with interrupt synchronization is that the interrupt program should execute as fast as possible. The interrupt should occur when it is time to perform a needed function, and the ISR should come in clean, perform that function, and return right away. Placing backward branches (busy-wait loops, iterations) in the interrupt software should be avoided if possible. The percentage of time spent executing interrupt software should be minimized. For an input device, the *interface latency* is the time between when new input are available and the time when the software reads the input data. We can also define *device latency* as the response time of the external I/O device. For example, if we request that a certain sector be read from a disk, then the device latency is the time it takes to find the correct track and spin the disk (seek) so that the proper sector is positioned under the read head. For an output device, the interface latency is the time between when the output device is idle and the time when the software writes new data. A *real-time* system is one that can guarantee an upper bound on the interface latency.

**Performance tip:** It is poor design to employ backward jumps in an ISR, because they may affect the latency of other interrupt requests. Whenever you are thinking about using a backward jump, consider redesigning the system with more or different triggers to reduce the number of backward jumps.

## 4.2 Reentrancy and Critical Sections

A program segment is reentrant if it can be concurrently executed by two (or more) threads. This issue is very important when using interrupt programming. To implement reentrant software, we place local variables on the stack and avoid storing into global memory variables. When writing in assembly, we use *registers* or the *stack* for parameter passing to create reentrant subroutines. Typically each thread will have its own set of registers and stack. A nonreentrant subroutine will have a section of code called a *vulnerable window* or *critical section*. An error occurs if:

1. One thread calls the nonreentrant subroutine
2. That thread is executing in the “vulnerable” window when interrupted by a second thread
3. The second thread calls the same subroutine

A number of scenarios can happen next. In the first scenario, the second thread is allowed to complete the execution of the subroutine, control is then returned to the first thread, and the first thread finishes the subroutine. This first scenario is the usual case with interrupt programming. In the second scenario, the second thread executes part of it, is interrupted and then reentered by a third thread, the third thread finishes, the control is returned to the second thread and it finishes, and last the control is returned to the first thread and it finishes. This second scenario can happen in interrupt programming if interrupts are reenabled during the execution of the ISR. In the third scenario, the second thread executes part of it, is interrupted and the first thread continues, the first thread finishes, control is returned to the second thread and it finishes. This third scenario will occur only when you use a thread scheduler like the ones described in the next chapter. Since most embedded systems do not have a thread scheduler, we will focus on the first two scenarios. A vulnerable

window may exist when two different subroutines access and modify the same memory-resident data structure.

In Figure 4.7, we saw two concurrent threads communicating via a FIFO queue. When processing input, it is possible for the main program to start to execute `Fifo_Get`, be interrupted, and the input interrupt routine calls `Fifo_Put`. To verify correctness of our system, we must consider what would happen if the `Fifo_Put` subroutine is executed in between any two assembly instructions of the `Fifo_Get` routine. A similar problem arises when processing outputs. In this situation the main program may start to execute `Fifo_Put`, be interrupted, and the output interrupt routine calls `Fifo_Get`. To verify correctness of this system, we must consider what would happen if the `Fifo_Get` subroutine is executed in between any two assembly instructions of the `Fifo_Put` routine. If we are processing both input and output, then two FIFOs would be used, so none of the individual functions can be reentered. Nevertheless, the `Fifo_Put` and `Fifo_Get` routines share access to global variables, so we must search for potential critical sections. The three functions `Fifo_Init`, `Fifo_Put` and `Fifo_Get` all manipulate the same global variables; therefore, to operate properly we need a mechanism to implement mutual exclusion, which simply means only one thread at a time is allowed access to the FIFO module. For most of this book, we will implement mutual exclusion by disabling interrupts. In the next chapter, we will describe how to create a preemptive thread scheduler and use semaphores to implement mutual exclusion.

This first example (Program 4.1) is a nonreentrant assembly subroutine that uses a memory variable `Second`. This subroutine could have been made reentrant by implementing `Second` as a local variable, but the purpose of the example is to illustrate what can go wrong when a nonreentrant subroutine is reentered.

#### Program 4.1

This subroutine is nonreentrant because of the read-modify-write access to a global.

```

Second rmb 2      Temporary global variable
* Input parameters: Reg X,Y contain 2 16 bit numbers
* Output parameter: Reg X is returned with the average
Ave    sty Second Save the second number in memory
       xgdx      Reg D contains first number
       addd Second Reg D=First+Second
       lsrd      (First+Second)/2
       adcb #0   round up?
       adca #0
       xgdx
       rts

```

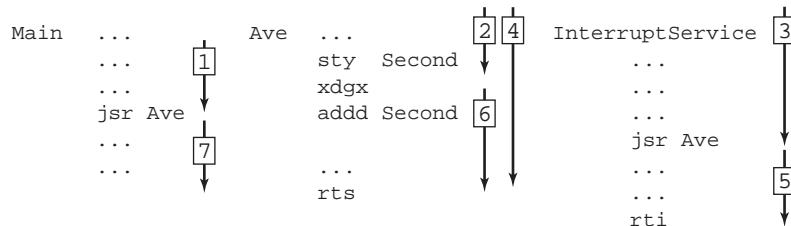
A *vulnerable window* exists between the `sty` and the `addd` instructions. Assume there are two concurrent threads (`Main`, `InterruptService`) that both call this subroutine. Concurrent means that both threads are ready to run. Because there is only one computer, exactly one thread will be active (running) at a time. Typically, the operating system (OS) switches execution control back and forth using interrupts. For example, the `Main` thread might be executing when an interrupt causes the computer to switch over and execute the `InterruptService`. When the `InterruptService` is done, it executes a `rti`, and the control returns back to the `Main` thread. An error occurs if (Figure 4.14):

1. The `Main` thread calls `Ave`
2. The `Main` thread executes the `sty` instruction, saving its second number in `Second`

3. The OS halts the Main thread (using an interrupt) and starts the `InterruptService` thread
  4. The `InterruptService` thread calls `Ave`
- The `InterruptService` thread executes the `sty`, saving its second number in `Second`
- The `InterruptService` thread finishes `Ave`
5. The OS returns control back to the Main thread
  6. The Main thread executes the `addd` instruction but gets the wrong `Second` number

**Figure 4.14**

One sequence of operations that results in the reentering of the subroutine `Ave`.



A *vulnerable window* exists in the following program between the `Result=y` and the `Result+=x` instructions.

```

int Result; /* Temporary global variable */
int Ave(int x,y){
    Result=y; /* Save the second number in global memory */
    Result=(Result+x)>>1; /* (First+Second)/2 */
    return(Result);
}

```

An error occurs if (Figure 4.15):

1. The Main thread calls `Ave`
2. The Main thread executes the `Result=y` instruction, saving its second number in `Result`
3. The OS halts the Main thread (using an interrupt) and starts the `InterruptService` thread
4. The `InterruptService` thread calls `Ave`

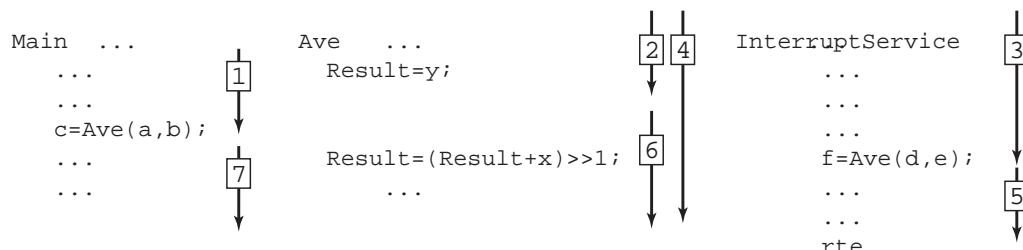
The `InterruptService` thread executes the `Result=y`, saving its number over the other copy

The `InterruptService` thread finishes `Ave`

5. The OS returns control back to the Main thread
6. The Main thread calculates `(Result+x)>>1` but gets the wrong `Result` number

The solution to this simple problem is to implement `Result` as a local variable either in a register or on the stack. But we saw earlier in the chapter that global variables are needed for interthread communication, so not all nonreentrant activity can be solved by using local variables. In this chapter we will disable interrupts during the vulnerable window. In the next chapter semaphores are used to provide mutual exclusive access to certain memory-resident data structures.

An *atomic operation* is one that once started is guaranteed to finish. In most computers, once an instruction has begun, the instruction must be finished before the computer can



**Figure 4.15**

One sequence of operations that results in the reentering of the C function Axe.

process an interrupt. Therefore, the following read-modify-write sequence is atomic because it cannot be halted in the middle of its operation:

inc counter where counter is a global variable

On the other hand, this read-modify-write sequence is *nonatomic* because it can start, then be interrupted:

```
ldaa counter      where counter is a global variable  
inca  
staa counter
```

In general, critical sections can be grouped into three categories, all involving nonatomic writes to shared global variables. We will classify I/O ports as global variables for the consideration of critical sections. We will group registers into the same category as local variables because each thread will have its own registers and stack.

The first group is the *read-modify-write* sequence:

1. The software reads the global variable, producing a copy of the data.
  2. The software modifies the copy (at this point the original variable is still unmodified).
  3. The software writes the modification back into the global variable.

An assembly example is shown in Program 4.2. In C, this read-modify-write could be implemented as shown in Program 4.3.

## Program 4.2

This subroutine is also nonreentrant because of the read-modify-write access to a global

```
Money rmb 2      bank balance implemented as a global
* add $100 to the account
more ldd Money where Money is a global variable
      addd #100
      std Money Money=Money+100
      rts
```

### Program 4-3

**Program 1.5**  
This C function is nonreentrant because of the read-modify-write access to a global.

```
unsigned int Money; /* bank balance implemented as a global */
/* add 100 dollars */
void more(void){
    Money += 100;}
```

In the second group is the *write followed by read*, where the global variable is used for temporary storage:

1. The software writes to the global variable (this becomes the only copy of important information).
  2. The software reads from the global variable, expecting the original data to still be there.

An assembly example is shown in Program 4.4. In C, Program 4.5 shows an illustrative example of this write-read.

#### **Program 4.4**

This assembly subroutine is nonreentrant because of the write-read access to a global.

```
temp rmb 2      temporary result implemented as a global
* calculate RegX=RegX+2*RegD
mac  stx temp   Save X so that it can be added
     lsl d        RegD=2*RegD
     add d temp   RegD=RegX+2*RegD
     xgd x        RegX=RegX+2*RegD
     rts
```

#### **Program 4.5**

This C function is nonreentrant because of the write-read access to a global.

```
int temp; /* global temporary */
/* calculate x+2*d */
int mac(int x, int d){
    temp = x+2*d; /* write to a global variable */
    return (temp); /* read from global */
```

In the third group, we have a *nonatomic multistep write* to a global variable:

1. The software writes part of the new value to a global variable.
2. The software writes the rest of the new value to a global variable.

Program 4.6 shows an assembly example, and in C, this multistep write sequence could be implemented as shown in Program 4.7.

#### **Program 4.6**

This assembly subroutine is nonreentrant because of the multistep write access to a global.

```
Info rmb 4      32-bit data implemented as a global
* set the variable using RegX and RegY
set  stx Info   Info is a 32 bit global variable
     sty Info+2
     rts
```

#### **Program 4.7**

This C function is nonreentrant because of the multistep write access to a global.

```
int info[2]; /* 32-bit global */
void set(int x, int y){
    info[0]=x;
    info[1]=y;
```

**Observation:** When considering reentrant software and vulnerable windows, we classify accesses to I/O ports the same as accesses to global variables.

#### **We can make a subroutine reentrant by using a stack variable**

If we can eliminate the global variables, then the subroutine becomes reentrant. The example of Program 4.1 is redesigned by placing the temporary on the stack, as shown in Program 4.8. There are no “vulnerable” windows because each thread has its own registers and stack.

#### **We can make a subroutine reentrant by disabling interrupts**

Sometimes one must access global memory to implement the desired function. Consider the example of a message mailbox (Program 4.9). This mailbox has a status flag (0 means empty)

and an 8-bit value. The subroutine send will check the status, and if empty it will store the message (passed in RegB) into the mailbox. In this example there are multiple concurrent threads that may call Send. The mailbox will be covered in more detail in Chapter 5.

### Program 4.8

This assembly subroutine is reentrant because it does not write to any globals.

```
* Input parameters: Reg X,Y contain 2 16 bit numbers
* Output parameter: Reg X is returned with the average
Ave    pshy      Save the second number on the stack
       tsy       Reg Y points the Second number
       xgdx     Reg D contains first number
       addd 0,Y   Reg D=First+Second
       lsrd      (First+Second)/2
       adcb #0    round up?
       adca #0
       xgdx
       puly
       rts
```

### Program 4.9

This assembly subroutine is nonreentrant because of the read-modify-write access to a global.

```
Status  rmb 1    0 means empty, -1 means it contains something
Message rmb 1    data to be communicated
* Input parameter: Reg B contains an 8 bit message
* Output parameter: Reg CC (C bit) is 1 for OK, 0 for busy error
Send    tst Status  check if mailbox is empty
        bmi Busy    full, can't store, so return with C=0
        stab Message store
        dec Status  signify it now contains a message
        sec      stored OK, so return with C=1
Busy    rts
```

Clearly there is a *vulnerable window* or *critical section* between the `tst` and `dec` instructions that makes this subroutine nonreentrant. One can make this subroutine reentrant by disabling interrupts during the vulnerable window. It is important not to disable interrupts too long so as not to affect the interface latency of the other threads. Notice also that the interrupts are not simply disabled then enabled, but rather the interrupt status is saved, the interrupts disabled, then the interrupt status is restored. Consider what would happen if you simply added a `sei` at the beginning and a `cli` at the end of the above subroutine, then called it with the interrupts disabled.

### Program 4.10

This assembly subroutine is reentrant because it disables interrupts during the critical section.

```
Status  rmb 1    0 means empty, -1 means it contains something
Message rmb 1    data to be communicated
* Input parameter: Reg B contains an 8 bit message
* Output parameter: Reg CC (C bit) is 1 for OK, 0 for busy error
Send    clc      Initialize carry=0
        tpa      save current interrupt state
        psha
        sei      disable interrupts during vulnerable window
        tst Status  check if mailbox is empty
        bmi Busy    full, so return with C=0
        staa Message store
        dec Status  signify it now contains a message
        pula
        ora#1    OK, so return with C=1
        psha
Busy    pula     restore interrupt status
        tap
        rts
```

**Program 4.11**

This C function is reentrant because it disables interrupts during the critical section.

```
int Empty;           // -1 means empty, 0 means it contains something
short Message;      // data to be communicated
int Send(short data){ int OK;
unsigned char SaveCCR;
    asm tpa          // previous interrupt enable
    asm staa SaveCCR // save previous
    asm sei          // make atomic, start critical section
    OK = 0;          // Assume it is not OK
    if(Empty){
        Message = data;
        Empty = 0;    // signify it is now contains a message
        OK = -1;      // Successfull
    }
    asm ldaa SaveCCR // recall previous
    asm tap          // end critical section
    return(OK);
}
```

Some machines provide a *test and set* function to solve this reentrant problem. This single (nondivisible) operation is equivalent to the subroutine in Program 4.12. The test and set operation, once started, must be allowed to complete. More details about semaphores can be found in Chapter 5.

**Program 4.12**

This assembly subroutine can be used as part of a binary semaphore.

```
* Global parameter: Semi4 is the memory location to test and set
* If the location is zero, it will set it (make it -1)
*       and return Reg CC (Z bit) is 1 for OK
* If the location is nonzero, it will return Reg CC (Z bit) = 0
Semi4 fcb 0           Semaphore is initially free
Tas   tst  Semi4      check if already set
        bne  Out       busy, operation failed, so return with z=0
        dec   Semi4     signify it is now busy
        bita #0         operation successful, so return with z=1
Out   rts
```

Critical sections are very important when writing high-level language software, too. Obviously, we minimize the use of global variables. But when global variables are necessary, we must be able to recognize potential sources of bugs due to nonreentrant code. We must study the assembly language output produced by the compiler. For example, we cannot determine whether the following read-modify-write operation is critical without knowing if it is atomic:

```
time++;
```

If the compiler generates the following object code, then `time++;` is atomic (therefore not critical):

```
inc time
```

If the compiler generates the following object code, then `time++;` is not atomic (therefore critical):

```
ldd  time
addd #1
std  time
```

Similarly, it is possible that your compiler might generate noncritical code for Program 4.5, so we must always examine assembly listings when considering reentrancy.

## 4.3 First-In–First-Out Queue

### 4.3.1 Introduction to FIFOs

As we saw earlier, the FIFO circular queue is quite useful for implementing a buffered I/O interface. It can be used for both buffered input and buffered output. The order-preserving data structure temporarily saves data created by the source (producer) before they are processed by the sink (consumer). The class of FIFOs studied in this section will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be carefully shared by more than one thread. The advantage of using a FIFO structure for a data flow problem is that we can decouple the producer and consumer threads. Without the FIFO we would have to produce one piece of data, then process it, produce another piece of data, then process it. With the FIFO, the producer thread can continue to produce data without having to wait for the consumer to finish processing the previous data. This decoupling can significantly improve system performance (Figure 4.16).

**Figure 4.16**

The FIFO is used to buffer data between the producer and consumer.



You have probably already experienced the convenience of FIFOs. For example, you can continue to type another command into the Windows command interpreter while it is still processing a previous command. The ASCII codes are put (calls *Fifo-Put*) in a FIFO whenever you hit the key. When the Windows command interpreter is free, it calls *Fifo-Get* for more keyboard data to process. A FIFO is also used when you ask the computer to print a file. Rather than waiting for the actual printing to occur character by character, the print command will *PUT* the data in a FIFO. Whenever the printer is free, it will *GET* data from the FIFO. The advantage of the FIFO is it allows you to continue to use your computer while the printing occurs in the background. To implement this magic of background printing we will need interrupts.

There are many producer/consumer applications. In Table 4.2 the tasks on the left are producers that create or input data, while the tasks on the right are consumers that process or output data.

**Table 4.2**

Producer-consumer examples.

Source/Producer		Sink/Consumer
Keyboard input	→	Program that interprets
Program with data	→	Printer output
Program sends message	→	Program receives message
Microphone and ADC	→	Program that saves sound data
Program that has sound data	→	DAC and speaker

The producer puts data into the FIFO. The `Fifo_Put` operation does not discard information already in the FIFO. If the FIFO is full and the user calls `Fifo_Put`, the `Fifo_Put` routine will return a full error, signifying the last (newest) data were not properly saved. The

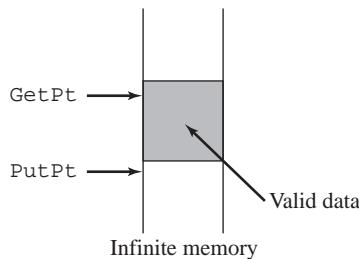
sink process removes data from the FIFO. The `Fifo_Get` routine will modify the FIFO. After a get, the particular information returned from the `Fifo_Get` routine is no longer saved on the FIFO. If the FIFO is empty and the user tries to get, the `Fifo_Get` routine will return an empty error, signifying no data could be retrieved. The FIFO is order-preserving, such that the information is returned by repeated calls of `Fifo_Get` in the same order as the data were saved by repeated calls of `Fifo_Put`.

There are many ways to implement a statically allocated FIFO. We can use either a pointer or an index to access the data in the FIFO. We can use either two pointers (or two indices) or two pointers (or two indices) and a counter. The counter specifies how many entries are currently stored in the FIFO. There are even hardware implementations of FIFO queues. We begin with the two-pointer implementation. It is a little harder to implement but does have some advantages over the other implementations.

### 4.3.2 Two-Pointer FIFO Implementation

We will begin with the two-pointer implementation. If we were to have infinite memory, a FIFO implementation is easy (Figure 4.17). `GetPt` points to the data that will be removed by the next call to `Fifo_Get`, and `PutPt` points to the empty space where the data will be stored by the next call to `Fifo_Put` (Program 4.13).

**Figure 4.17**  
The FIFO implementation with infinite memory.



```

GetPt rmb 2 ;Pointer to oldest data
PutPt rmb 2 ;Pointer to free memory
* Call by value with RegA
Fifo_Put ldx PutPt ;place to put next
    staa 1,x+ ;Store data into FIFO
    stx PutPt ;Update pointer
    ldaa #-1 ;success
    rts
* Call by reference with RegX
Fifo_Get ldy GetPt ;place to remove next
    ldaa 1,y+ ;Read data from FIFO
    staa 0,x ;return by reference
    sty GetPt ;Update pointer
    ldaa #-1 ;success
    rts
  
```

```

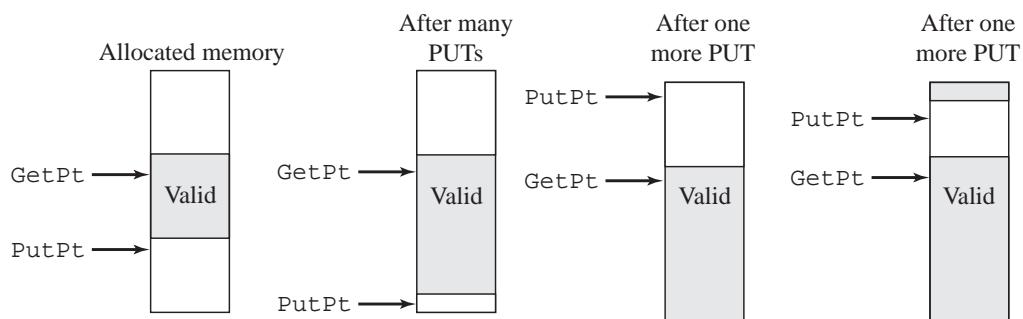
char static volatile *PutPt; // put next
char static volatile *GetPt; // get next
// call by value
int Fifo_Put(char data){
    *PutPt = data; // Put
    PutPt++;
    return(1); // true if success
}
// call by reference
int Fifo_Get(char *datapt){
    *datapt = *GetPt; // return by reference
    GetPt++; // next
    return(1); // true if success
}
  
```

### Program 4.13

Code fragments showing the basic idea of a FIFO.

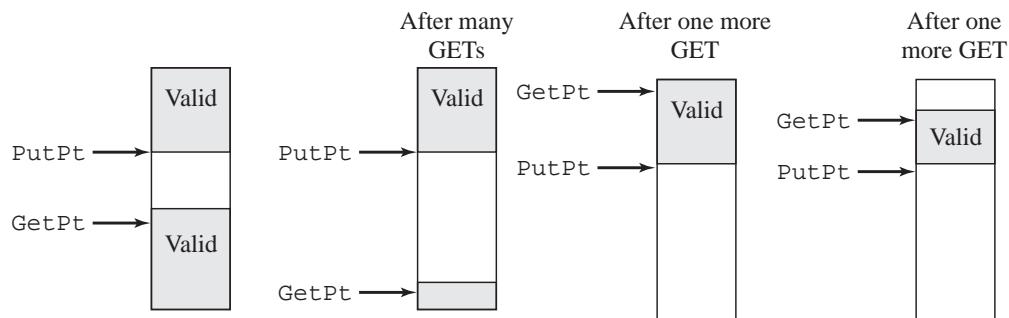
Three modifications are required to the above subroutines. If the FIFO is full when `Fifo_Put` is called, then the subroutine should return a full error. Similarly, if the FIFO is empty when `Fifo_Get` is called, then the subroutine should return an empty error.

The `PutPt` and `GetPt` must be wrapped back up to the top when they reach the bottom (Figures 4.18, 4.19).



**Figure 4.18**

The `Fifo_Put` operation showing the pointer wrap.



**Figure 4.19**

The `Fifo_Get` operation showing the pointer wrap.

Two mechanisms can be used to determine whether the FIFO is empty or full. A simple method is to implement a counter containing the number of bytes currently stored in the FIFO. `Fifo_Get` would decrement the counter, and `Fifo_Put` would increment the counter. The second method is to prevent the FIFO from being completely full. For example, if the FIFO had 10 bytes allocated, then the `Fifo_Put` subroutine would allow a maximum of 9 bytes to be stored. If there were 9 bytes already in the FIFO and another `Fifo_Put` were called, then the FIFO would not be modified and a full error would be returned. In this way if `PutPt` equals `GetPt` at the beginning of `Fifo_Get`, then the FIFO is empty. Similarly, if `PutPt+1` equals `GetPt` at the beginning of `Fifo_Put`, then the FIFO is full. Be careful to wrap the `PutPt+1` before comparing it to `Fifo_Get`. This second method does not require the length to be stored or calculated. The implementation of this FIFO module is shown in Program 4.14.

To check for FIFO full, the `Fifo_Put` routine in Program 4.14 attempts to put using a temporary `PutPt`. If putting makes the FIFO look empty, then the temporary `PutPt` is discarded and the routine is exited without saving the data. This is why a FIFO with 10 allocated bytes can hold only 9 data points. If putting doesn't make the FIFO look empty, then the temporary `PutPt` is stored into the actual `PutPt`, saving the data as desired.

```

;Two-pointer implementation of the FIFO
FIFOSIZE equ 10 ;can hold 9 elements
PutPt rmb 2 ;where to put next
GetPt rmb 2 ;where to get next
Fifo rmb FIFOSIZE
Fifo_Init ldx #Fifo
    tpa ;make atomic
    sei ;critical section
    stx PutPt
    stx GetPt ;Empty
    tap ;end critical section
    rts
*****Put a byte into the FIFO*****
;Input RegA is 8-bit data to put
;Output RegA is -1 if ok, 0 if full
Fifo_Put ldx PutPt ;Temporary
    staa 1,x+ ;Try to put data
    cpx #Fifo+FIFOSIZE
    bne PutNoWrap
    ldx #Fifo ;Wrap
PutNoWrap clra
    cpx GetPt ;Full if same
    beq PutDone
    coma ;-1 means OK
    stx PutPt
PutDone rts
*****Get a byte from the FIFO*****
; Input RegX place for 8-bit data
; Output RegA is -1 if ok, 0 if empty
Fifo_Get clra ;assume it will fail
    ldy GetPt
    cpy PutPt ;Empty if the same
    beq GetDone
    coma ;RegA=-1 means OK
    ldab 1,y+ ;Data from FIFO
    stab 0,x ;Return by reference
    cpy #Fifo+FIFOSIZE
    bne GetNoWrap
    ldy #Fifo ;Wrap
GetNoWrap sty GetPt
GetDone rts

```

```

// Two-pointer implementation of the FIFO
#define FIFOSIZE 10 // can hold 9
char static volatile *PutPt; // put next
char static volatile *GetPt; // get next
char static Fifo[FIFOSIZE];
void Fifo_Init(void){
unsigned char SaveCCR;
asm tpa
asm staa SaveCCR
asm sei // make atomic
PutPt=GetPt=&Fifo[0]; // Empty
asm ldaa SaveCCR
asm tap // end critical section
}
int Fifo_Put(char data){
char volatile *tempPt;
tempPt = PutPt;
*(tempPt++) = data; // try to Put
if(tempPt==&Fifo[FIFOSIZE]){
    tempPt = &Fifo[0]; // wrap
}
if(tempPt == GetPt ){
    return(0); // Failed, fifo full
}
else{
    PutPt = tempPt; // Success, update
    return(1);
}
}
int Fifo_Get(char *datapt){
if(PutPt == GetPt ){
    return(0); // Empty if PutPt=GetPt
}
else{
    *datapt = *(GetPt++);
    if(GetPt==&Fifo[FIFOSIZE]){
        GetPt = &Fifo[0]; // wrap
    }
    return(1);
}
}

```

### Program 4.14

Two-pointer implementation of a FIFO.

To check for FIFO empty, the `Fifo_Get` routine in Program 4.14 simply checks to see if `GetPt` equals `PutPt`. If they match at the start of the routine, then `Fifo_Get` returns with the “empty” condition signified.

Since `Fifo_Put` and `Fifo_Get` have read-modify-write accesses to global variables, they themselves are not reentrant. Similarly, `Fifo_Init` has a multiple-step write-access to global variables. Therefore `Fifo_Init` is not reentrant. Consequently, interrupts could be temporarily disabled to prevent one thread from reentering these FIFO functions. Notice that at the end of the critical section, interrupts are not enabled, but rather the interrupt status is restored to its previous state (enabled or disabled). This method of disabling interrupts allows you to nest a function call from one nonreentrant function to another.

One advantage of this pointer implementation is that if you have a single thread that calls the `Fifo_Get` (e.g., the main program) and a single thread that calls the `Fifo_Put` (e.g., the serial port receive interrupt handler) as shown in Figure 4.7, then this `Fifo_Put` function can interrupt this `Fifo_Get` function without loss of data. So in this particular situation, interrupts would not have to be disabled. It would also operate properly if there were a single interrupt thread calling `Fifo_Get` (e.g., the serial port transmit interrupt handler) and a single thread calling `Fifo_Put` (e.g., the main program). On the other hand, if the situation is more general, and multiple threads could call `Fifo_Put` or multiple threads could call `Fifo_Get`, then the interrupts would have to be temporarily disabled.

- 4.3.3 Two-Pointer/Counter FIFO Implementation** The other method to determine if a FIFO is empty or full is to implement a counter. In the code of Program 4.15, `Size` contains the number of bytes currently stored in the FIFO. The advantage of implementing the counter is that FIFO quarter-full and three-quarter-full conditions are easier to implement. If you were studying the behavior of a system, it might be informative to measure the values of `Size` as a function of time.

<pre>; Pointer, counter implementation FIFOSIZE equ 10 ;can hold 10 PutPt    rmb 2 ;where to put next GetPt    rmb 2 ;where to get next Size     rmb 1 ;empty if Size=0 Fifo     rmb FIFOSIZE Fifo_Init ldx #Fifo           tpa      ;make atomic           sei      ;critical section           stx PutPt           stx GetPt ;Empty           clr Size           tap      ;end critical section           rts *****Put a byte into the FIFO***** * Input RegA contains 8-bit data to put * Output RegA is -1 if ok, 0 if full Fifo_Put pshc      ;save old CCR           sei      ;make atomic           ldab Size           cmpb #FIFOSIZE ;Full ?           bne PNotFull           clra           bra PutDone PNotFull inc Size ;one more element           ldx PutPt           staa 1,x+ ;Put data into fifo           cpx #Fifo+FIFOSIZE           bne PutNoWrap           ldx #Fifo ;Wrap PutNoWrap ldaa #-1 ;success means OK           stx PutPt PutDone   pulc      ;end critical section           rts</pre>	<pre>// Pointer, counter implementation #define FIFOSIZE 10 // can hold 10 char static volatile *PutPt; // put next char static volatile *GetPt; // get next char static Fifo[FIFOSIZE]; unsigned char Size; // Number of elements void Fifo_Init(void){ unsigned char SaveCCR; asm tpa asm staa SaveCCR asm sei           // make atomic PutPt=GetPt=&amp;Fifo[0]; // Empty Size = 0; asm ldaa SaveCCR asm tap        // end critical section } int Fifo_Put(char data){ unsigned char SaveCCR; if(Size == FIFOSIZE){ return(0); // fifo was full } else{ asm tpa asm staa SaveCCR asm sei           // make atomic Size++; // one more element in FIFO *(PutPt++) = data; // put data if(PutPt == &amp;Fifo[FIFOSIZE]){ PutPt = &amp;Fifo[0]; // Wrap } asm ldaa SaveCCR asm tap        // end critical section return(-1); // Successful }</pre>
---	---

```
*****Get a byte from the FIFO*****
* Input RegX points to place for data
* Output RegA is -1 if ok, 0 if empty
Fifo_Get pshc    ;save old CCR
          sei     ;critical section
          clra   ;assume it will fail
          tst   Size
          beq   GetDone
          dec   Size ;one less element
          ldy   GetPt
          coma   ;RegA=-1 means OK
          ldab 1,y+ ;Data from FIFO
          stab 0,x ;Return by reference
          cpy   #Fifo+FIFOSIZE
          bne   GetNoWrap
          ldy   #Fifo ;Wrap
GetNoWrap sty   GetPt
GetDone   pulc   ;end critical section
          rts
```

```
int Fifo_Get(char *datapt) { char SaveSP;
unsigned char SaveCCR;
if(Size == 0){
    return(0); // Empty if Size=0
}
else{
asm tpa
asm staa SaveCCR
asm sei           // make atomic
*datapt=*(GetPt++);
Size--; // one less element in FIFO
if(GetPt == &Fifo[FifoSize]){
    GetPt = &Fifo[0]; // wrap
}
asm ldaa SaveCCR
asm tap           // end critical section
return(-1); // Successful
}
```

### Program 4.15

Implementation of a two-pointer with counter FIFO.

To check for FIFO full, the `Fifo_Put` routine in Program 4.15 simply compares `Size` to the maximum allowed value. If the FIFO is already full, then the routine is exited without saving the data. With this implementation a FIFO with 10 allocated bytes actually can hold 10 data points.

To check for FIFO empty, the `Fifo_Get` routine of Program 4.15 simply checks to see if `Size` equals 0. If `Size` is zero at the start of the routine, then `Fifo_Get` returns with the “empty” condition signified.

### 4.3.4 FIFO Dynamics

As you recall, the FIFO passes the data from the producer to the consumer. In general, the rates at which data are produced and consumed can vary dynamically. Human beings do not enter data into a keyboard at a constant rate. Even printers require more time to print color graphics versus black and white text. Let  $t_p$  be the time (in seconds) between calls to `Fifo_Put` and  $r_p$  be the arrival rate (producer rate in bytes per second) into the system. Similarly, let  $t_g$  be the time (in seconds) between calls to `Fifo_Get` and  $r_g$  be the service rate (consumer rate in bytes per second) out of the system.

$$r_g = \frac{1}{t_g} \quad r_p = \frac{1}{t_p}$$

If the minimum time between calls to `Fifo_Put` is greater than the maximum time between calls to `Fifo_Get`,

$$\min t_p \quad \max t_g$$

then a FIFO is not necessary and the data flow program could be solved with a mailbox. On the other hand, if the time between calls to `Fifo_Put` becomes less than the time between calls to `Fifo_Get` because either:

- The arrival rate temporarily increases
- The service rate temporarily decreases

then information will be collected in the FIFO. For example, a person might type very fast for a while, followed by a long pause. The FIFO could be used to capture without loss all

the data as they come in very fast. Clearly on average the system must be able to process the data (the consumer thread) at least as fast as the average rate at which the data arrives (producer thread). If the average producer rate is larger than the average rate the consumer is capable of handling data,

$$\bar{r}_p > \bar{r}_g$$

then the FIFO will eventually overflow no matter how large the FIFO. If the producer rate is temporarily high and that causes the FIFO to become full, then this problem can be solved by increasing the FIFO size.

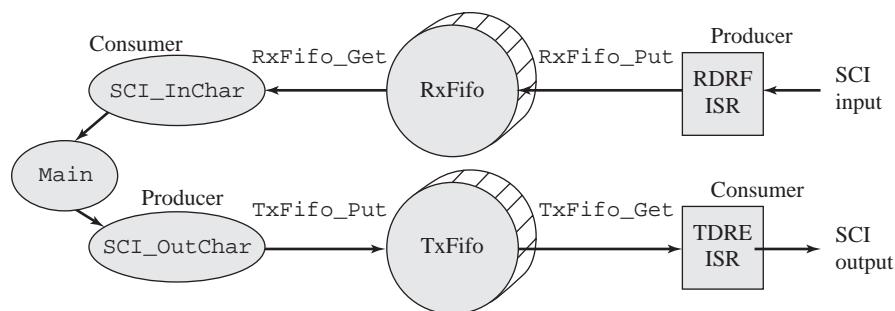
There is a fundamental difference between an empty error and a full error. Consider the application of using a FIFO between your computer and its printer. This is a good idea because the computer can temporarily generate data to be printed at a very high rate, followed by long pauses. The printer is like a turtle. It can print at a slow but steady rate. The computer will put a byte into the FIFO that it wants printed. The printer will get a byte out of the FIFO when it is ready to print another character. A full error occurs when the computer calls `Fifo_Put` at too fast a rate. A full error is serious, because if ignored data will be lost. On the other hand, an empty error occurs when the printer is ready to print but the computer has nothing in mind. An empty error is not serious, because in this case the printer just sits there doing nothing.

**Checkpoint 4.2:** If the FIFO becomes full, can the situation be solved by increasing the size?

Figure 4.20 shows a data flow graph with buffered input and buffered output. FIFOs implemented in this section are statically allocated global structures. The system shown in Figure 4.20 has two channels, one for input and one for output, and each channel employs a separate FIFO queue. An assembly language implementation of this system can be found as TUT4 within TExaS, and a C implementation as project SCIA on the CD. The details of this implementation will be presented in Chapter 7.

**Figure 4.20**

A data flow graph showing two FIFOs that buffer data between producers and consumers.



**Checkpoint 4.3:** What does it mean if the RxFifo in Figure 4.20 is empty?

**Checkpoint 4.4:** What does it mean if the TxFifo in Figure 4.20 is empty?

## 4.4 General Features of Interrupts on the 9S12

In this section we will present the specific details for the 9S12. As you develop experience using interrupts, you will come to notice a few common aspects that most computers share. The following paragraphs outline three essential mechanisms that are needed to utilize interrupts. Although every computer that uses interrupts includes all three mechanisms, there is a wide spectrum of implementation methods.

All interrupting systems have the *ability for the hardware to request action from the computer*. The interrupt requests can be generated by using a separate connection to the microprocessor for each device or by using a negative-logic wire-or requests employing open-collector logic. The Freescale microcomputers support both types.

All interrupting systems must have the *ability for the computer to determine the source*. A vectored interrupt system employs separate connections for each device so that the computer can give automatic resolution. You can recognize a vectored system because each device has a separate interrupt vector address. With a polled interrupt system, the software must poll each device, looking for the device that requested the interrupt.

The third necessary component of the interface is the *ability for the computer to acknowledge the interrupt*. Normally there is a flag in the interface that is set on the busy-to-done state transition. Acknowledging the interrupt involves clearing the flag that caused the interrupt. It is important to shut off the request so that the computer will not mistakenly request a second (and inappropriate) interrupt for the same condition. Some Intel systems use a hardware acknowledgment that automatically clears the request. Most Freescale microcomputers use a software acknowledge. So when we identify the flag, it will be important to know exactly what conditions will set the flag (and request an interrupt) and how the software will clear it (acknowledge) in the ISR.

Even though there are no standard definitions for the terms *mask*, *enable*, and *arm* in the computer science or computer engineering communities, in this book we will assign the following specific meanings. To *arm* (*disarm*) a device means to enable (shut off) the source of interrupts. One arms (disarms) a device if one is (is not) interested in interrupts at all. For example, the TOI bit is the arm bit for the TOF flag. Similarly the 9S12 has eight arm bits (C7I–COI) for the output compare and input capture interrupts. The Freescale literature calls the arm bit an “interrupt enable mask.” To *enable* (*disable*) means to allow interrupts at this time (postpone interrupts until a later time). We disable interrupts if it is currently not convenient to process. In particular, to disable interrupts, we set the I bit in the condition code register using the `sei` instruction. There are some interrupts that cannot be disabled, such as XIRQ, SWI, illegal opcode, reset. For most of this book (except in Chapter 5) we will disable interrupts while executing in a vulnerable window.

**Common error:** The system will crash if the ISR does not either acknowledge or disarm the device requesting the interrupt.

**Common error:** The ISR software does not have to explicitly disable interrupts at the beginning (`sei`) or explicitly reenable interrupts at the end (`cli`).

The sequence of events that occurs during an interrupt service is quite similar on the 9S12. The sequence begins with the *Hardware needs service (busy-to-done) transition*. This signal is connected to an input of the microcomputer that can generate an interrupt. For example, key wake-up, input capture, SCI, and SPI systems support interrupt requests. Some interrupts are internally generated like output compare, real-time interrupt (RTI), and timer overflow.

The second event is the *setting of a flag* in one of the I/O status registers of the microcomputer. This is the same flag that a busy-wait interface would be polling on. Examples include key wake-up (e.g., `PIFP7`), input capture (`C3F`), SCI (`RDRF`), SPI (`SPIF`), output compare (`C5F`), RTI (`RTIF`), and timer overflow (`TOF`). For an interrupt to be requested the appropriate *flag* bit must be *armed*. Examples include key wake-up (e.g., `PIEP7`), input capture (`C3I`), SCI (`RIE`), SPI (`SPIE`), output compare (`C5I`), RTI (`RTII`), and timer overflow (`TOI`). The sequence order of these three conditions does not matter, as long as all three become true:

The interrupting event occurs that sets the flag  
The device is armed  
The microcomputer interrupts are enabled

TOF=1  
TOI=1  
I=0

The third event in the interrupt processing sequence is the *thread switch*. The thread switch is performed by the microcomputer hardware automatically. The specific steps include:

1. The microcomputer will finish the current instruction.<sup>2</sup>
2. All the registers are pushed on the stack (i.e., PC, Y, X, A, B, CCR) the CCR is on top, with the I bit still equal to 0.
3. The microcomputer will get vector address from memory and put it into the PC.
4. The microcomputer will set I = 1.

The fourth event is the software *execution of the ISR*. For a polled interrupt configuration the ISR must poll each possible device and branch to a specific handler for that device. The polling order establishes device priority. For a vectored interrupt configuration, you could poll anyway to check for run-time hardware/software errors. The ISR must either acknowledge or disarm the interrupt. We acknowledge an interrupt by clearing the flag that was set in the second event shown above. We will see later in the chapter that there is an optional step at this point for low-priority interrupt handlers. After we acknowledge a low-priority interrupt, we may reenable interrupts (c1i) to allow higher-priority devices to go first. All ISRs must perform the necessary operations (read data, write data, etc.) and pass parameters through global memory (e.g., FIFO queue).

The last event is another thread switch to *return control back to the thread that was running* when the interrupt was processed. In particular, the software executes a rti at the end of the ISR, which will pull all the registers off the stack (i.e., CCR, B, A, X, Y, PC). Since the CCR was pushed on the stack in step 2 above with I=0, the execution of rti automatically reenables interrupts. After the ISR executes rti, the stack is restored to the state it was before the interrupt (there may be one more or less entry in the FIFO however).

CPU12 exceptions include resets and interrupts. Each exception has an associated 16-bit vector that points to the memory location where the routine that handles the exception is located. Vectors are stored in the upper 128 bytes of the standard 64-kbyte address map.

A hardware priority hierarchy determines which reset or interrupt is serviced first when simultaneous requests are made. Six sources are not maskable. The remaining sources have a mask bit that can be enabled (armed) or turned off (disarmed). The priorities of the non-maskable sources are:

1. Power-on-reset (POR) or regular hardware RESET pin
2. Clock monitor reset
3. COP watchdog reset
4. Unimplemented instruction trap
5. Software interrupt instruction (swi)
6. XIRQ signal (if X bit in CCR = 0)

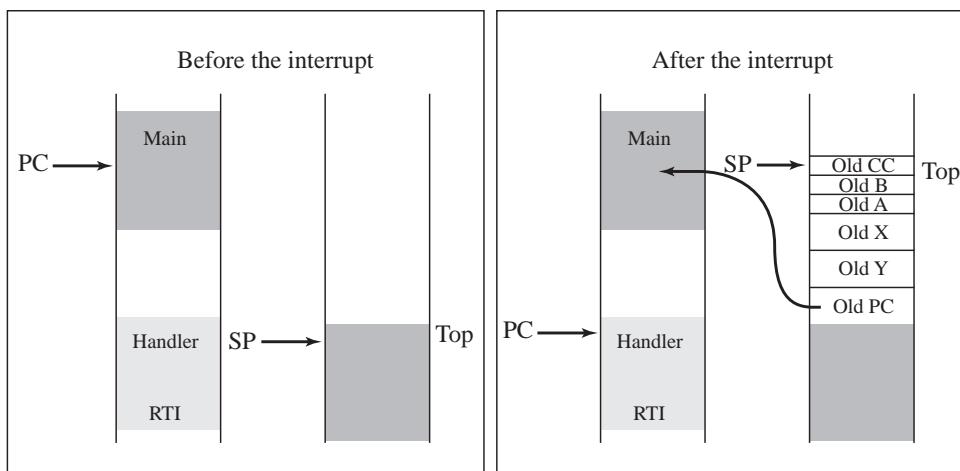
Maskable interrupt sources include on-chip peripheral systems and external interrupt service requests. Interrupts from these sources are recognized when the global interrupt enable bit (I) in the CCR is cleared. The default state of the I bit out of reset is 1, but it can be written at any time.

The 9S12 interrupt hardware also automatically saves all registers on the stack during the thread-switch (i.e., PC, Y, X, A, B, CCR). The “oldPC” value on the stack points to the place in the foreground thread to resume once the interrupt is complete. At the end of the interrupt handler, another thread switch occurs as the rti instruction restores registers from the stack (including the PC) (Figure 4.21).

**Checkpoint 4.5:** During the startup of an ISR, which happens first: the CCR is pushed on the stack, or the I bit is set?

---

<sup>2</sup>Again, the 9S12 allows the instructions rev, revw, and wav to be interrupted.

**Figure 4.21**

9S12 stack before and after an interrupt.

The 9S19 has two external requests,  $\overline{\text{IRQ}}$  and  $\overline{\text{XIRQ}}$ , that are level-zero active. Many of the internal I/O devices can generate interrupt requests based on external events (e.g., key wake-up, input capture, SCI, SPI). These interrupt requests will temporarily set the I bit in the CCR during the interrupt program to prevent other interrupts (including itself). On the other hand, the XIRQ request temporarily sets both the I and X bits in the CCR during the interrupt program to postpone all other interrupt sources. The 9S12 can support:

- Key wake-up interrupts
- Input capture/output compare interrupts
- ADC interrupts
- Timer interrupts (timer overflow, real time interrupt, pulse accumulator)
- Serial port interrupts (SCI, SPI, CAN and I2C)

For detailed information, you must refer to the data sheet for your specific microcontroller.

The interrupts have a fixed priority, but you can elevate one request to highest priority using the HPPIO register (\$001F). The relative priorities of the other interrupt sources remain the same.

We typically use XIRQ to interface a single highest-priority device. XIRQ has a separate interrupt vector (\$FFF4) and a separate enable bit (X). Once the X bit is cleared (enabled), the software cannot disable it. A XIRQ interrupt is requested when the external XIRQ pin is low and the X bit in the CCR is 0. XIRQ processing will automatically set  $\mathbf{X=I=1}$  (an IRQ cannot interrupt an XIRQ service) at the start of the XIRQ handler. Just like regular interrupts, the X and I bits will be restored to their original values by the `rti` instruction.

**Checkpoint 4.6:** What makes XIRQ interrupts high priority?

## 4.5 Interrupt Vectors and Priority

The *priority* is given in the order shown in Table 4.3 with key wake-up P having the lowest priority and reset having the highest. Using the HPPIO register, the software can elevate one device to a high priority. Priority on the 9S12 means when two requests are made at the same time to determine in which order the requests will be serviced. More specifically, “at the same time” means multiple trigger flags occur during the execution of the same instruction. Usually an ISR runs with the I bit set (disabled). This means one interrupt will not suspend another interrupt while its ISR is running, even if the requesting interrupt is higher priority

Vector Address	CW number	Interrupt Source or Trigger flag	Enable	Local Arm	HPRI0 to Elevate
\$FFFE	0	Reset	none	none	—
\$FFF0	1	COP Clock Monitor Fail Reset	none	COPCTL.CME	—
			none	COPCTL.FCME	—
\$FFFFA	2	COP Failure Reset	none	COP rate selected	—
\$FFF8	3	Unimplemented Instruction Trap	none	none	—
\$FFF6	4	SWI	none	none	—
\$FFF4	5	XIRQ	X bit	none	—
\$FFF2	6	IRQ	I bit	INTCR.IRQEN	\$F2
\$FFF0	7	Real Time Interrupt, RTIF	I bit	CRGINT.RTIE	\$F0
\$FFEE	8	Timer Channel 0, C0F	I bit	TIE.C0I	\$EE
\$FFEC	9	Timer Channel 1, C1F	I bit	TIE.C1I	\$EC
\$FFEAA	10	Timer Channel 2, C2F	I bit	TIE.C2I	\$EA
\$FFE8	11	Timer Channel 3, C3F	I bit	TIE.C3I	\$E8
\$FFE6	12	Timer Channel 4, C4F	I bit	TIE.C4I	\$E6
\$FFE4	13	Timer Channel 5, C5F	I bit	TIE.C5I	\$E4
\$FFE2	14	Timer Channel 6, C6F	I bit	TIE.C6I	\$E2
\$FFE0	15	Timer Channel 7, C7F	I bit	TIE.C7I	\$E0
\$FFDE	16	Timer Overflow, TOF	I bit	TIE.TOI	\$DE
\$FFDC	17	Pulse Acc. Overflow, PAOVF	I bit	PACTL.PAOVI	\$DC
\$FFDA	18	Pulse Acc. Input Edge, PAIF	I bit	PACTL.PAI	\$DA
\$FFD8	19	SPI0 Transfer Complete, SPIF	I bit	SPI0CR1.SPIE	\$D8
		SPI0 Transmit Empty, SPTEF		SPI0CR1.SPTIE	
\$FFD6	20	SCI0 Transmit Buff Empty, TDRE	I bit	SCI0CR2.TIE	\$D6
		SCI0 Transmit Complete, TC		SCI0CR2.TCIE	
		SCI0 Receiver Buffer Full, RDRF		SCI0CR2.RIE	
		SCI0 Receiver Idle, IDLE		SCI0CR2.ILIE	
\$FFD4	21	SCI1 Transmit Buff Empty, TDRE	I bit	SCI1CR2.TIE	\$D4
		SCI1 Transmit Complete, TC		SCI1CR2.TCIE	
		SCI1 Receiver Buffer Full, RDRF		SCI1CR2.RIE	
		SCI1 Receiver Idle, IDLE		SCI1CR2.ILIE	
\$FFD2	22	ATD0 Sequence Complete, ASCIF	I bit	ATD0CTL2.ASCIE	\$D2
\$FFD0	23	ATD1 Sequence Complete, ASCIF	I bit	ATD1CTL2.ASCIE	\$D0
\$FFCE	24	Key Wakeup J, PIFJ.[7:6],[1:0]	I bit	PIEJ.[7:6],[1:0]	\$CE
\$FFCC	25	Key Wakeup H, PIFH.[7:0]	I bit	PIEH.[7:0]	\$CC
\$FFC8	27	Pulse Acc. Overflow, PBOVF	I bit	PBCTL.PBOVI	\$DC
\$FFC0	31	I2C	I bit	IBCR.IBIE	\$C0
\$FFBE	32	SPI1 Transfer Complete, SPIF	I bit	SPI1CR1.SPIE	\$BE
		SPI1 Transmit Empty, SPTEF		SPI1CR1.SPTIE	
\$FFBC	33	SPI2 Transfer Complete, SPIF	I bit	SPI2CR1.SPIE	\$BC
		SPI2 Transmit Empty, SPTEF		SPI2CR1.SPTIE	
\$FFB6	36	CAN wake-up	I bit	CANRIER.WUPIE	\$B6
\$FFB4	37	CAN errors	I bit	CANRIER.CSCIE	\$B4
				CANRIER.OVRIE	
\$FFB2	38	CAN receive	I bit	CANRIER.RXFIE	\$B2
\$FFB0	39	CAN transmit	I bit	CANTIER.TXEIE[2:0]	\$B0
\$FF8E	56	Key wake-up P, PIFP[7:0]	I bit	PIEP.[7:0]	\$8E

**Table 4.3**

Some of the interrupt vectors for the 9S12. CW stands for CodeWarrior.

than the running ISR. As long as the software is careful to acknowledge each specific request as it is being serviced, concurrent interrupt requests are not lost—just postponed until the running ISR executes `rti` and the I bit is cleared to zero (enabled) again.

Not all interrupt sources are available on every 9S12, but this list defines some of the interrupt sources. Any one particular application usually uses just a few interrupts.

In particular, those devices that need prompt service should be armed to request an interrupt. The software arms (specific for each possible source) and enables ( $I=0$  globally) interrupts. The external event triggers the interrupt by setting the trigger flag. The interrupt service routine (ISR) is executed in response to the trigger. The ISR acknowledges the interrupt by clearing the trigger flag.

For some interrupt sources, such as the SCI interrupts, flags are automatically cleared during the response to the interrupt requests. For example, the RDRF flag in the SCI system is cleared by the automatic clearing mechanism consisting of a read of the SCI status register while RDRF is set and followed by a read of the SCI data register. The normal response to an RDRF interrupt request is to read the SCI status register to check for receive errors and then to read the received data from the SCI data register. These two steps satisfy the automatic clearing mechanism without requiring any special instructions. On the other hand, many trigger flags employ a confusing (but effective) way for the software to acknowledge it. Flags such as RTIF, CnF, TOF, PIFJn, PIFHn, and PIFPn are cleared when the software writes a 1 into the bit position of that flag. Writing a zero to the flag register has no effect, and writing a \$FF clears all the flag bits in the register. Many of the potential interrupt requests share the same interrupt vector. For example, there are eight possible key wake-up interrupt sources (PH7-PH0) that all use the vector at \$FFCC. Therefore, when this request is processed, the ISR software must determine which of the eight possible signals caused the interrupt.

How we establish the vector depends on your compiler or assembler. Program 4.16 shows a key wake-up ISR that is triggered on the rise of PJ7, like you would have if you were to add interrupt synchronization and a mailbox to Example 3.3. The point of this program is to illustrate the syntax to establish an interrupt vector. In assembly, we use the `org` pseudo-op to place the address of the ISR in the appropriate location. From Table 4.3, we see the vector address for key wake-up J is \$FFCE. This means at address \$FFCE there is a 16-bit pointer to the ISR. The Metrowerks CodeWarrior uses the keyword `interrupt` along with the interrupt number. From Table 4.3, we see the CodeWarrior number for key wake-up J is 24. The assembly ISR ends in a `rti` instruction, whereas the C version uses the standard syntax of a function (brace or `return`) to signify the end of ISR. However, the computer will generate the interrupt vector and add the `rti` instruction so both versions of Program 4.16 generate essentially the same machine code.

<pre> KeyHanJ     movb #\$80,PIFJ ;clear flag7     movb PTT,Mail   ;create Mail     movb #1,Status  ;Signal     rti     org \$FFCE     fdb KeyHanJ </pre>	<pre> // interrupts on rise of PJ7 void interrupt 24 KeyHanJ(void){     PIFJ = 0x80; // acknowledge     Mail = PTT;   // mailbox     Status = 1;   // Signal } </pre>
---	---

### Program 4.16

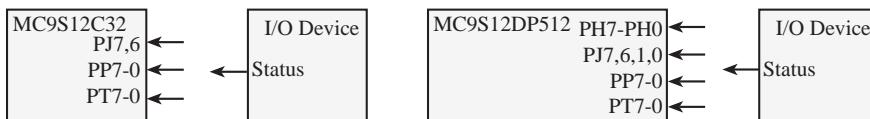
Software syntax to set the interrupt vector for a key wake-up interrupt.

In C, an ISR “looks” like a function, but you never explicitly call an ISR from some other place in the software, and the ISR must not have formal input or output parameters.

## 4.6 External Interrupt Design Approach

This short section deals with the design steps that occur when interfacing an I/O device with interrupt synchronization. Because the details of specific I/O devices can vary considerably, this section only serves as a framework, listing the issues we should consider when using interrupts. Computer engineering design, like all disciplines, is an iterative process. We will begin with the external hardware design.

First, we identify the busy-to-done state transition to cause the interrupt. We ask the question, what status signal from the I/O device signifies when the input device has new data and needs the software to read them. For an output device, we look for a signal that specifies when it is finished and needs the software to give it more data. As shown in Figure 4.22 this signal will be an output of the I/O device and an input to the microcomputer.



**Figure 4.22**

We connect external devices to microcomputer lines that can generate interrupts.

Next, we connect I/O status signal to a microcomputer input that can generate interrupts. On the MC9S12DP512, we could use the key wake-up on Ports H, J, P, or input capture on Port T. On the MC9S12C32, we could use key wake-up on Ports J, P or input capture on Port T.

There are four major components to the interrupting software system. In no particular order, they are the ritual, the main program, the ISR(s), and the interrupt vectors.

The *ritual* is executed once on start-up. It is probably a good idea to disable interrupts at the beginning of the ritual so that interrupts are not requested before the entire initialization sequence is allowed to finish. The ritual usually initializes the global data structures (e.g., `Fifo_Init`), sets the I/O port direction register(s) as needed, and sets the I/O port interrupt hardware control register, specifying the proper conditions to request the interrupt. An optional step is to clear the flag (the one that when set will request an interrupt). We like to add this optional step so that the first interrupt to occur will be the result of activity that occurred after the ritual, and not the result of activities prior to executing the ritual. For example, it is possible for the power-on sequence to different hardware modules to be different, causing edges to occur on digital lines that falsely mimic actual I/O activity. The last steps of the ritual are to arm the device and enable interrupts.

The *main program* executes in the foreground. At the start the SP is initialized, and the ritual is executed. The main program generally performs tasks that are not time-critical. The main program interacts with the ISRs via global memory data structures (e.g., FIFO queue).

The *interrupt handler* executes in the background. For a polled interrupt, it must determine the source by polling potential devices. The polling order will establish a priority, although later in the chapter we discuss more sophisticated methods for implementing priority. The ISR must acknowledge (clear the flag that requested the interrupt) or disarm. We acknowledge if we are interested in more interrupts and disarm if we are no longer interested in interrupts. Information is exchanged with the main program and other interrupt handlers (including itself) via global memory. The ISR executes `rts` to return control back to the program previously executing. Because of the real-time nature of interrupting devices, debugging tools must be minimally intrusive. Examples of good debugging tools to use for interrupts are (1) instrumentation that dumps into an array, (2) instrumentation that dumps into an array with filtering, and (3) instrumentation using an output port. The following are examples of bad debugging tools because they significantly affect the dynamic response of the interface (i.e., they require too much time to execute): (1) single-stepping, (2) breakpoints, and (3) print statements. These techniques were discussed in Section 2.11.

The last component is the *interrupt vectors*. On general-purpose computers, interrupt vectors are in RAM, and the software dynamically attaches and unattaches interrupt handlers to these vectors. On most embedded systems, the interrupt vectors reside in PROM or ROM and their values are determined at compile time and initialized by the ROM programmer. Nevertheless, we must establish interrupt vectors to point to the appropriate ISR. The syntax for setting vectors is compiler-dependent. Program 4.17 establishes an interrupt vector using the ImageCraft ICC12 and Metrowerks CodeWarrior compilers.

```
// ImageCraft ICC12 C for MC9S12C32
unsigned short Count;
#pragma interrupt_handler RTIHan()
void RTIHan(void){
    CRGFLG = 0x80; // ack, clear RTIF
    Count++; // number of interrupts
}
#pragma abs_address:0xffff0
void (*RTI_vector[1])() = { RTIHan };
#pragma end_abs_address
void RTI_Init(void){
    asm(" sei"); // Make ritual atomic
    CRGINT = 0x80; // RTIE=1 enable rti
    RTICTL = 0x33; // 4096us or 244.14Hz
    Count = 0; // interrupt counter
    asm(" cli");
}

// Metrowerks CodeWarrior C for MC9S12C32
unsigned short Count;

void interrupt 7 RTIHan(void){
    CRGFLG = 0x80; // ack, clear RTIF
    Count++; // number of interrupts
}

void RTI_Init(void){
    asm sei // Make ritual atomic
    CRGINT = 0x80; // RTIE=1 enable rti
    RTICTL = 0x33; // 4096us or 244.14Hz
    Count = 0; // interrupt counter
    asm cli
}
```

**Program 4.17**

ICC12 and CodeWarrior C syntax to set an interrupt vector.

**4.7 Polled versus Vectored Interrupts**

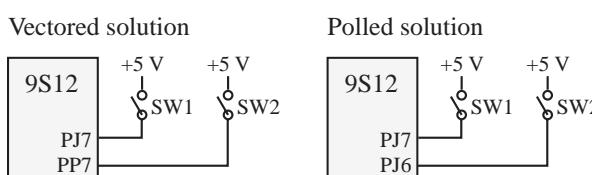
As we defined earlier, when more than one source of interrupt exists, the computer must have a reliable method to determine which interrupt request has been made. There are two common approaches, and the 9S12 applies a combination of both methods. The first approach is called vectored interrupts. With a *vectored interrupt* system each potential interrupt trigger source has a unique interrupt vector address. You simply place the correct handler address in each vector, and the hardware automatically calls the correct software when an interrupt is requested. With a *polled interrupt* system, multiple interrupt sources share the same interrupt vector address. Once the interrupt has occurred, the ISR software must poll the potential devices to determine which device needs service.

**Example 4.1** Interface two switches and signal associated semaphores when each switch is pressed.

**Solution** We will assume the switches do not bounce. The semaphore SW1 will be signaled when switch SW1 is pressed, and similarly, semaphore SW2 will be signalled when switch SW2 is pressed. In the first solution, we will use vectored interrupts by connecting one switch to Port J and the other switch to Port P. Since the two sources have separate vectors, the switch on Port J will automatically activate KeyHanJ and switch on Port P will automatically activate KeyHanP. The interface on the left of Figure 4.23 shows the solution with vectored interrupts.

**Figure 4.23**

Two solutions of switch-triggered interrupts.



The software solution using vectored interrupts is in Program 4.18. We initialize the positive logic switches using a passive pull-down resistor and a rising-edge key wake-up trigger. In this way, we get an interrupt request when the switch is touched. That is, an

interrupt occurs on the 0 to 1 rising edge of PJ7 or PP7. To acknowledge an interrupt, we clear the trigger flag. Writing an \$80 to the flag register will clear bit 7 without affecting the other bits in the register.

<pre> Init bclr DDRJ,#\$80 ;PJ7=SW1 bset PERJ,#\$80 ;pull down PJ7 bset PPSJ,#\$80 ;rise on PJ7 bset PIEJ,#\$80 ;arm PJ7 bclr DDRP,#\$80 ;PP7=SW2 bset PERP,#\$80 ;pull down PP7 bset PPSP,#\$80 ;rise on PP7 bset PIEP,#\$80 ;arm PP7 cli rts KeyHanJ movb #\$80,PIFJ ;acknowledge     movb #1,SW1      ;Signal SW1     rti KeyHanP movb #\$80,PIFP ;acknowledge     movb #1,SW2      ;Signal SW2     rti     org \$FFCE ; Key Wakeup J     fdb KeyHanJ     org \$FF8E ;Key Wakeup P     fdb KeyHanP </pre>	<pre> void Init(void){     DDRJ &amp;= ~0x80; // PJ7=SW1     PERJ  = 0x80; // pull down PJ7     PPSJ  = 0x80; // rise on PJ7     PIEJ  = 0x80; // arm PJ7     DDRP &amp;= ~0x80; // PP7=SW2     PERP  = 0x80; // pull down PP7     PPSP  = 0x80; // rise on PP7     PIEP  = 0x80; // arm PP7     asm cli }  void interrupt 24 KeyHanJ(void){     PIFJ = 0x80; // acknowledge     SW1 = 1;      // Signal SW1 occurred } void interrupt 56 KeyHanP(void){     PIFP = 0x80; // acknowledge     SW2 = 1;      // Signal SW2 occurred } </pre>
--	--

### Program 4.18

Example of a vectored interrupt.

The interface on the right of Figure 4.23 shows the solution with polled interrupts. Touching either switch will cause a Port J interrupt. The ISR must poll to see which one or possibly both caused the interrupt. Fortunately, even though they share a vector, the acknowledgements are separate. The code `PIFJ=0x80;` will clear bit 7 in the status register without affecting bit 6, and the code `PIFJ=0x40;` will clear bit 6 in the status register without affecting bit 7. This means the timing of one switch does not affect whether or not pushing the other switch will signal its semaphore. On the other hand, whether we are using polled or vectored interrupt (because there is only one CPU), the timing of one interrupt may delay the servicing of another interrupt. The polled solution is Program 4.19. It takes three conditions to cause an interrupt.

1. The PJ7 and PJ6 are armed in the initialization;
2. The 9S12 is enabled for interrupts with the `cli` instruction;
3. The key wakeup trigger PIFJ7 is set on the rising edge of PJ7 or the key wakeup trigger PIFJ6 is set on the rising edge of PJ6.

Because the two triggers have separate acknowledgments, if both triggers are set, both will get serviced. Furthermore, the polling sequence does not matter.

<pre> Init bclr DDRJ,#\$C0 ;PJ7=SW1, PJ6=SW2 bset PERJ,#\$C0 ;pulldown PJ7 PJ6 bset PPSJ,#\$C0 ;rise on PJ7 PJ6 bset PIEJ,#\$C0 ;arm PJ7 PJ6 cli rts </pre>	<pre> void Init(void){     DDRJ &amp;= ~0xC0; // PJ7=SW1, PJ6=SW2     PERJ  = 0xC0; // pull down PJ7 PJ6     PPSJ  = 0xC0; // rise on PJ7 PJ6     PIEJ  = 0xC0; // arm PJ7 PJ6     asm cli } </pre>
---	---

<pre> ;interrupt on SW1 or SW2 pushed KeyHanJ     brclr PIFJ,#\$80,ch6     movb #\$80,PIFJ ;acknowledge SW1     movb #1,SW1      ;Signal SW1 ch6  brclr PIFJ,#\$40,done     movb #\$40,PIFJ ;acknowledge SW2     movb #1,SW2      ;Signal SW2 done rti     org  \$FFCE ; Key Wakeup J     fdb  KeyHanJ </pre>	<pre> // interrupt on either SW1 or SW2 pushed void interrupt 24 KeyHanJ(void){     if(PIFJ&amp;0x80){ // poll PJ7         PIFJ = 0x80; // acknowledge SW1         SW1 = 1;      // Signal SW1 occurred     }     if(PIFJ&amp;0x40){ // poll PJ6         PIFJ = 0x40; // acknowledge SW2         SW2 = 1;      // Signal SW2 occurred     } } </pre>
---	--

### Program 4.19

Example of a polled interrupt.

**Common error:** If two interrupts were requested, it would be a mistake to service just one and acknowledge them both.

**Observation:** External events are often asynchronous to program execution, so careful thought is required to consider the effect if an external interrupt request were to come in between each pair of instructions.

**Observation:** The computer automatically sets the I bit during processing so that an interrupt handler will not interrupt itself.

Two polling techniques are presented in this book. The first is *minimal* polling. This method performs a minimally sufficient check to determine the source. Usually this entails simply checking the particular flag bit that caused the interrupt. The above interrupt polling are examples of minimal polling. A more robust polling method is called *polling for 0s and 1s*. This verifies as much information in the control/status register as possible and usually entails checking for the presence of both 1s and 0s in the control/status register. For example, assume the interrupt flag bit is in bit 7, bit 6 is unknown, and bits 5–0 should be 000111. We write this expected condition as 1x000111. Simple polling only checks the flag bit, while polling for 0s and 1s compares the actual 7 bits with their expected values. One implementation is

```

ldaa CSR      read control/status register
anda #%1011111 clear bits that are indeterminate
cmpa #%1000111 expected value if this device active
beq Handler   execute if this device is requesting

```

It is good software engineering practice to install consistency checks during the prototype and debugging stages. Once debugged, the system can be optimizing by removing them.

## 4.8 Pseudo-Interrupt Vectors

Some development boards do not allow you to erase and reprogram the real interrupt vectors from \$FF80 to \$FFFF. In these development systems, the fixed or protected ROM at \$FF80 to \$FFFF point to places defined by the debugger, and an indirect call method allows you to run your ISRs. The locations to which the real vectors point are called *pseudo-interrupt* vectors. Typically, the pseudo-interrupt vectors are defined in the same order as the real vectors. In the old 6811 development boards, each pseudo vector was in RAM and required 3 bytes. Three bytes were required to place a `jmp` instruction to your ISR. During a 6811 initialization, the program places `jmp` instructions into the pseudo vectors. In contrast, most 9S12 debuggers only require 2 bytes for each pseudo vector, as shown in Table 4.4. Only the first nine are shown in the table. However, in each case, there is a fixed offset between the address of the real interrupt vector and the address of the pseudo-interrupt vector. The MON12 debugger on

9S12 boards from Axiom (<http://www.axman.com>) and the D-Bug12 debugger from Technological Arts implement 16-bit pseudo vectors in RAM. During initialization at run-time, your program must place pointers to your ISRs into the pseudo vectors. Every time an interrupt occurs the Axiom MON12 debugger requires 21 extra bus cycles to implement the indirect jump to your ISR. The Freescale Serial Monitor used by Metrowerks CodeWarrior and TExaS also employs, pseudo vectors. The difference is the Serial Monitor pseudo vectors are in EEPROM. Your software does not have to perform any run-time initialization of the pseudo vector. Rather, the Serial Monitor itself will automatically translate a “Program ROM” command from \$FF80-\$FFFF down to \$F780-\$F7FF. For example, this code is the proper way to set the TC0 interrupt vector in a system without pseudo vectors; see Table 4.3.

```
org  $FFEE
fdb TC0han
```

**Table 4.4**

The first nine pseudo-interrupt vectors for the 9S12.

Real Vector	MON12 Pseudo Vector	D-Bug12 Pseudo Vector	Serial Monitor Pseudo Vector	Interrupt Source or Trigger Flag
\$FFFE	none	none	none	Reset
\$FFFC	\$0FFC	none	\$F7FC	COP Clock Monitor Fail Reset
\$FFFA	\$0FFA	none	\$F7FA	COP Failure Reset
\$FFF8	\$0FF8	\$3E78	\$F7F8	Unimplemented Instruction Trap
\$FFF6	\$0FF6	\$3E76	\$F7F6	SWI
\$FFF4	\$0FF4	\$3E74	\$F7F4	XIRQ
\$FFF2	\$0FF2	\$3E72	\$F7F2	IRQ
\$FFF0	\$0FF0	\$3E70	\$F7F0	Real Time Interrupt, RTIF
\$FFEE	\$0FEE	\$3E6E	\$F7EE	Timer Channel 0, C0F

However, when your software is downloaded and programmed into EEPROM, this vector transparently and automatically ends up being programmed at \$F7EE. Every time an interrupt occurs the Serial Monitor requires 19 extra bus cycles to implement the pseudo vector. The actual Serial Monitor code for an interrupt is

```
uvector08: bsr ISRHandler ;TC0 interrupt starts executing here
...
ISRHandler: pulx           ;pull bsr return address off stack
ldy -$0636,X               ;get value of pseudo vector
cpy #$FFF                 ;is it programmed?
beq BadVector
jmp ,Y                     ;jump to your ISR
```

SCI interrupts with the serial monitor include an overhead longer than 19 cycles, because the SCI interrupts are used by the debugger itself to perform its actions. In particular, after a SCI interrupt the debugger will check the LOAD/RUN switch on the board to see if the debugger or user program should process the interrupt.

**Performance tip:** The BDM debugger allows you to set the real interrupt vectors, eliminating the 19-cycle overhead during each interrupt.

## 4.9 Key Wake-Up Interrupt Examples

The number of key wake-up pins varies from one 9S12 version to another, but they all operate similarly. The MC9S12C32 has 10 (Ports J and P), and the MC9S12DP512 has 20 (Ports H, J, and P). Key wake-up pins can be configured to cause interrupts on the rise or the fall. It is good design to initially clear the trigger flags, so the first interrupt is generated by a touch occurring after the ritual is executed and not due to edges that may occur during

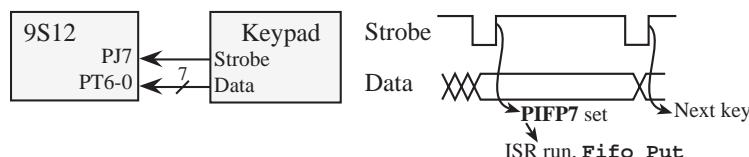
power up or during configuration. There are seven software steps to initialize pins for key wakeup interrupts. For each pin,

1. The direction register is cleared making the pin an input (DDRH, DDRJ, DDRP)
2. Activate the rise or fall to trigger an interrupt (PPSH, PPSJ, PPPS)
3. Enable or disable the passive pull-up/pull-down resistor (PERH, PERJ, PERP)
4. Arm the pin to generate a key wake up interrupt (PIEH, PIEJ, PIEP)
5. Initially clear key wake up trigger (PIFH, PIFJ, PIFP)
6. Initialize variables and structures (Fifo\_Init)
7. Enable interrupts on the 9S12 (cli)

**Example 4.2** Redesign Example 3.3 of a parallel keypad interface using interrupt synchronization. When the operator types a key, the ASCII data becomes available and there is a rising edge on a **Strobe** signal. The data are put into a FIFO queue. The main program will get data out of the FIFO.

**Solution** Figure 3.20, redrawn as Figure 4.24, shows a circuit and the timing for this interface. When the user types a key on this keypad, the 7-bit ASCII code becomes available on the **PTT**, followed by a rise in the signal **PJ7**. The data remains available until the next key is typed. Program 4.20 shows the software solution. The initialization sets PJ7 to interrupt on the rise.

**Figure 4.24**  
A keyboard is interfaced to the microcomputer.



The ISR will poll to verify the interrupt has occurred. If the software is executing at this point and bit 7 in PIFJ is not set, then a serious hardware or software error has occurred. In this system, the function **Error** will handle this serious error. We expect PIFJ7 to be set, so the ISR inputs data, acknowledges the interrupt, and puts the data into a FIFO. The implementation of the FIFO was presented previously as Program 4.14.

<pre> Key_Init sei          ;make atomic     movb #\$80,DDRT   ;PT7 heartbeat     bclr DDRJ,#\$80   ;PJ7 input     bset PPSJ,#\$80   ;rise on PJ7     bclr PERJ,#\$80   ;no resistor     bset PIEJ,#\$80   ;arm PJ7     movb #\$80,PIFJ   ;clear flag7     jsr  Fifo_Init     cli              ;Enable IRQ     rts KeyHan bclr PTT,#\$80  ; heartbeat     brset PIFJ,#\$80,ok     jsr  Error        ;really bad ok     movb #\$80,PIFJ  ;ack     ldaa PTT           ;read key     jsr  Fifo_Put     bset PTT,#\$80     rti     org \$FFCE         ;Key wakeup J     fdb  KeyHan </pre>	<pre> // PT6-PT0 inputs = keyboard Data // PJ7=Strobe interrupt on rise void Key_Init(void){ asm sei     DDRT  = 0x80; // PT6-0 DATA     PPSJ  = 0x80; // rise on PJ7     PERJ &amp;= ~0x80; // no resistor     PIEJ  = 0x80; // arm PJ7     PIFJ  = 0x80; // clear flag7     Fifo_Init(); asm cli }  void interrupt 24 KeyHan(void){     PTT &amp;= ~0x80; // heartbeat     if((PIFJ&amp;0x80)==0) Error();     PIFJ =0x80; // clear flag     Fifo_Put(PTT);     PTT  = 0x80; } </pre>
--	---

### Program 4.20

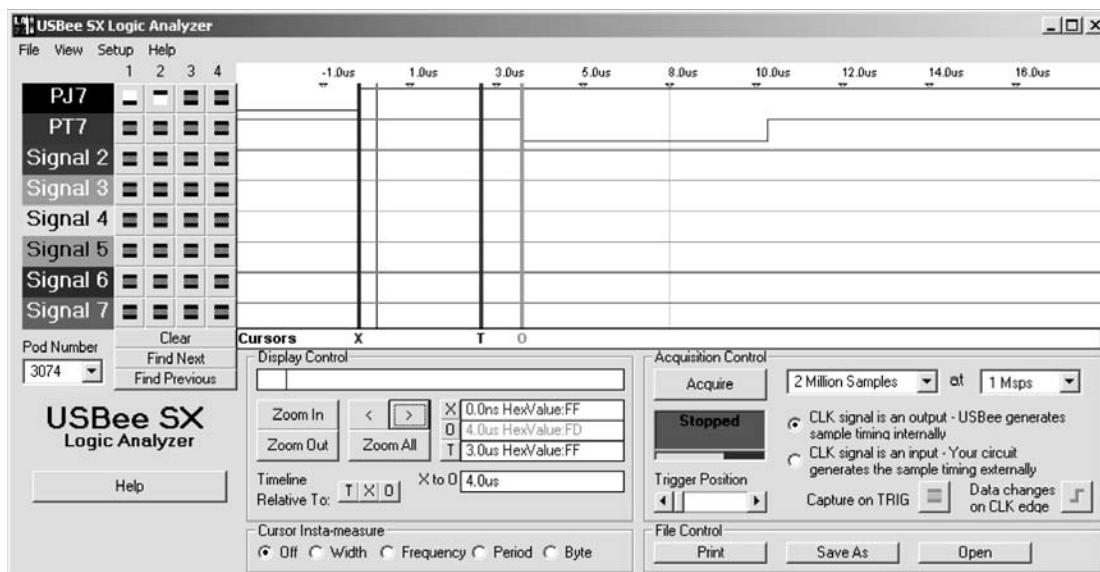
Interrupting keyboard software.

A debugging heartbeat was added to PT7. This instrument was added to allow us to see if and when the ISR runs. The falling edge of PT7 signifies the start of the ISR, and the rising edge occurs at the end of the ISR. The `bset` and `bclr` instructions each take four cycles, so running at 8 MHz, this debugging instrument adds 1  $\mu$ s of execution time to each ISR. If the user types a new character every 100 ms, this added 1  $\mu$ s will be minimally intrusive.

**Performance tip:** In the early stages of a development project, inserting consistency checks (like polling in the above example when it wasn't necessary) can identify hardware and software bugs. Once the system is thoroughly tested, you could remove the checks for enhanced speed.

**Observation:** When power is applied to a system, each device turns on separately; therefore, it is possible to get initial rising and/or falling edges that do not represent actual I/O events. Therefore, it is good software practice to clear interrupt flags in the initialization, so that the first interrupt represents the first I/O event.

The latency for Example 4.2 is the time between the rise of **Strobe** and the time when the software reads the input data. Even though the data collected here is specific for Example 4.2, the results are applicable to all interrupt-driven I/O. Figure 4.25 is a logic analyzer trace showing the timing between rise of PJ7 and the fall of PT7. This measurement was collected on a MC9S12DP512 running at 8 MHz. After the current instruction is executed (`bra loop` in the main program takes 0 to 3 cycles), the context switch takes nine cycles, the serial monitor takes 19 cycles, and the `bclr PTT, #$80` takes four cycles. This estimate of 32 to 35 cycles is consistent with the 4  $\mu$ sec latency measured on the logic analyzer. The latency for this system is small, because there are no other interrupts to postpone the key wake-up ISR. In a more complicated system, we would take this measurement of 4  $\mu$ sec and add the maximum time the 9S12 runs with  $I=0$ . Typically, this added latency occurs because of the time it takes to run other ISRs.



**Figure 4.25**

Logic analyzer output showing latency of the keyboard interface.

**Observation:** The CycleView mode of the TExaS simulator allows you to observe the bus activity during an interrupt service—both the thread-switch from main to ISR and from ISR to main.

**Observation:** Data are lost when the FIFO gets full.

**Checkpoint 4.7:** Look up in the CPU12 manual how long it takes to execute `rti`.

**Checkpoint 4.8:** For the system used to collect Figure 4.25, about how long does it take to execute one pass through the ISR? The ISR is only eight instructions long, so why did it take so long to execute?

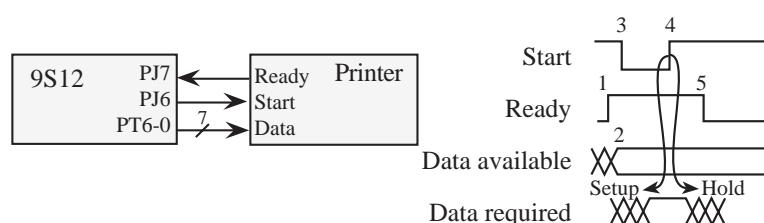
**Checkpoint 4.9:** For \$100 you can buy a BDM debugger, eliminating the need for the serial monitor. Using a BDM, what would you estimate the latency to be for an interrupt-driven I/O device using an 8-MHz 9S12, assuming no other interrupts?

**Example 4.3** Redesign Example 3.6 of a parallel printer interface using interrupt synchronization. The main program will put ASCII data into the FIFO. The ISR will get data from the FIFO and output it to the printer.

**Solution** Figure 3.23, redrawn as 4.26, shows the timing of this interface. The rising edge of **Ready** will trigger an interrupt using key wake-up on PJ7, labeled as step 1. This edge corresponds to the busy-to-done state transition in the printer. We will use a helper function, called `out`, which will initiate a print operation by performing Steps 2, 3, and 4, as shown in Program 4.21. In C, we specify a private function using the qualifier `static`. A static function is one that only can be called by other functions in the same file. The ISR will remove an ASCII character from the FIFO and output it to the printer. The printer will drop its **Ready** line, signifying the print operation is in progress (Step 5). At the completion of each output, a new interrupt will be requested. The printer uses the rise of **Start** to latch the data. The FIFO routines of Program 4.14 are used.

There are three special cases to handle. If the printer is idle (signified by PIEJ bit 7 disarmed), a call to `Print_Out` will initiate the output and arm the PJ7 interrupts. The sequence for this initial output can be seen as Steps 1, 2, 3, and 4 in Figure 4.26. The second special case occurs when the user calls `Print_Out` and the FIFO is full. There are two ways to handle this condition. We could just discard the data. We handle this condition by spinning, trying to put data into the FIFO over and over until it is successful. No matter how we handle it, the FIFO being full is a poor design. Therefore, we use the debugging phase of the project to increase the size until the FIFO either rarely becomes full or never becomes full. The third special case occurs when an interrupt occurs (printer is idle), but the FIFO is empty. In this case, the key wake-up will be disarmed. The flowchart for this producer/consumer system is shown on the right side of Figure 4.7. If the printer is busy and data are in the FIFO, the interrupt sequence is 1, 2, 3, 4, and 5, as shown in Figure 4.26. The process begins with Step 1: The printer is done with the last character. This edge triggers an interrupt. In Step 2: the ISR software gets from the FIFO and outputs to Port T. Steps 3 and 4 occur as the ISR causes the **Start** pulse on PJ6. Step 5, the fall of **Ready**, occurs as the printer begins printing the new data.

**Figure 4.26**  
Interrupt-driven hardware interface between a printer and the microcomputer.



The initialization is similar to Program 3.9, disarming PJ7 because the FIFO is empty, and there is no data to print. A debugging heartbeat was added to PT7, allowing us to see if and when the ISR runs. The falling edge of PT7 signifies the start of the ISR and the rising edge occurs at the end of the ISR. This debugging instrument adds eight cycles of execution time to each ISR, which in most cases will be minimally intrusive.

<pre> Print_Init     sei           ;make atomic     jsr Fifo_Init ;buffered I/O     bset PTJ,\$40  ;Start=1     bset DDRJ,\$40 ;PJ6 out     bclr DDRJ,\$80 ;PJ7 input     bset PPSJ,\$80  ;rise on PJ7     bclr PERJ,\$80 ;no resistor     bclr PIEJ,\$80  ;disarm PJ7     movb #\$FF,DDRT ;PT7 heartbeat     cli           ;Enable IRQ     rts  out     ldab PTT      ;RegA has data     andb #\$80     ;leave bit 7     aba          ;friendly     staa PTT      ;output     bclr PTF,\$40   ;Start=0     bset PTF,\$40   ;Start=1     movb #\$80,PIFJ ;clear flag7     rts  Print_Out psha      ;RegA has data     brclr PIEJ,\$80,go  loop     ldaa 0,s      ;data     jsr Fifo_Put   ;A=0 if full     tbeq A,loop    ;try again     bra done  go     jsr out       ;start print     bset PIEJ,\$80  ;arm PJ7  done     pula     rts  KeyHan     bclr PTT,\$80    ; heartbeat     brset PIFJ,\$80,ok     jsr Error       ;really bad  ok     movb #\$80,PIFJ ;ack     leas -1,sp     tsx            ;X=&gt; data     jsr Fifo_Get     tbeq A,mt      ;A=0 if empty     pula          ;data     bsr out        ;output it     bra then  mt     leas 1,sp     bclr PIEJ,\$80  ;disarm PJ7  then     bset PTT,\$80     rti     org \$FFCE    ;Key wakeup J     fdb KeyHan </pre>	<pre> void Print_Init(void){     asm sei           // make atomic     Fifo_Init();      // FIFO uses as buffer     PTJ  = 0x40;      // Start=1     DDRJ  = 0x40;      // PJ6 Start out     DDRJ &amp;= ~0x80;    // PJ7 Ready in     PPSJ  = 0x80;      // rise on PJ7     PERJ &amp;= ~0x80;    // no resistor     PIEJ &amp;= ~0x80;    // initially disarmed     DDRT = 0xFF;      // PT7-0 Data out     asm cli }  // helper function, low-level handshake void static out(unsigned char data){     PTT = (PTT&amp;0x80)+data; // out to printer     PTJ &amp;= ~0x40;          // Start=0     PTJ  = 0x40;          // Start=1     PIFJ = 0x80;          // clear flag7 } // Send one character to the printer void Print_Out(unsigned char data){     if(PIEJ&amp;0x80){        // already running         while(Fifo_Put(data))==0{}; // save     } else{                // not running         out(data);         // start up printer         PIEJ  = 0x80;      // arm PJ7     } }  void interrupt 24 KeyHan(void){     unsigned char newdata;     PTT &amp;= ~0x80;          // heartbeat     if((PIFJ&amp;0x80)==0)Error();     if(Fifo_Get(&amp;newdata))         out(newdata); // start next     else{         PIEJ &amp;=~0x80; // disarm     }     PTT  = 0x80; } </pre>
---	--

### Program 4.21

Software solution for the printer interface.

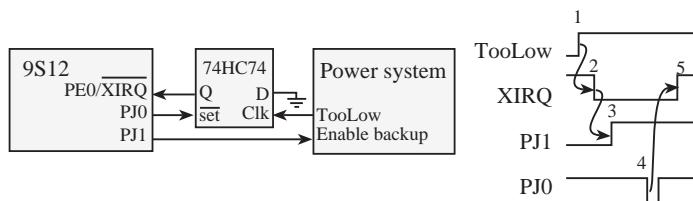
The solution in Example 4.3 disarmed the interrupt when no data were available to print, and it then rearmed it when new data were produced. Another solution involves the use of non-printing dummy characters. In this technique, a special non-printing character like NULL (\$00) or SYN (\$16) is transmitted when the device is ready but there is nothing to print. This method is a little simpler, because one does not have to arm, disarm, and re-arm. The disadvantage is that software overhead is required to process interrupts when no real function is being performed. In other words, the main thread will execute slower. In addition, the printer must discard these dummy characters. The arm/disarm technique will be presented again in the serial communications chapter.

## 4.10 Power System Interface Using XIRQ Synchronization

The objective of this section is to use XIRQ interrupts to create a very low latency interface (Figure 4.27). This power system will monitor the voltage level from the regular power supply. If the level drops below a safe threshold, it will trigger an XIRQ interrupt by signaling the problem with a rising edge on **TooLow**. This rising edge will clear the 74HC74 flip-flop, making its output, XIRQ, low. The XIRQ handler will service the crisis by enabling the backup power system by making PJ1=1. The handler can acknowledge the XIRQ interrupt by toggling PJ0=0, then PJ0=1 again. Assuming the program is running with XIRQ interrupts enabled, the latency of this interface is quite low (Program 4.22).

**Figure 4.27**

Hardware interface of an XIRQ interrupting device.



Software can only enable XIRQ and cannot disable XIRQ. In other words, once the software clears the Xbit, the software cannot make the X bit become 1. In this way, XIRQ interrupts are non-maskable. When the software makes Port J bit 0 low, the 74HC74 flip-flop is set, causing the XIRQ line to go high (inactive). When the software makes Port J bit 0 high, the 74HC74 is ready to receive a rising edge on **TooLow**. The rising edge of **TooLow** clears the flip-flop, requesting an XIRQ interrupt.

<pre> Init bset DDRJ,#\$03 ;outputs     bclr PTJ,#\$03 ;backup off     bset PTJ,#\$01 ;ready to receive     ldaa #\$10      ;Enable XIRQ     tap     rts  XHan bset PTJ,#2  ;backup power     bclr PTJ,#1   ;ack XIRQ     bset PTJ,#\$01 ;ready to receive     rti     org \$FFFF4     fdb XHan     ;XIRQ vector </pre>	<pre> void Init(void){     DDRJ  = 0x03; // PJ1,PJ0 outputs     PTJ &amp;= ~0x03; // backup off     PTJ  = 0x01; // ready to receive     asm ldaa #0x10     asm tap        // enable XIRQ }  void interrupt 5 XHan(void){     PTJ  = 0x02; // backup power     PTJ &amp;= ~0x01; // ack XIRQ     PTJ  = 0x01; // ready to receive } </pre>
---	--

**Program 4.22**

Software to implement battery backup using XIRQ interrupts.

The latency of this interface is defined as the time from the rising edge of **ToLow** to when the software enables the backup power. It includes the time to finish the current instruction, the nine cycles to process the XIRQ, and the four cycles needed to execute the `bset` instruction in `XHan`. If you are using a serial monitor, an additional 19 cycles are required for the debugger to process the interrupt request. XIRQ interrupts are fundamentally different than the other interrupts. When considering latency for the other interrupts, we must add the maximum time the software runs with interrupts disabled. In a system with more than one interrupt source, the time running with `I=1` is usually much larger than the nine cycles it takes for the context switch. Conversely, XIRQ interrupts are not postponed by disabling interrupts with the `I` bit. The longest atomic instructions on the 9S12 are division and multiply-accumulate. For example, `ediv edivs fdiv idivs` and `emacs` require 11 to 13 cycles to execute. This means once `Init` is called there will be a maximum delay of 26 cycles between the rising edge of **ToLow** and the rising edge of **PJ1**.

**Observation:** Some older computers have a nonmaskable interrupt (NMI) that is always enabled. These computers have problems when a NMI is requested before the system has been initialized, because the system is not yet ready to handle the NMI request.

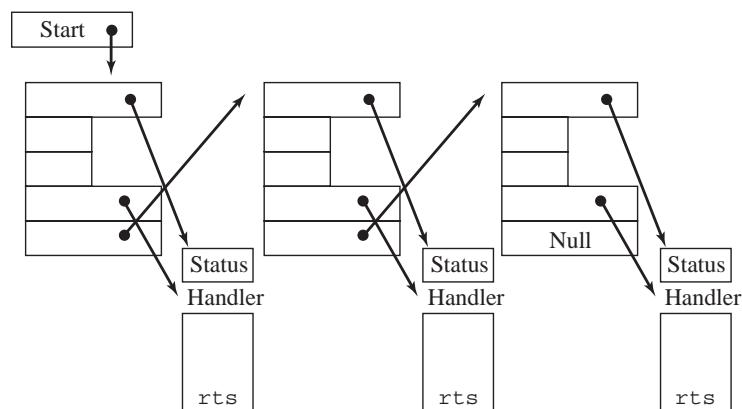
**Common error:** It is a mistake to enable XIRQ interrupts before the system has been initialized to handle the XIRQ request.

## 4.11 Interrupt Polling Using Linked Lists

The process of polling involves software at the beginning of the ISR that checks one by one the list of possible devices that might have caused the interrupt. When the interrupt polling software detects a device that needs service, it executes the appropriate device driver to service the interrupt. There are two considerations when we decide whether or not to poll at the beginning of the interrupt handler. First, if there are two or more potential sources of interrupt that operate through the same interrupt vector, then we must poll to determine which one is interrupting. Examples of this situation are (1) key wake-up and (2) the asynchronous serial port, SCI. Second, even if we do not have to poll because our interrupting device has its own dedicated vector, sometimes we will poll anyway just to add a layer of robustness to our software. In this way, if the software arrives at the first location of the interrupt handler due to a software or hardware error, our software can determine an error has occurred because the device that should have been ready is not.

In this example, we implement polling using linked lists (Figure 4.28). The purpose of using linked lists is to make it easier to debug, change the polling order, add devices, or subtract devices. Although this example uses a statically allocated linked list, if the linked list was created at run time, then changing polling order or adding/subtracting devices could be performed dynamically.

**Figure 4.28**  
Linked list data structure used to implement polling.



In this implementation, we have three potential interrupt sources using the same interrupt vector. In this system we will have three key wakeup interrupts on PJ2, PJ1, and PJ0. The purpose of this example is to illustrate the use of linked lists when implementing interrupt polling. There will be one node for each potential device. Each node of the statically allocated linked list has enough information to poll the device. If successful, the node also has a handler address that will be called. This solution is simple because all key wakeups on Port J have the same status register, and we only need to test for 1s and no 0s. Program 4.23 shows the 9S12 assembly language.

### Program 4.23

9S12 assembly structure for interrupt polling using linked lists.

```

start    fdb 11PJ2      ;place to start polling
Mask     equ 0          ;      and mask
DevHan   equ 1          ;      device handler
NextPt   equ 3          ;      next pointer
num      fcb 3          ;number of devices
11PJ2    fcb $04        ;look at bit 2
          fdb PJ2han    ;device handler
          fdb 11PJ1      ;pointer to next device to poll
11PJ1    fcb $02        ;look at bit 1
          fdb PJ1han    ;device handler
          fdb 11PJ0      ;pointer to next device to poll
11PJ0    fcb $01        ;look at bit 0
          fdb PJ0han    ;device handler
          fdb 0           ;end of list

```

The interrupt handler performs the following four steps: (1) read status register (ldaa, Y), (2) determine if this device is requesting (anda), and (3) execute the handler (jsr) if it is requesting (Program 4.24). This particular implementation does not report an error if the interrupt is from an unknown source. We will see in Section 4.14 how to modify this solution to implement round-robin polling. We assume each device handler terminates with a RTS instruction and saves registers B and X. In C, this linked list polling system is shown in Program 4.25

### Program 4.24

9S12 assembly implementation of interrupt polling using linked lists.

```

IrqHan  ldx start      ;Reg X points to linked list place to start
        ldab num       ;number of possible devices
next    ldaa PIFJ      ;read status
        anda Mask,x   ;check if proper bit is set
        beq Notyet    ;skip if this device not requesting
        jsr [DevHan,x] ;call device handler, will return here
Notyet  ldx NextPt,x  ;Reg X points to next entry
        decb          ;device counter
        bne next      ;check next device
        rti

```

```

const struct Node{
    unsigned char Mask;          // And Mask
    void (*Handler)(void);       // Handler for this task
    const struct Node *NextPt;   // Link to Next Node
};

```

*continued on p. 230*

### Program 4.25

C language implementation of interrupt polling on the 9S12 using linked lists.

*continued from p. 229*

```

unsigned char Counter2,Counter1,Counter0;
void PJ2Han(void){ // regular function that returns (rts) when done
    PIFJ = 0x04; // acknowledge
    Counter2++;
}
void PJ1Han(void){ // regular function that returns (rts) when done
    PIFJ = 0x02; // acknowledge
    Counter1++;
}
void PJ0Han(void){ // regular function that returns (rts) when done
    PIFJ = 0x01; // acknowledge
    Counter0++;
}
typedef const struct Node NodeType;
typedef NodeType * NodePtr;
NodeType Tasks[3]={
    {0x04, PJ2Han, &Tasks[1]},
    {0x02, PJ1Han, &Tasks[2]},
    {0x01, PJ0Han, 0 } };
void interrupt 24 KeyHanJ(void){ NodePtr Pt; unsigned char Status;
Pt = &Tasks[0]; // points to linked list
while(Pt){ // executes device handlers for all requests
    if(PIFJ&(Pt->Mask)){
        (*Pt->Handler)();} // Execute handler
    }
    Pt = Pt->NextPt; // returns after all devices have been polled
}
}

```

### Program 4.25

C language implementation of interrupt polling on the 9S12 using linked lists.

## 4.12 Interrupt Priority

**Priority** determines the order of service when two or more requests are made simultaneously. Priority also allows a higher priority request to suspend a lower priority request currently being processed. Usually, if two requests have the same priority, we do not allow them to interrupt each other. Later in the chapter, we will implement round robin scheduling with equal priority, so that no one device can monopolize the computer. However, in this section, we will study methods to implement priority. More sophisticated microcontrollers, like the Arm Cortex M3, include a hardware interrupt priority controller. This hardware component inside the microcontroller assigns a priority level to each interrupt trigger. This mechanism allows a higher priority trigger to interrupt the ISR of a lower priority request. Conversely, if a lower priority request occurs while running an ISR of a higher priority trigger, it will be postponed until the higher priority service is complete. On the 9S12, we can implement a priority system if there are only two requests. If we have more than two requests, we can implement a solution on the 9S12 where one request has high priority and all of the other requests have equal, but lower priority.

Consider the simple case with two devices called Device 1 and Device 2, such that Device 1 is higher priority than Device 2. There are two steps that we need to perform in order to let Device 1 go ahead of Device 2. First, if both devices request service simultaneously, then obviously we let Device 1 go first. The more difficult situation is when

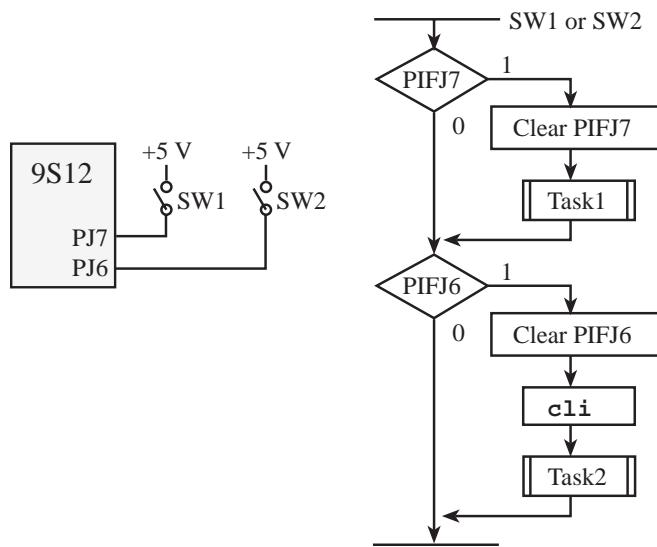
Device 2 requests an interrupt and is allowed to start its ISR and then Device 1 requests service. For this scenario, we need a mechanism to suspend the ISR of Device 2, serve Device 1 immediately, and then finish the service for Device 2 after we are done with Device 1. We can accomplish this effect by re-enabling interrupts in the Device 2 handler after the Device 2 trigger has been acknowledged. The software will crash if we re-enable interrupts in the Device 2 handler before we acknowledge it. If we enable interrupts in an ISR while the trigger flag is still set, the three conditions (1) flag set, (2) flag armed, and (3) I=0 would cause Device 2 to interrupt itself over and over. Obviously, we would not want to re-enable interrupts while servicing the higher priority device. This *selective re-enabling* approach works for two devices, but it doesn't allow us to create a fully ordered priority solution for three or more devices. For example, assume we have three devices, such that 1 is higher priority than 2, and 2 is higher priority than 3. Clearly, we would re-enable interrupts for the lowest priority device and not re-enable interrupts for the highest priority device. The question with no good answer is whether to re-enable interrupts while servicing the middle priority device. If we do re-enable interrupts while servicing the middle device, then we implement a two-level priority system where 1 is higher than 2 and 3, but 2 and 3 essentially have the same priority. If we do not re-enable interrupts while servicing the middle device, then we implement a different two-level priority system where 1 and 2 are higher than 3, but 1 and 2 have the same priority. Luckily, when we are interested in priority on an embedded system, it usually fits the two-level situation where one device needs to have priority over all the rest. To implement a two-level priority system, we enable interrupts for the lower priority and do not re-enable interrupts for the higher priority devices.

**Example 4.4** There are two switches with two associated software tasks. The associated task should be executed whenever its switch is pushed. That is, a rising edge triggers an interrupt, and the ISR calls the task. However, Switch SW1 and Task1 should have priority over Switch SW2 and Task2.

**Solution** In the first solution, the two switches are interfaced to the same key wake-up as shown in Figure 4.29. SW1 should have higher priority over SW2. Since they share a common interrupt vector, polling is required. Since SW1 is higher priority, it is polled first. In this way if SW1 and SW2 occur simultaneously, SW1 is serviced first. If SW2 has interrupted, we will clear its trigger flag and reenable interrupts. In this way, if we are executing Task2, and SW1 occurs Task2 will be suspended, and Task1 will be run immediately.

**Figure 4.29**

A polled interrupt where Switch SW1 has higher priority than Switch SW2.



Touching either switch will cause a Port J interrupt. The polled solution in Program 4.26 is similar to Program 4.19. The key wake-up trigger PIFJ7 is set on the rising edge of PJ7 and the key wake-up trigger PIFJ6 is set on the rising edge of PJ6. Because the two triggers have separate acknowledgments, if both triggers are set, both will get serviced.

<pre> Init bclr DDRJ,#\$C0 ;PJ7=SW1, PJ6=SW2     bset PERJ,#\$C0 ;pulldown PJ7 PJ6     bset PPSJ,#\$C0 ;rise on PJ7 PJ6     bset PIEJ,#\$C0 ;arm PJ7 PJ6     cli     rts  ;interrupt on SW1 or SW2 pushed KeyHanJ     brclr PIFJ,#\$80,ch6     movb #\$80,PIFJ ;acknowledge SW1     jsr Task1 ;service SW1 ch6 brclr PIFJ,#\$40,done     movb #\$40,PIFJ ;acknowledge SW2     cli     jsr Task2 ;service SW2 done rti     org \$FFCE ; Key Wakeup J     fdb KeyHanJ </pre>	<pre> void Init(void){     DDRJ &amp;= ~\$0xC0; // PJ7=SW1, PJ6=SW2     PERJ  = \$0xC0; // pull down PJ7 PJ6     PPSJ  = \$0xC0; // rise on PJ7 PJ6     PIEJ  = \$0xC0; // arm PJ7 PJ6     asm cli }  // interrupt on either SW1 or SW2 pushed void interrupt 24 KeyHanJ(void){     if(PIFJ&amp;0x80){ // poll PJ7         PIFJ = 0x80; // acknowledge SW1         Task1(); // service SW1     }     if(PIFJ&amp;0x40){ // poll PJ6         PIFJ = 0x40; // acknowledge SW2         asm cli         Task2(); // service SW2     } } </pre>
--	--

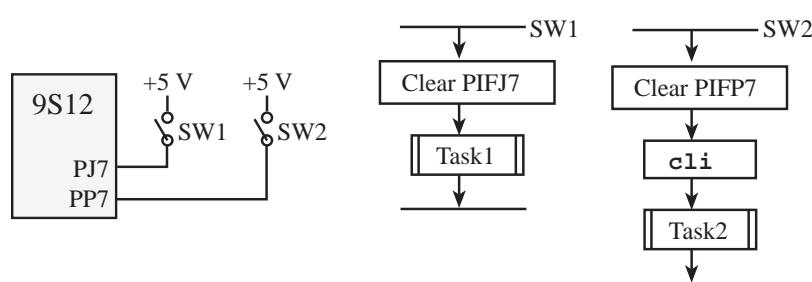
### Program 4.26

Polled solution with SW1 having priority over SW2.

One of the disadvantages of the polled implementation is that it takes some time to poll. If we wish to reduce the latency, then two separate vectors can be used, as shown in Figure 4.30. The vectored solution in Program 4.27 is similar to Program 4.18. The key wake-up trigger PIFJ7 is set on the rising edge of PJ7, and the key wake-up trigger PIFP7 is set on the rising edge of PP7. Port J has priority over Port P, because its interrupt number is smaller. Therefore, if SW1 and SW2 occur simultaneously, SW1 is serviced first. We have saved time by removing the polling. In this solution, if SW1 comes right after SW2, the latency for servicing SW1 in Program 4.27 is shorter than Program 4.26. This is because the time running the SW2 handler with interrupts disabled is less. Specifically, the `movb` and `cli` instructions in the SW2 handler only require five bus cycles to execute.

**Figure 4.30**

A vectored interrupt where Switch SW1 has higher priority than Switch SW2.



```

Init bclr DDRJ,#$80 ;PJ7=SW1
    bset PERJ,#$80 ;pull down PJ7
    bset PPSJ,#$80 ;rise on PJ7
    bset PIEJ,#$80 ;arm PJ7
    bclr DDRP,#$80 ;PP7=SW2
    bset PERP,#$80 ;pull down PP7
    bset PPSP,#$80 ;rise on PP7
    bset PIEP,#$80 ;arm PP7
    cli
    rts
KeyHanJ movb #$80,PIFJ ;acknowledge
    jsr Task1      ;service SW1
    rti
KeyHanP movb #$80,PIFP ;acknowledge
    cli
    jsr Task2      ;service SW2
    rti
    org $FFCE   ; Key Wakeup J
    fdb KeyHanJ
    org $FF8E   ;Key Wakeup P
    fdb KeyHanP

void Init(void){
    DDRJ &= ~0x80; // PJ7=SW1
    PERJ |= 0x80; // pull down PJ7
    PPSJ |= 0x80; // rise on PJ7
    PIEJ |= 0x80; // arm PJ7
    DDRP &= ~0x80; // PP7=SW2
    PERP |= 0x80; // pull down PP7
    PPSP |= 0x80; // rise on PP7
    PIEP |= 0x80; // arm PP7
    asm cli
}

void interrupt 24 KeyHanJ(void){
    PIFJ = 0x80; // acknowledge
    Task1();      // service SW1
}
void interrupt 56 KeyHanP(void){
    PIFP = 0x80; // acknowledge
    asm cli
    Task2();      // service SW2
}

```

**Program 4.27**

Vectored solution with SW1 having priority over SW2.

## 4.13 Round-Robin Polling

Rather than implement priority, sometimes we wish to implement *no priority*. This means we can guarantee service under heavy load for equally important devices. For interrupts that occur on an infrequent basis, priority doesn't really matter. But as the number of interrupts increase and the interrupt request rates increase, we may wish to use round-robin polling. This implementation works only for polled interrupts and does not apply to vectored interrupts. Two round-robin schemes are shown for a situation with three devices called A, B, C. The particular example sequence of events is shown for the situation that B and C always want service and A never does.

Scheme 1 Start polling after device that last got service. Example sequence of events:

Interrupt, poll A, B, C (B, C need service)

Service B

Interrupt, poll C, A, B (B, C need service)

Service C

Interrupt, poll A, B, C (B, C need service)

Service B

Interrupt, poll C, A, B

Scheme 2 Cycle through list of devices independent of which device last got service.

Example sequence of events:

Interrupt, poll A, B, C

Interrupt, poll B, C, A

Interrupt, poll C, A, B

Interrupt, poll A, B, C, etc.

Our previous polling examples in Section 4.11 can be modified to implement either round-robin scheme. Program 4.28 implements the second simple method that rotates the polling order independent of which devices request service.

```

NodeType sys[3]={
    {0x04, PJ2Han, &sys[1]},
    {0x02, PJ1Han, &sys[2]},
    {0x01, PJ0Han, &sys[0]} };
NodePtr Pt=&sys[0]; // points to the one that got polled first at last interrupt
void interrupt 24 IRQHan(void){ unsigned char Counter,Status;
    Counter=3; // quit after three devices checked
    Pt=Pt->NextPt; // rotates ABC BCA CAB polling orders
    while(Counter--){
        if(PIFJ&(Pt->Mask)){
            (*Pt->Handler)(); /* Execute handler */
            Pt=Pt->NextPt; } } // returns after all devices have been polled

```

**Program 4.28**

C language implementation of round-robin polling on the 9S12.

## 4.14 Periodic Interrupts

The purpose of this section is to present alternative methods to create a real-time periodic interrupt (RTI). A RTI is one that is requested on a fixed time basis. This interfacing technique is required for data acquisition and control systems, because software servicing must be performed at accurate time intervals. For a data acquisition system, it is important to establish an accurate sampling rate. The time in between ADC samples must be equal (and known) for the digital signal processing to function properly. Similarly for microcomputer-based control systems, it is important to maintain both the ADC and DAC timing.

Another application of RTIs is called *intermittent* or *periodic polling*. In regular busy-waiting the main program polls the I/O devices continuously (refer back to Section 3.3). With intermittent polling, the I/O devices are polled on a regular basis (established by the RTI). If no device needs service, then the interrupt simply returns. This method frees the main program from the I/O tasks. We use periodic polling (Figure 4.31) if the following two conditions apply:

1. The I/O hardware cannot generate interrupts directly.
2. We wish to perform the I/O functions in the background.

In each of the examples of this section, we will design a real-time periodic interrupt software system that increments a global variable. It is real-time because we can put an upper bound on the latency, which is the time between when the software task is supposed to run (in this case at fixed time intervals), and when the ISR increments the global variable Time. In general, the latency of interrupt-driven tasks includes

1. the maximum time the system runs with interrupts disabled
2. the time to finish the current instruction in the main program
3. the time to perform the thread switch (push registers, fetch interrupt vector)
4. the time to execute the interrupt service routine

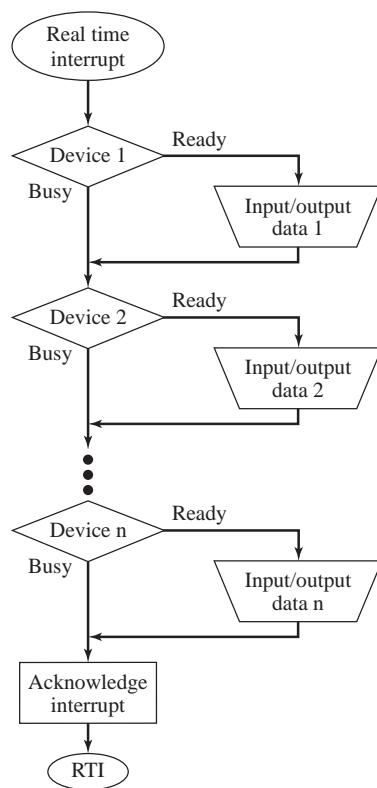
The initialization program will arm the device and establish the periodic interrupt rate. A flag will be set by the timer hardware at a periodic rate. Because the flag is armed, an interrupt will be requested each time the flag is set. The interrupt service routine will acknowledge the interrupt by clearing the flag, then it will increment the global variable.

**Checkpoint 4.10:** In a typical system, which of the four components of latency is largest?

**Observation:** The TCNT timer is very accurate because of the stability of the crystal clock.

**Figure 4.31**

An ISR flowchart that implements periodic polling.



There are three mechanisms on the 9S12 that generate periodic interrupts: real-time interrupt, timer overflow, and output compare. Table 4.5 shows the 9S12 registers used in periodic interrupts. The entries shown in bold will be used in this section.

Address	msb																lsb	Name
\$0044	<b>1</b>	<b>5</b>	<b>1</b>	<b>4</b>	<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>98</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	TCNT
\$0050	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC0	
\$0052	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC1	
\$0054	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC2	
\$0056	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC3	
\$0058	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC4	
\$005A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC5	
\$005C	<b>1</b>	<b>5</b>	<b>1</b>	<b>4</b>	<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>98</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	TC6
\$005E	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC7	

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0037	<b>RTIF</b>	PROF	0	LOCKIF	LOCK	TRACK	SCMIF	SCM	CRGFLG
\$0038	<b>RTIE</b>	0	0	LOCKIE	0	0	SCMIE	0	CRGINT
\$003B	0	<b>RTR6</b>	<b>RTR5</b>	<b>RTR4</b>	<b>RTR3</b>	<b>RTR2</b>	<b>RTR1</b>	<b>RTR0</b>	RTICTL
\$0046	<b>TEN</b>	TSWAI	TSBCK	TFFCA	0	0	0	0	TSCR1
\$004D	<b>TOI</b>	0	0	0	TCRE	<b>PR2</b>	<b>PR1</b>	<b>PR0</b>	TSCR2
\$0040	IOS7	<b>IOS6</b>	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0	TIOS
\$004C	C7I	<b>C6I</b>	C5I	C4I	C3I	C2I	C1I	COI	TIE
\$004E	C7F	<b>C6F</b>	C5F	C4F	C3F	C2F	C1F	COF	TFLG1
\$004F	<b>TOF</b>	0	0	0	0	0	0	0	TFLG2

**Table 4.5**

9S12 registers used to configure periodic interrupts.

First, the real-time interrupt mechanism can generate interrupts at a fixed rate. Seven bits (RTR6-0) in the RTICTL register (\$003B) specify the interrupt rate. The 7-bit value is composed of two parts:

Let **RTR6**, **RTR5**, **RTR4** be **n**, which is a 3-bit number ranging from 0 to 7.

Let **RTR3**, **RTR2**, **RTR1**, **RTR0** be **m**, which is a 4-bit number ranging from 0 to 15.

If **n** is zero, then the RTI system is off. A 9S12 with an 8 MHz crystal will have an OSCCLK frequency of 8 MHz and a default E clock frequency of 4 MHz. Table 4.6 shows the available interrupt periods, assuming an 8 MHz crystal. A 9S12 with a 16 MHz crystal will have an OSCCLK frequency of 16 MHz and a default E clock frequency of 8 MHz. Table 4.7 shows the available interrupt periods, assuming a 16 MHz crystal.

n [6:4] of the RTICTL									
	<b>000</b>	<b>001</b>	<b>010</b>	<b>011</b>	<b>100</b>	<b>101</b>	<b>110</b>	<b>111</b>	
<b>m</b> [3:0]	0000	off	0.128	0.256	0.512	1.024	2.048	4.096	8.192
	0001	off	0.256	0.512	1.024	2.048	4.096	8.192	16.384
	0010	off	0.384	0.768	1.536	3.072	6.144	12.288	24.576
	0011	off	0.512	1.024	2.048	4.096	8.192	16.384	<b>32.768</b>
	0100	off	0.640	1.280	2.560	5.120	10.240	20.480	40.960
	0101	off	0.768	1.536	3.072	6.144	12.288	24.576	49.152
	0110	off	0.896	1.792	3.584	7.168	14.336	28.672	57.344
	0111	off	1.024	2.048	4.096	8.192	16.384	32.768	65.536
	1000	off	1.152	2.304	4.608	9.216	18.432	36.864	73.728
	1001	off	1.280	2.560	5.120	10.240	20.480	40.960	81.920
	1010	off	1.408	2.816	5.632	11.264	22.528	45.056	90.112
	1011	off	1.536	3.072	6.144	12.288	24.576	49.152	98.304
	1100	off	1.664	3.328	6.656	13.312	26.624	53.248	106.496
	1101	off	1.792	3.584	7.168	14.336	28.672	57.344	114.688
	1110	off	1.920	3.840	7.680	15.360	30.720	61.440	122.880
	1111	off	2.048	4.096	8.192	16.384	32.768	65.536	131.072

**Table 4.6**

9S12 real-time interrupt period in ms assuming an 8 MHz crystal.

n [6:4] of the RTICTL									
	<b>000</b>	<b>001</b>	<b>010</b>	<b>011</b>	<b>100</b>	<b>101</b>	<b>110</b>	<b>111</b>	
<b>m</b> [3:0]	0000	off	0.064	0.128	0.256	0.512	1.024	2.048	4.096
	0001	off	0.128	0.256	0.512	1.024	2.048	4.096	8.192
	0010	off	0.192	0.384	0.768	1.536	3.072	6.144	12.288
	0011	off	0.256	0.512	1.024	2.048	4.096	8.192	16.384
	0100	off	0.320	0.640	1.280	2.560	5.120	10.240	20.480
	0101	off	0.384	0.768	1.536	3.072	6.144	12.288	24.576
	0110	off	0.448	0.896	1.792	3.584	7.168	14.336	28.672
	0111	off	0.512	1.024	2.048	4.096	8.192	16.384	<b>32.768</b>
	1000	off	0.576	1.152	2.304	4.608	9.216	18.432	36.864
	1001	off	0.640	1.280	2.560	5.120	10.240	20.480	40.960
	1010	off	0.704	1.408	2.816	5.632	11.264	22.528	45.056
	1011	off	0.768	1.536	3.072	6.144	12.288	24.576	49.152
	1100	off	0.832	1.664	3.328	6.656	13.312	26.624	53.248
	1101	off	0.896	1.792	3.584	7.168	14.336	28.672	57.344
	1110	off	0.960	1.920	3.840	7.680	15.360	30.720	61.440
	1111	off	1.024	2.048	4.096	8.192	16.384	32.768	65.536

**Table 4.7**

9S12 real-time interrupt period in ms, assuming a 16 MHz crystal.

To clear the RTIF flag (acknowledge the interrupt), the software writes a one to it. Program 4.29 will increment a global variable, Time, every 32.767 ms. The interrupt rate is determined by the crystal clock and the RTICTL value

$$\text{RTI interrupt frequency (Hz)} = 15625 * 2^n / (\text{m} + 1)$$

$$\text{RTI interrupt period (ms)} = 0.064 * (\text{m} + 1) * 2^n, \text{ assuming an 8 MHz crystal}$$

**Observation:** The phase-lock-loop (PLL) on the 9S12 will not affect the RTI rates.

**Checkpoint 4.11:** How would you modify Program 4.29 to count at 61.035 Hz? (there are four answers).

<pre>; MC9S12C32 assembly, 8 MHz crystal         org \$3800 ;RAM Time      rmb 2         org \$4000 RTI_Init sei           ;make atomic         movb #\$73,RTICTL ;32.768ms         movb #\$80,CRGINT ;arm RTI         movw #0,Time         cli            ;enable IRQ         rts RTIHan   movb #\$80,CRGFLG    ;ack         ldd  Time         addd #1         std  Time         rti         org \$FFFF0         fdb  RTIHan ;vector</pre>	<pre>// MC9S12C32 CodeWarrior C unsigned short Time; void RTI_Init(void){     asm sei           // Make atomic     RTICTL = 0x73;    // 30.517Hz     CRGINT = 0x80;    // Arm     Time = 0;         // Initialize     asm cli }  void interrupt 7 RTIHan(void){     CRGFLG = 0x80;    // Acknowledge     Time++; }</pre>
---	--

### Program 4.29

Implementation of a periodic interrupt using the real-time clock feature.

The timer overflow interrupt feature also can be used to generate interrupts at a fixed rate, as listed in Table 4.8. The 16-bit TCNT register is incremented at a fixed rate. The TOF trigger flag is set when the counter overflows and wraps back around (automatically) to zero. If armed, the TOF trigger flag will generate an interrupt. Three bits (PR2, PR1, and PR0) in the TSCR2 register determine the rate at which the counter will increment, hence they will determine the TOF interrupt rate. Program 4.30 shows a 30.517-Hz periodic ISR

PR2	PR1	PR0	Divide by	E = 4 MHz		E = 8 MHz		E = 24 MHz	
				TCNT Period	TOF Period	TCNT Period	TOF Period	TCNT Period	TOF Period
0	0	0	1	250 ns	16.384 ms	125 ns	8.192 ms	42 ns	2.73067 ms
0	0	1	2	500 ns	32.768 ms	250 ns	16.384 ms	83 ns	5.46133 ms
0	1	0	4	1 μs	65.536 ms	500 ns	32.768 ms	167 ns	10.9227 ms
0	1	1	8	2 μs	131.072 ms	1 μs	65.536 ms	333 ns	21.8453 ms
1	0	0	16	4 μs	262.144 ms	2 μs	131.072 ms	667 ns	43.6907 ms
1	0	1	32	8 μs	524.288 ms	4 μs	262.144 ms	1.33 μs	87.3813 ms
1	1	0	64	16 μs	1048.576 ms	8 μs	524.288 ms	2.67 μs	174.763 ms
1	1	1	128	32 μs	2097.152 ms	16 μs	1048.576 ms	5.33 μs	349.525 ms

**Table 4.8**

Timer overflow periods for various E clock frequencies.

<pre>; 9S12 assembly     org \$3800 ;RAM Time    rmb 2     org \$4000 TOF_Init sei      ;make atomic     movb #\$80,TSCR1 ;enable TCNT     movb #\$81,TSCR2 ;arm, 32.768ms     movw #0,Time     cli       ;enable IRQ     rts TOFHan movb #\$80,TFLG2 ;acknowledge     ldd Time     addd #1     std Time     rti     org \$FFDE     fdb TOFHan ;vector</pre>	<pre>// 9S12 CodeWarrior C unsigned short Time; void TOF_Init(void){     asm sei      // Make atomic     TSCR1 = 0x80; // enable counter     TSCR2 = 0x81; // Arm, 30.517Hz     Time = 0;     // Initialize     asm cli      // enable interrupts }  interrupt 16 void TOFHan(void){     TFLG2 = 0x80; // Acknowledge     Time++; }</pre>
--	---

**Program 4.30**

Implementation of a periodic interrupt using timer overflow.

implemented with TOF. To clear the TOF flag (acknowledge the interrupt), the software writes a 1 to it. To create a TOF periodic interrupt, we enable the timer (TEN=1), arm the timer overflow (TOI), and set the rate (PR2-0). Let  $n$  be the 3-bit number (0 to 7) formed from the least significant three bits of TSCR2. Let  $f_E$  be the frequency of the E clock (adjusted by the PLL). The TOF interrupt rate is

$$\text{TOF interrupt frequency} = f_E / 2^{(n+16)}$$

$$\text{TOF interrupt period} = 2^{(n+16)} / f_E$$

**Observation:** The phase-lock-loop (PLL) on the 9S12 will affect the TOF and output compare rates.

The third mechanism to generate periodic interrupts is output compare. There are eight independent output compare channels numbered 0 to 7. Let  $i$  be the channel number,  $0 \leq i \leq 7$ . To enable output compare, the corresponding bit in the TIOS register must be set. When the TCNT register matches  $TC_i$ , the output compare flag,  $C_{iF}$  is set. If armed ( $C_{iI}=1$ ), then it will request an interrupt. To clear the  $C_{iF}$  flag (acknowledge the interrupt), the software writes a one to it. The ISR will acknowledge the interrupt and set  $TC_i = TC_i + \text{PERIOD}$ , where PERIOD is a value specifying the time duration to the next interrupt. The interrupting period is determined by the TCNT period (set by TSCR2) multiplied by the constant PERIOD. Let  $n$  be the 3-bit number (0 to 7) formed from the least significant three bits of TSCR2. Let  $f_E$  be the frequency of the E clock (adjusted by the PLL). The output compare interrupt rate is

$$\text{OC interrupt frequency} = f_E / 2^n / \text{PERIOD}$$

$$\text{OC interrupt period} = \text{PERIOD} * 2^n / f_E$$

TCTL1 and TCTL2 registers are also used for output compare. If  $OM_n=OL_n=0$ , then an output compare event will not directly affect the output pin. If the pair ( $OM_n, OL_n$ ) equals (0,1), then the output pin will toggle on each output compare. If the pair ( $OM_n, OL_n$ ) equals (1,0), then the output pin will clear to low on each output compare. If the pair ( $OM_n, OL_n$ ) equals (1,1), then the output pin will set to high on each output compare. Program 4.31 shows a 1-kHz periodic interrupt using output compare 6.

**Performance Tip:** When using output compare interrupts, if you do not need the associated PTT pin as part of the system, you can configure it as an output and set (OMn=0,OLn=1), creating a heartbeat with no software overhead (i.e., it will be completely nonintrusive).

A real-time system is one with a small and bounded latency. For real-time systems with periodic tasks, we define *time-jitter*,  $\tau$ , which is the difference between when a periodic task is supposed to be run and when it actually is run. The goal of most periodic systems is to start the task at a periodic rate,  $\frac{1}{T}$ . Let  $t_n$  be the nth time the task is started. In particular, the goal is to make  $t_n - t_{n-1} = \frac{1}{T}$ . The jitter is defined as the constant,  $\tau$ , such that

$$t_n - t_{n-1} < \tau \quad \text{for all } i$$

We will classify the system as in real time if  $\tau$  is small and bounded.

**Checkpoint 4.12:** How would you modify Program 4.31 to count at 1 Hz?

<pre>; 9S12 assembly, 4 MHz E clock PERIOD equ 1000 ;in usec           org \$3800 ;RAM Time    rmb 2           org \$4000 OC6_Init sei      ;make atomic           movb #\$80,TSCR1 ;enable TCNT           movb #\$02,TSCR2 ;luss           bset TIOS,\$#40 ;activate OC6           bset TIE,\$#40   ;arm OC6           movw #0,Time           ldd  TCNT   ;time now           addd #50     ;first in 50us           std   TC6           cli      ;enable IRQ           rts OC6Han  movb #\$40,TFLG1 ;acknowledge           ldd  TC6           addd #PERIOD           std   TC6   ;next in 1 ms           ldd  Time           addd #1           std   Time           rti           org  \$FFE2           fdb  OC6Han ;vector</pre>	<pre>// 9S12 CodeWarrior C, 4 MHz E clock #define PERIOD 1000 unsigned short Time;  void OC6_Init(void){     asm sei      // Make atomic     TSCR1 = 0x80;     TSCR2 = 0x02; // 1 MHz TCNT     TIOS  = 0x40; // activate OC6     TIE  = 0x40; // arm OC6     TC6 = TCNT+50; // first in 50us     Time = 0;      // Initialize     asm cli      // enable IRQ }  interrupt 14 void OC6handler(void){     TC6 = TC6+PERIOD; // next in 1 ms     TFLG1 = 0x40;     // acknowledge C6F     Time++; }</pre>
--	--

### Program 4.31

Implementation of a periodic interrupt using output compare.

## 4.15 Low Power Design

Reducing the amount of power used by an embedded system will save money and extend battery life. In CMOS digital logic, power is required to make signals rise or fall. Most microcontrollers allow you to adjust the frequency of the bus clock. The 9S12 allows you to change the E clock using the PLL, previously described in Section 1.8. Selecting the E clock frequency is a tradeoff between power and performance. To optimize for power, we choose the slowest E clock frequency that satisfies the minimum requirements of the

system. When we implement the software system with interrupts, it allows us to focus the CPU on executing tasks when they need to run. Because there are fewer backward jumps wasting time, interrupt-driven systems typically will be able to perform the same functions at a slower bus clock.

A second factor in low power design follows the axiom “turn the light off when you leave the room.” Basically, we turn off devices when they are not being used. Most I/O devices on the 9S12 are initialized as off, so we have to turn them on to use them. However, rather than turning it on once in the ritual and leaving it on continuously, we could dynamically turn it on when needed then turn it off when done. In Chapter 11, we will learn ways to turn off external analog circuits when they are not needed.

In general, as we reduce the voltage, the power is reduced. Most 9S12 families have versions that will run at 3.3 V. If a device has a fixed resistance from supply voltage to ground, reducing from 5 to 3.3 V will drop the power by a factor of  $(5^2 - 3.3^2)/5^2 = 56\%$ . Reducing the voltage on some devices will slow them down, as investigated in Exercise 1.5.

The output ports of the 9S12 have a reduced drive mode. The output pin will operate slower but will draw less current when in reduced drive. Any unused input pins should have internal or external pull-up (or pull-down). An input pin not connected to anything may oscillate at a frequency of an external field, wasting power unnecessarily.

One way to save power is to perform all operations as background tasks and put the processor to sleep when in the foreground. The `wai` instruction stops the bus clock, and the processor stops executing instructions. An interrupt will wake up the sleeping processor. A typical 9S12 consumes up to 35 mA while running at 24 MHz. However, in sleep mode with RTI active and the PLL not active, the supply current drops to somewhere between 2.5 and 8 mA, depending on what else is active. To illustrate this approach, consider the two systems in Program 4.32. Both systems execute `Stuff` 30.517 times a second. Measurements taken on a Technological Arts NC12C32 Nanocore12 (see Figure 1.34) running at 4 MHz show the software on the left requires 8 mA to run and the software on the right requires only 6 mA to run.

<pre> unsigned short Time; void main(void){     asm sei          // Make atomic     RTICTL = 0x73; // 30.517Hz     CRGINT = 0x80; // Arm     Time = 0;        // Initialize     asm cli     while(1){     } }  void interrupt 7 RTIHan(void){     CRGFLG = 0x80; // Acknowledge     Time++;     Stuff(); } </pre>	<pre> unsigned short Time; void main(void){     asm sei          // Make atomic     RTICTL = 0x73; // 30.517Hz     CRGINT = 0x80; // Arm     Time = 0;        // Initialize     asm cli     while(1){         asm wai     } }  void interrupt 7 RTIHan(void){     CRGFLG = 0x80; // Acknowledge     Time++;     Stuff(); } </pre>
---	---

### Program 4.32

Example showing how to save power by putting the processor to sleep.

**Performance tip:** Whenever your software performs a backward jump (e.g., waiting for an event), it may be possible to put the processor to sleep, thus saving power.

**Observation:** The Texas Instruments MSP430 family of microcontrollers requires a lot less power than the 9S12 family, and have many low-power features.

## 4.16 Exercises

**4.1** Syntactically, I/O ports are public globals. In order to separate mechanisms from policies (i.e., improve the quality of the software system), how should I/O be actually used?

- a) Local in allocation
- b) Global in allocation
- c) Public in scope
- d) Private in scope
- e) Volatile
- f) Nonvolatile

**4.2** Why do we add the `volatile` qualifier in all I/O port definitions? For example,

```
#define TCNT *(unsigned short volatile *)(0 0044)
```

**4.3** What happens if an interrupt service routine does not acknowledge or disarm?

- a) Software crashes because no more interrupts will be requested.
- b) The next interrupt is lost.
- c) This interrupt is lost.
- d) Software crashes because interrupts are requested over and over.

**4.4** The main program synthesizes data and a periodic output-compare interrupt will output the data separated by a fixed time. A FIFO queue is used to buffer data between a main program (e.g., main program calls `Fifo_Put`). An output-compare interrupt service routine calls `Fifo_Get` and actually outputs. Experimental observations show this FIFO is usually empty and has at most three elements. What does it mean? Choose from the following.

- a) The system is CPU bound.
- b) Bandwidth could be increased by increasing FIFO size.
- c) The system is I/O bound.
- d) The FIFO could be replaced by a global variable.
- e) The latency is small and bounded.
- f) Interrupts are not needed in this system.

**4.5** Answer Exercise 4.4 under the condition that the FIFO often becomes full.

**4.6** A key wake-up input is armed so that interrupts occur when new data arrives into the 9S12. Consider the situation in which a FIFO queue is used to buffer data between the key wake-up ISR and the main program. The ISR inputs and saves the data by calling `Fifo_Put`. When the main program wants input, it calls `Fifo_Get`. Experimental observations show this FIFO is usually empty, and has at most three elements. What does it mean? Choose from the following.

- a) The system is CPU bound.
- b) Bandwidth could be increased by increasing FIFO size.
- c) The system is I/O bound.
- d) The FIFO could be replaced by a global variable.
- e) The latency is small and bounded.
- f) Interrupts are not needed in this system.

**4.7** Answer Exercise 4.6 under the condition that the FIFO often becomes full.

**4.8** Consider the following interrupting system. The active-edge inputs on PJ7 and PP7 can occur at any time, including at the same time. The object is to count the number of each type of interrupt.

```
void interrupt 24 KeyHanJ(void){
    unsigned short static Count=0;
    PIFJ = 0x80; // acknowledge
    Count++;      // count the number of PJ7 edges
}
void interrupt 56 KeyHanP(void){
    unsigned short static Count=0;
    PIPF = 0x80; // acknowledge
    Count++;      // count the number of PP7 edges
}
```

How would you best describe the usage of Count in this system?

- a) This is a perfectly appropriate usage of Count, because there are two permanently allocated variables with private scope, such that each variable counts the number of interrupts for each ISR.
- b) There is a critical section bug because of the read/modify/write access to a shared global.
- c) Because both ISRs share the same Count, the system can not distinguish between a PJ7 and a PP7 interrupt.
- d) Count is initialized each interrupt, so its value is not the total number.
- e) The acknowledge statements in the two ISRs are not friendly because they affect all 8 bits of the flag register.
- f) None of the above is true.

**4.9** Consider the following interrupt service routine. The goal is to measure the elapsed time from one interrupt call to the other. What qualifier do you place in the xx position to make this measurement operational? Choose from volatile static float const or public.

```
unsigned short Elapsed; // time between interrupt
void interrupt 20 handler(void){
    xx unsigned short last=0;
    Elapsed = TCNT-last;
    last = TCNT;
}
```

**4.10** What purpose might there be to use the PLL and slow down the 9S12?

- a) The system is CPU bound.
- b) To make the batteries last longer on a battery-powered system.
- c) In order to adjust the interrupt period when using RTI interrupts to a convenient value.
- d) In order to balance the load between foreground and background threads.
- e) To reduce latency.
- f) None of the above, because there is never a reason to run slower.

**4.11** This is a *functional debugging* question. However, the debugging instrument still needs to be *minimally intrusive*. Assume  $y=Function(x)$  is a function with 16-bit input  $x$  and 16-bit output  $y$  and is called from an ISR as part of a real-time system. The SCI, PTT, and PTP are unused by the system, and PTT and PTP are digital outputs. The debugging code will be placed at the end just before the return, unless otherwise stated. SCI\_OutSDec outputs a 16-bit signed integer. BufX and BufY are 16-bit signed global buffers of length 100, and n is a global variable initialized to 0. Which debugging code would you add to verify the correctness of this function?

- a) PTT=x; PTP=y;
- b) SCI\_OutSDec(x); SCI\_OutSDec(y); // busy-wait
- c) if(n<100){BufX[n]=x; BufY[n]=y; n++;}
- d) PTT |= 0x01; // at beginning  
PTT &= ~0x01; // at end
- e) if(n<100){BufX[n]=x; BufY[n]=TCNT; n++;}

**4.12** Three events must occur for a key wakeup interrupt on PP5 to be generated:

1. Software sets the arm bit (PIEP bit 5).
2. Software enables interrupts ( $I=0$ ) by executing cli.
3. Hardware sets the flag bit (PIFP bit 5).

Which time sequence of these three events cause the interrupt to be generated?

- a) Only the order 1,2,3
- b) Only the order 3,2,1
- c) Only the order 2,1,3
- d) Any order will generate an interrupt

- 4.13** The following multithreaded system has a critical section. Modify these programs to remove the error. You may assume the RTI interrupts are enabled and are running. The **unsigned long** data type is a 32-bit integer.

```
unsigned long Time;
void main(void){
    while(1){
        if(Time == 1000000){
            SCI_OutString("done");
        }
    }
}
```

```
void interrupt 7 handler(){
    PTT |= 0x01;
    CRGFLG = 0x80; // ack
    Time++;
    PTT &= ~0x01;
}
```

- 4.14** The following multithreaded system has a shared global variable. Is there a critical section? If yes, edit the code to remove the critical section. If no, justify your answer in 16 words or less.

```
unsigned short Time;
void interrupt 9 TC1han(){
    PTT |= 0x02;
    TFLG1 = 0x02;
    TC1 = TC1+1000
    Time++;
    PTT &= ~0x02;
}
```

```
void interrupt 8 TC0han(){
    PTT |= 0x01;
    TFLG1 = 0x01;
    TC0 = TC0+1000
    Time++;
    PTT &= ~0x01;
}
```

- 4.15** What happens if two interrupt requests are made during the same instruction? Is one lost? If both are serviced, which one goes first?

- 4.16** Specify whether each statement is TRUE or FALSE.

- A **signed char** variable can store values from -128 to +128.
- Consider an **input** device. The **interface latency** is the time from when the software asks for new data until the time new data are ready.
- Consider an **output** device. The **interface latency** is the time from when the software sends new data to the output device until the time the output operation is complete.
- The **static** qualifier is used with functions to specify the function is permanent, created at compile time and is never destroyed. For example,

```
static short AddTwo(short in){ return in+2; }
```

- The **static** qualifier is used with a variable defined inside a function to specify the variable is permanent, created at compile time, initialized to 0, and is never destroyed. For example,

```
void function(short in){ static short myData;
```

- The **const** qualifier is used with a global variable to specify the variable should be allocated in ROM. For example, **const** short myData=5;
- Code that is **friendly** means it can be executed by more than one thread without causing a crash or loss of data.
- The **volatile** qualifier is used with variables to tell the compiler that code that accesses this variable should be optimized as much as possible.
- A read-modify-write access to a shared global variable always creates a **critical** section.
- The compiler automatically places an **sei** instruction at the beginning of the **interrupt service routine** and a **cli** at the end so that the computer runs with interrupts disabled while servicing the interrupt.

- 4.17** Consider the following TOF interrupting system with its corresponding assembly code generated by the compiler. Are there any critical sections in the software system? If so, state the location of the critical section, and changes required to remove it. This is all the software.

```

unsigned short time;
interrupt 16 void TOFhan(void){
    time++;
    TFLG2 = 0x80;
}
void TOFinit(void){
    time = 0;
    TSCR1 = 0x80;
    TSCR2 = 0x85;
asm cli
}
short calc(short a, short b){
    return a*b;
}
void main(void){  short c;
    TOFinit();
    c = 1;
    for(;;) {
        c = calc(c,time);
    }
}

TOFhan:
    ldx  time
    inx
    stx  time
    movb #128,TFLG2
    rti
TOFinit:
    movw #0,time
    movb #128,TSCR1
    movb #133,TSCR2
    cli
   rts
calc: ldy  2,SP
    emul
    rts
main: pshd
    bsr  TOFinit
    ldd  #1
    std  0,SP
loop pshd
    idaa time
    ldab time+1
    bsr  calc
    leas 2,SP
    bra  loop

```

- 4.18** Consider this `Heap_Release` function, which is part of a dynamic memory manager. The function exists at ROM locations \$4113 to \$411F, and the 16-bit `FreePt` is allocated in RAM at location \$3852.

<pre> unsigned short static Heap[SIZE*NUM]; unsigned short static *FreePt; void Heap_Release(unsigned short *pt){     unsigned short *oldFreePt;     oldFreePt = FreePt;     FreePt = pt;     *pt = (unsigned short)oldFreePt; } </pre>	<code>Heap_Release</code> <pre> 4113 3b      PSHD 4114 fc3852   LDD   FreePt 4117 ee80     LDX   0,SP 4119 7e3852   STX   FreePt 411c 6c00     STD   0,X 411e 3a       PULD 411f 3d       RTS </pre>
---	--

Is there a critical section in the `Heap_Release` function? If so, specify the exact range of 16-bit hexadecimal addresses between which the critical section exists. For example, if you say there is a critical section between \$4113 and \$4117, then you are specifying an interrupt occurring between the `PSHD` and `LDD` instructions or between the `LDD` and `LDX` instructions could result in a corrupt heap, but interrupts occurring at other places in the function will be okay.

- 4.19** Five events occur in this order.

Step 1. The I bit in the CCR is set to one.

Step 2. Output compare interrupt 5 is armed.

Step 3. The TCNT matches TC5 setting C5F.

Step 4. The I bit in the CCR is cleared to zero.

Step 5. The TCNT counts all the way around and matches TC5 again setting C5F.

When is the first interrupt?

- |   |  |
|---|--|
| <b>a)</b> No interrupt occurs<br><b>b)</b> After Step 1<br><b>c)</b> After Step 2 | <b>d)</b> After Step 3<br><b>e)</b> After Step 4<br><b>f)</b> After Step 5 |
|---|--|

- 4.20** The PLL is not active, and the E clock frequency is 8 MHz. The TCNT timer is active with TSCR2 equal to 2. The following code occurs in the output-compare 7 ISR

```
TC7 = TC7 + 100;
```

What is the time period between output compare 7 interrupts?

- 4.21** The PLL is active, and the E clock frequency is 24 MHz. The TCNT timer is active with TSCR2 equal to 5. The following code occurs in the output-compare 6 ISR

```
TC6 = TC6 + 1000;
```

What is the time period between output compare 6 interrupts?

- 4.22** These seven events all occur during each output compare 5 interrupt. Order these events into a proper time sequence. More than one answer may be correct.

- The TCNT equals **TC5** and the hardware sets the flag bit (e.g., C5F=1)
- The output compare 5 vector address is loaded into the PC
- The I bit in the CCR is set by hardware
- The software executes `movb #$20, TFLG1`
- The CCR, A, B, X, Y, PC are pushed on the stack
- The software executes something like

```
ldd TC5
addd #1000
std TC5
```

- The software executes `rti`

- 4.23** Consider output compare 0 interrupts. Assume the name of the interrupt service routine is `TC0Handler`.

- What three events in general need to be true for any interrupt to occur? Furthermore, give those three events specifically for output compare 0.
- List the events that occur as the computer switches from running in the foreground to running an output-compare 0 interrupt in the background? For Example, the 9S12 is running the main program, *stuff happens*, the 9S12 is running `TC0Handler`. List the events in *stuff happens*.
- Write assembly or C code to acknowledge an output compare 0 interrupt.

- D4.24** Design and implement a FIFO that can hold up to four elements. Each element is 3 bytes. There will be three subroutines: `Init`, `Put` one element into FIFO, and `Get` one element from the FIFO. Show the private RAM-based variables. Write a function that initializes the FIFO. Write a function that puts one 3-byte element into the FIFO. Write a function that gets one 3-byte element from the FIFO. Document how parameters are passed in each function. You can use the FIFO in Program 4.14 as a starting point for parameter passing, but some changes to how parameters are passed will be required for both `Get` and `Put`.

- D4.25** Solve the FSM described in Exercise D2.24 using output compare 0 interrupts. Include three components: the FSM structure, an initialization function, and an output compare 0 ISR. There should be no backwards jumps.

- D4.26** Solve the FSM described in Exercise D2.24 using output compare 2 interrupts. Run one pass through the FSM every 10 ms. Include three components: the FSM structure, an initialization function, and an output-compare 2 ISR. There should be no backwards jumps.

- D4.27** Solve the traffic light FSM described in Example 2.1 using output compare 3 interrupts. Run the FSM in the ISR. Include three components: the FSM structure, an initialization function, and an output-compare 3 ISR. There should be no backwards jumps.

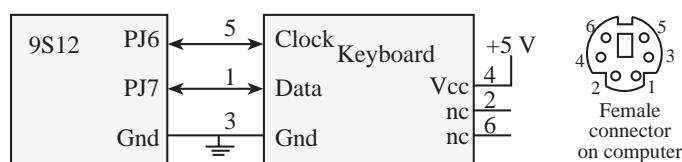
- D4.28** Redesign the receiver portion of Example 3.5 so the input occurs in the background using key wake-up interrupts. You will first interrupt on the rising edge of **Ready** and then interrupt on the falling edge of **Ready**. In this way, there will be no backwards jumps in the ISR. Pass the data from background to foreground using the FIFO module of Program 4.14. Rewrite the `Init` function so that it arms and enables key wake-up interrupts and calls `Fifo_Init`. Your

ISR will perform the actual input and call `Fifo_Put`. Rewrite the `In` function to call `Fifo_Get` over and over until it gets data.

**D4.29** Redesign the transmitter portion of Example 3.5 so the output occurs in the background using key wake-up interrupts. You will interrupt on the rising edge of `Ack`. There will be no backwards jumps in the ISR. Pass the data from foreground to background using the FIFO module of Program 4.14. Rewrite the `Init` function so that it arms and enables key wake-up interrupts and calls `Fifo_Init`. Your ISR will call `Fifo_Put` and perform the actual output. Disarm interrupts when the FIFO is empty. Rewrite the `out` function to call `Fifo_Put`. Similar to Example 4.3, the `out` function may have to initiate the transmission and arm the interrupts if the channel is idle.

**D4.30** *Read the entire question before starting.* The objective is to interface a standard IBM PC keyboard to a microcomputer using key wake-up and output compare (Figure 4.32). The TTL level **Clock** I/O is connected to **PJ6**. The **Clock** is sometimes an input and sometimes an open-collector output. The TTL level **Data** is connected to **PJ7**. The **Data** also is sometimes an input and sometimes an open-collector output. You may write the software in assembly or C.

**Figure 4.32**  
Keyboard interface.



You will *not* use FIFOs in this problem, although your system could be significantly enhanced with FIFOs that decouple the main and background threads. Rather, there will be two global variables, creating a mail box.

```
unsigned char info; // 8 bit data from the keyboard
unsigned char flag; // true means info contains valid information
```

You will implement a simple half-duplex solution. In other words, your microcomputer software will be operating in either receive or transmit mode, but never both at the same time. You will use `info` and `flag` (no other globals are allowed). You can assume the main program leaves the system mostly in receive mode, and the operator does not type a key during the short intervals when the main program activates transmit mode. A real IBM PC keyboard interface must handle the situation where communication is attempted in both directions simultaneously (but you don't have to worry about it here).

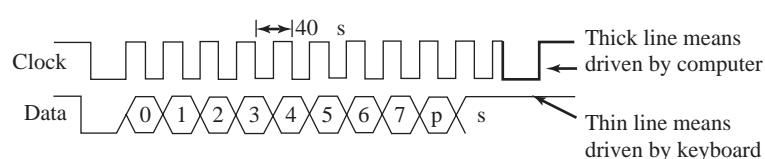
*Receive mode* is where data are being received from the keyboard into the microcomputer. When the operator types a key on the keyboard, one or more (up to six) *scan codes* (8 bits each) are sent from the keyboard to the microcomputer. You will write a background interrupt activated by `PJ6` that performs communication with the keyboard. You will also write a C function with the following prototype:

```
void ReceiveMode(void); // place the keyboard system in receive mode
```

The main program, which you do not write, is free to access the `info` and `flag` to collect and process the scan codes. In the Figure 4.33 timing diagram, the thin lines represent keyboard outputs that are open-collector with resistor pull-up, and the thick lines represent computer outputs that are also open-collector.

**Figure 4.33**

Keyboard receive timing.

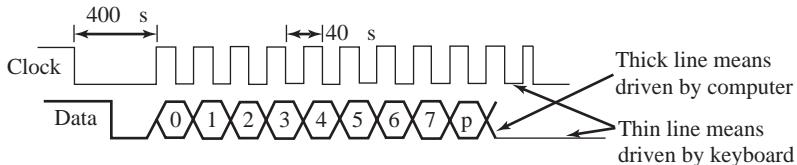


To receive a byte, your microcomputer first places both the **Clock** and **Data** in input mode. In the idle state, the keyboard makes both **Clock** and **Data** high (actually the keyboard signals are also open-collector, and the high levels are generated by resistor pull-ups on the keyboard side). When the keyboard wishes to send a scan code (remember one to six scan codes are sent after the operator types/releases a key), the keyboard will first set the **Data** low, then create a high-to-low transition on the **Clock**. This is like the start bit we saw in the asynchronous serial interface. You will accept and ignore the start bit. Then, every 40  $\mu$ s the keyboard will put the next information bit on the **Data** line and create another high-to-low transition on **Clock**. Notice the **Data** changes on the low-to-high transition, and you should input the binary bits on the high-to-low edges. It will send eight bits of information (bit 0 first), which you will read and then create an 8-bit byte from the eight individual bits. It also sends an odd parity bit (p) and stop bit (s) that you will read and ignore. It would be a good idea to do some error checking (returning with an error on a timeout, making sure the start bit is low, the odd parity is correct, and the stop bit is high), but these features are *not* required in this problem. Since the main program is performing other unrelated tasks, you will implement these receive functions in a background thread. If the **flag** is 0 when the stop bit is received (meaning the **info** is currently empty), put the new byte into **info** and set the **flag** (like a FIFO put). If the **flag** is still set when the stop bit is received (because the main program hasn't read the previous transmission), simply discard the new data (like an overrun on the SCI serial port or like a FIFO full error). As an acknowledgment back to the keyboard, your computer should make the **Clock** an open-collector output and drive **Clock** low for 40  $\mu$ s. Use OC2 (busy-wait) to create this 40- $\mu$ s timing. At the end of the PJ6 interrupt handler, your microcomputer system should leave **Clock** and **Data** in input mode, ready to accept the next scan code. There will be exactly one interrupt request for each scan code received.

*Transmit mode* is where commands are sent from the microcomputer to the keyboard. When the main program wishes to modify the keyboard functions (turn on LEDs, enable/disable certain functions) it sends a *command code* (eight bits each) to the keyboard (Figure 4.34). You will write a busy wait C function (with *no* background interrupt threads and *no* globals) with the following prototype:

```
void SendCommand(unsigned char); // transmit one command to the keyboard
```

**Figure 4.34**  
Keyboard transmit timing.



The main program, which you do not write, can call this function to transmit command codes to the keyboard. *Notice that this main program does not disable interrupts as it accesses the shared globals.* For example, the main program could first initialize the keyboard using transmit mode, then switch to receive mode to input and process scan codes:

```
void main(void){ unsigned char ScanCode;
    OtherInitialization(); // unrelated to your keyboard (don't write this)
    SendCommand(0x25); // specify keyboard operation mode
    SendCommand(0x42); // specify keyboard operation mode
    ReceiveMode(); // your function that initializes the keyboard
    while(1){
        if(flag) {
            ScanCode=info; // read next byte (like a FIFO get)
            flag=0; // means can accept another receive transmission
            Process(ScanCode); // don't write this Process
            OtherProcess(); } // unrelated to your keyboard (don't write this)
```

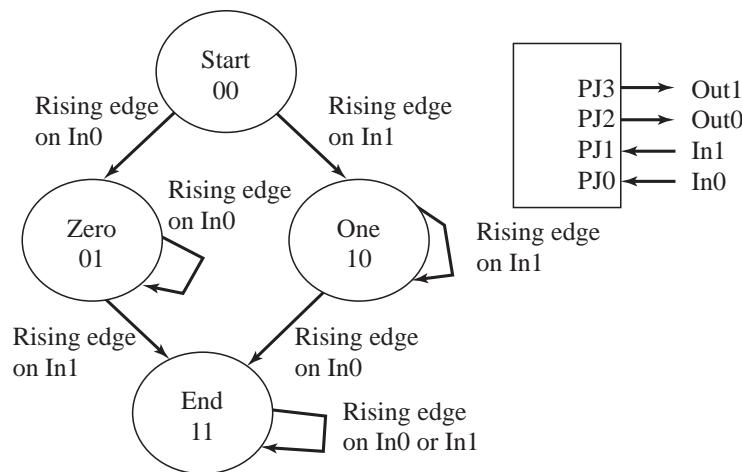
To transmit a byte, the microcomputer first disables the receive mode functions, places **Clock** and **Data** in open-collector output mode with the output high (floating), and waits for **Clock** to be high. Next the computer sets **Clock** low for 400  $\mu$ s. Again use OC2 (busy-wait) to create this timing delay. Then you pull the **Data** low (while the **Clock** is still low). Next, you release the **Clock** by making it

an input (it should float high because of the pull-up in the keyboard). The keyboard now should capture control of the **Clock** and drive it low. On the next nine low-to-high transitions you will set a new binary bit on the **Data** line. On these nine high-to-low transitions, the binary bit is latched into the keyboard. The first eight bits are the command code (bit 0 first), and the ninth bit is odd parity. After the odd parity bit is received by the keyboard, both the **Clock** and **Data** lines are held low by the keyboard until it is ready to accept another command. At the end, your microcomputer system should leave both **Clock** and **Data** in input mode.

**D4.31** You will design and implement a FSM using the *key wake-up interrupts* (Figure 4.35). One of the limitations of the previous FSM implementations is that they require 100% of the processor time and run in the foreground. In this system, there are two inputs and two outputs. We will implement a FSM where state transitions only occur on the rising edges of one of the two inputs. These rising edges should cause a key wake-up interrupt, and the FSM controller will be run in the interrupt handler. Your system will use a statically allocated linked data structure, with a one-to-one correspondence to the FSM machine. You should be able to change the data structure to accommodate other two-input, two-output FSMs of similar structure without changing the program controller. Even though you could shut off the controller once it gets to the “End” state, continue to accept key wake-up interrupts. You can assume that the inputs occur independently, and many occur at the same time.

**Figure 4.35**

Finite-state machine.

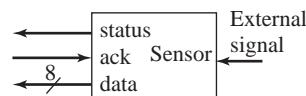


The FSM controller sequence is:

1. Wait for a rising edge on **In1** or **In0** (causing a key wake-up interrupt)
2. Go to the next state depending on the current state and which rising edge occurred
3. Output the pattern of the new state on **Out1** and **Out0**
  - a) Show the linked data structure. You may write this in C or assembly.
  - b) Show the ritual that is executed once at the beginning. There will be a main program (which you will not write) and possibly other interrupts, but this is the only device using Port J. The initial state is **Start** with its initial output of 00.
  - c) Show the key wake-up interrupt handler. Don't worry about how the interrupt vector is set.

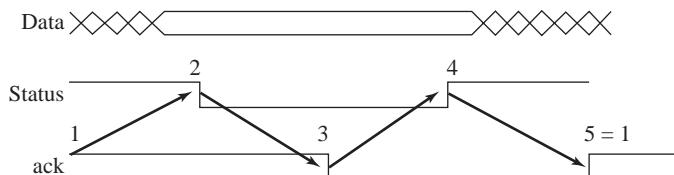
**D4.32** Design the hardware/software interface that receives data from the following fully interlocked sensor device (Figure 4.36). The sensor has **status** and **data** outputs and an **ack** input.

**Figure 4.36**



The following timing diagram (Figure 4.38) illustrates the sequence of events required to input a sensor reading into the computer. The procedure begins with the software setting its **ack** output high. Second, the sensor will perform a conversion, place the 8-bit result on its **data** lines, and set its **status** low. Third, the software will read the **data**, then set its **ack** low. Fourth, the sensor will bring its **status** high again. The fifth step (really the same as the first step for the next data transfer) is for the software to set its **ack** high, meaning it wishes to collect another sensor reading.

**Figure 4.37**



The interface timing has the following additional constraints/observations:

- Sensor **data** outputs are valid when **status** is low (software reads **data** after 2 and before 3).
- Timing events 1, 2, 3, 4 occur exactly in this order.
- Delays from 1 to 2 and from 3 to 4 are determined by the sensor, varying from 10  $\mu$ s to 1 s.
- Delays from 2 to 3 and from 4 to 1 are system response times that should be minimized.

Your software must follow these conditions:

- Both the fall and rise of **status** will cause interrupts.
- These two interrupts will have separate vectors (vectored interrupt).
- If your computer doesn't support vectored interrupt, then you may use polled interrupts.
- A FIFO queue will link your producer (background interrupt) with a consumer (main program).
- You will not write the main program that gets from the FIFO and processes the data.
- You may use any one of the FIFOs in Chapter 4 without showing its implementation.
- You may ignore COP and TOF interrupts.
- Don't worry about how the interrupt vector is set.
- You may ignore FIFO full errors (although it would have been possible to prevent them).
- a) Show the hardware interface between the sensor and your computer. You may use any available port, but make sure two interrupts can be requested that have separate vectors.
- b) Show the ritual that will be called at the beginning of the main program.
- c) Show the interrupt handler that is executed on the fall of status (timing event 2).
- d) Show the interrupt handler that is executed on the rise of status (timing event 4).

## 4.17 Lab Assignments

**Lab 4.1** The overall objective is to create an alarm clock. A periodic interrupt establishes the time of day. Input/output of the system uses the busy-wait SCI serial port developed in Chapter 3. Input is used to set the time and the alarm. Design a command interpreter that performs the necessary operations. Output is used to display the current time. An LED or sound buzzer can be used to signify the alarm.

**Lab 4.2** The overall objective is to create a software-driven pulse-width-modulated output. A periodic interrupt will set an output pin high/low. Input/output of the system uses the busy-wait SCI serial port developed in Chapter 3. Input is used to set the period and duty-cycle for the wave. Design a command interpreter that performs the necessary operations. Connect the pulse-width modulated output to an oscilloscope.

**Lab 4.3** The overall objective is to create an interrupt-driven LED light display. Interface 8 to 16 colored LEDs to individual output pins. A periodic interrupt will change the LED pattern. Connect one or two switches to the system, and use them to control which LED light pattern is being displayed. Design a linked data structure that contains the light patterns. Create device drivers for the LED outputs, switch inputs, and periodic interrupt. The main program will initialize the LED outputs, switch

inputs, and periodic interrupt. All of the input/output will be performed in the ISR. You should create a general purpose timer system that accepts a function to execute and a period. For example,

```
void main(void){  
    LED_Init();      // Initialize LED output system  
    Switch_Init();  // Initialize switch input system  
    Timer_Init(&FSMcontroller,250); // Run FSMcontroller() every 250ms  
    while(1){}  
}
```

**Lab 4.4** The overall objective is to create an **interrupt-driven traffic light controller**. The system has three digital inputs and seven digital outputs. You can simulate the system with three switches and seven LEDs. The inputs are North, East, and Walk. The outputs are six for the traffic light and one for a walk signal. Implement the controller using a finite state machine. Choose a Moore or Mealy data structure as appropriate. A periodic interrupt will run the FSM. The main program will initialize the LED outputs, switch inputs, and periodic interrupt. All of the input/output will be performed in the ISR. You should create a general-purpose timer system that accepts a function to execute and a period. For example,

```
void main(void){  
    Traffic_Init();    // Initialize switches and LED  
    Timer_Init(&Traffic_Controller,100); // Run FSM every 100ms  
    while(1){}  
}
```

# 5 Real-Time Operating Systems

## Chapter 5 objectives are to:

- ❖ Define a thread control block
- ❖ Design and implement a preemptive thread scheduler
- ❖ Design and implement spin-lock semaphores
- ❖ Design and implement blocking semaphores
- ❖ Present applications that employ semaphores
- ❖ Design, implement, and test a real-time operating system

In Chapter 4, we used interrupts to create a multithreaded environment. In that configuration we had a single foreground thread (the main program) and multiple background threads (the ISRs). These threads are run using a simple algorithm. The ISR of an input device is invoked when new input is available. The ISR of an output device is invoked when the output device is idle and needs more data. Last, the ISR of a periodic task is run at a regular rate. The main program runs in the remaining intervals. Because most embedded applications are small in size and static in nature, this configuration is usually adequate. The limitation of a single foreground thread comes as the size and complexity of the system grows. As we saw in Chapter 2, it is appropriate to partition a large project into modules, then piece together those modules in either a layered or hierarchical manner. It is during this “piece together” phase of a large project that the simple single foreground thread environment breaks down. It is easy to combine each module’s initialization routine into one common initialization sequence. The only difficulty during initialization is to prevent the initialization of one module from undoing the initialization of a previously initialized module. For example, you must be careful when two modules initialize the same I/O control register. Similarly, it is straightforward to combine the ISRs of the modules. If the modules use separate vectors, then no additional effort is required. If two modules use the same interrupt vector, then one of the polling schemes described in Chapter 4 can be used. The difficulty arises when combining the foreground functions of the modules. With a single foreground thread, we are forced into a sequential program structure similar to Figure 3.17. For the projects where the modules are tightly coupled (interdependent), this sequential model is both natural and appropriate. On the other hand, the projects where the modules are more loosely coupled (independent) may more naturally fit a multiple foreground thread configuration. The goal of this chapter is to develop the software techniques to implement multiple foreground threads (the scheduler) and provide synchronization tools (semaphores) that allow the threads to interact with each

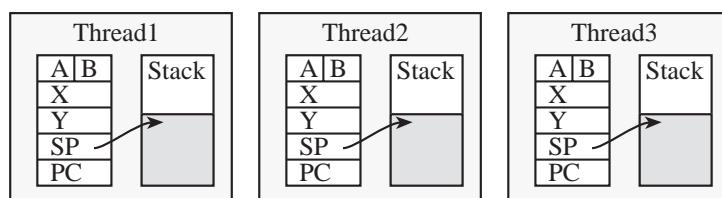
other. Systems that implement a thread scheduler still may employ regular I/O driven interrupts. In this way, the system supports multiple foreground threads and multiple background threads.

## 5.1 Introduction

We define a *thread* as the execution of a software task that has its own stack and registers (Figure 5.1). Another name for thread is lightweight process. Since each thread has a separate stack, its local variables are private, which means it alone has access. Multiple threads cooperate to perform an overall function. Since threads interact for a common goal, they do share resources, such as global memory and I/O devices (Figure 5.2).

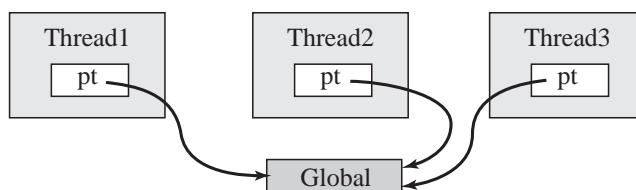
**Figure 5.1**

Each thread has its own registers and stack.



**Figure 5.2**

Threads share global memory and I/O ports.



Some simple examples of multiple threads are the interrupt-driven I/O examples of Chapter 4. In each of these examples, the background thread (ISR) executes when the I/O device is done performing the required I/O operation. The foreground thread (main program) executes during the times when no interrupts are needed. A global data structure is used to communicate between threads. Notice that the information stored on the stack or in the microcomputer registers by one thread is not accessible by another thread.

**Checkpoint 5.1:** What is the difference between a program and a thread?

**Checkpoint 5.2:** Why can't threads pass parameters to each other on the stack?

An operating system can be hard real time, soft real time, or not real time. In this book, *real time* and *hard real time* mean the same thing: a system with small and bounded interface latency. In other words, a real-time operating system (RTOS) is one that guarantees that the difference between when tasks are supposed to run and when they actually are run is short and bounded. As we saw in the last chapter, a 9S12 system can be made real time if we limit the time running with I=1 and attach important tasks to interrupt triggers. In a complex system with many tasks, it is often the case that only some of the tasks need real-time support. *Soft real time* describes a system with priority. For example, if there are two tasks ready to run, a soft real-time system will run the more important task first. The Microsoft Windows OS, Linux, and gaming routers employ priority scheduling of tasks/messages; therefore, these can be considered soft real time. In many embedded systems, real-time support is not needed. In these situations, we can have an OS that is *not real time*, meaning it does not guarantee when tasks will actually be run.

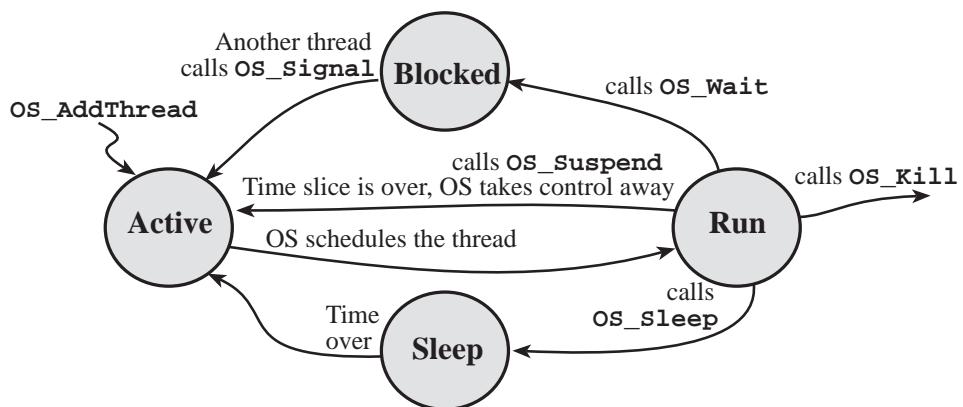
Threads themselves can be classified into three categories. A *periodic thread* is one that should run at a fixed time interval. ADC sampling, DAC outputs, and digital control are

examples of periodic tasks. Later in Section 5.4, we will present a RTOS that schedules periodic threads. An *aperiodic thread* is one that runs often, but the times when it needs to run cannot be anticipated. Threads that are attached to human input will fall into this category. A *sporadic thread* is one that runs infrequently or maybe never at all but is often of great importance. Examples of sporadic threads that have real-time requirements include power failure, CO warning, temperature overheating, and computer hardware faults.

A thread can be in one of four states, as shown in Figure 5.3. The arrows in Figure 5.3 describe the condition causing the thread to change states. In the simplest configuration, threads oscillate between the active and run states. A thread is in the *run state* if it is currently executing. On a microcontroller with a single processor like the 9S12, there can be at most one thread running at a time. As computational requirements for an embedded system rise, we can expect the future microcontroller to have multicore processors, like the ones seen now in our desktop PC. For a multicore processor, there can be multiple threads in the run state. A thread is in the *active state* if it is ready to run but waiting for its turn.

**Figure 5.3**

A thread can be in one of three states.



A thread is in the *blocked state* when it is waiting for some external event like input/output (e.g., keyboard input available, printer ready, I/O device available). We will use semaphores to implement communication and synchronization, and it is semaphore function `OS_Wait` that will block a thread if it needs to wait. For example, if a thread communicates with other threads, it can be blocked waiting for an input message or waiting for another thread to be ready to accept its output message. If a thread wishes to output to the display but another thread is currently outputting, it will block. If a thread needs information from a FIFO (`Fifo_Get`), it will be blocked if the FIFO is empty (because it can not retrieve any information). On the other hand, if a thread outputs information to a FIFO (`Fifo_Put`), it will be blocked if the FIFO is full (because it can not save its information). The semaphore function `OS_Signal` will be called when it is appropriate for the blocked thread to continue. For example, if a thread is blocked because it wanted to print and the printer was busy, it will be signaled when the printer is free. If a thread is blocked waiting on a message, it will be signaled when a message is available. Similarly, if a thread is blocked waiting on an empty FIFO, it will be signaled when new data are put into the FIFO. If a thread is blocked because it wanted to put into a FIFO and the FIFO was full, it will be signaled when another thread calls `Fifo_Get`, freeing up space in the FIFO.

**Checkpoint 5.3:** Why is it efficient to block a thread that needs a resource that is not available?

**Checkpoint 5.4:** If a thread is blocked because the output display is not available, when should you wake it up (signal it)?

Sometimes a thread needs to wait for a fixed amount of time. We will implement an `OS_Sleep` function that will make a thread dormant for a finite time. A thread in the *sleep state* will not be run. After the prescribed amount of time, the OS will make the thread active again. Sleeping would be used for tasks which are not real time. In Program 5.1, the `PeriodicStuff` is run approximately once a second.

### Program 5.1

This thread uses sleep to execute its task approximately once a second.

```
void Thread(void){
    InitializationStuff();
    while(1){
        PeriodicStuff();
        OS_Sleep(ONE_SECOND); // go to sleep for 1 second
    }
}
```

The *scheduler* is an OS function that runs the ready threads one by one. To envision a scheduler, we first list the threads that are ready to run. When the processor is free, the scheduler then will choose one thread from the ready list and cause it to run. In a *preemptive scheduler*, threads are suspended by a periodic interrupt, the scheduler chooses a new thread to run, and the return from interrupt will launch this new thread. In this situation, the OS itself decides when a running thread will be suspended, returning it to the active state. In Program 5.2, the preemptive scheduler runs `Task1`, `Task2`, and `Task3` concurrently.

### Program 5.2

Three threads run concurrently using a preemptive scheduler.

<code>void Thread1(void){     Initialization1();     while(1){         Task1();     } }</code>	<code>void Thread2(void){     Initialization2();     while(1){         Task2();     } }</code>	<code>void Thread3(void){     Initialization3();     while(1){         Task3();     } }</code>
--	--	--

In a *cooperative or non-preemptive scheduler*, the threads themselves decide when to stop running. This is typically implemented by having a thread call a function like `OS_Suspend`. This function will suspend the running thread (make it inactive), run the scheduler (choosing a new thread), and launch the new thread. Although easy to implement because it doesn't require interrupts, a cooperative scheduler is not appropriate for real-time systems. In Program 5.3, the cooperative scheduler runs `Task1`, `Task2`, and `Task3` in a cyclic manner.

### Program 5.3

Three threads run in a cooperative manner.

<code>void Thread1(void){     Initialization1();     while(1){         Task1();         OS_Suspend();     } }</code>	<code>void Thread2(void){     Initialization1();     while(1){         Task2();         OS_Suspend();     } }</code>	<code>void Thread3(void){     Initialization1();     while(1){         Task3();         OS_Suspend();     } }</code>
--	--	--

There are many scheduling algorithms one can use to choose the next thread to run. A *round robin scheduler* simply runs the ready threads in circular fashion, giving each the same amount of time to execute. A *weighted round robin scheduler* runs the ready threads in circular fashion but gives threads unequal weighting. One way to implement weighting is to vary the time each thread is allowed to run according to its importance. Another way to implement weighting is to run important threads more often. For example, assume there are three threads 1 2 3, and Thread 1 is more important. We could run the threads in this

repeating pattern: 1,2,1,3,1,2,1,3,... (i.e., every other time slice is given to Thread 1, and Threads 2 and 3 are run every fourth time slice). A *priority scheduler* assigns each thread a priority number (e.g., 1 is the highest). As we will see later, we add priority to a system that implements blocking semaphores and not to one that uses spinlock semaphores. Two or more threads can have the same priority. A priority 2 thread is run only if no priority 1 threads are ready to run. Similarly, we run a priority 3 thread only if no priority 1 or priority 2 threads are ready. If all threads have the same priority, then the scheduler reverts to a round-robin system. The advantage of priority is that we can reduce the latency (response time) for important tasks by giving those tasks a high priority. The disadvantage is that on a busy system, low priority threads may never be run. This situation is called *starvation*. A *deadline* is when a task should complete. The *time-to-deadline* is the time between now and the deadline. If you have a paper due on Friday and it is Tuesday, the time-to-deadline is 3 days. Furthermore, we define *slack time* as the time-to-deadline minus how long it will take to complete the task. If you have a paper due on Friday, it is Tuesday, and it will take you one day to write the paper, your slack time is 2 days. Once the slack time becomes negative, you will miss your deadline. There are many ways to assign priority:

- Minimize latency for real-time tasks.
- Assign a dollar cost for delayed service and minimize cost.
- Give priority to I/O bound tasks over CPU bound tasks.
- Give priority to tasks that need to run more frequently.
- Smallest time-to-deadline first.
- Smallest slack time first.

Assigning priority to tasks according to how often they run is called a *Rate Monotonic Scheduler*. Assume we have  $n$  tasks that are periodic, running with periods  $T_i$ . We assign priority according to this period, with more frequent tasks having a higher priority. Furthermore, let  $E_i$  be the maximum time to execute each task. Assuming there is little interaction between tasks, the *Rate Monotonic Theorem* can be used to predict if a scheduling solution exists. Tasks can be scheduled if

$$\sum_{i=0}^{n-1} \frac{E_i}{T_i} \leq n(2^{1/n} - 1) / \ln(2)$$

An *exponential queue* is a dynamic scheduling algorithm, with priority and varying time slices. If a thread blocks on I/O, its priority is increased and its time slice is halved. If it runs to completion of a time slice, its priority is decreased and its time slice is doubled.

One solution to starvation is called *aging*. In this scheme, threads have a permanent fixed priority and a temporary working priority. The permanent priority is assigned according the rules of the previous paragraph, but the temporary priority is used to actually schedule threads. Periodically, the OS increases the temporary priority of threads that have not been run in a long time. Once a thread is run, its temporary priority is reset back to its permanent priority.

Designing a RTOS requires many decisions to be made. Therefore, it is important to have performance criteria with which to evaluate one alternative to another. *Bandwidth* is defined as an information rate. It specifies the amount of actual data per time that are input, processed, or output. In a real-time system, *latency* is must be small and bounded. A direct measure of latency, shown in Figure 4.25, is the time between when a software task should run, and when it is actually run. An indirect measure of latency is the maximum time the system runs with interrupts disabled. *Time jitter*, introduced in Section 4.14, is an important performance measure for a RTOS. Here  $f_s$  is the frequency of the periodic task (e.g., starting the ADC), and  $t = 1/f_s$  is its period. In general, we can define time-jitter,  $\Delta t$ , as the difference between when a task is supposed to be run and when it is actually run. In this case, let  $t_n$  be the time the software task is actually run and let  $n$  be the time it was supposed to be run. Then the time-jitter at sample  $n$  is

$$\Delta t_n = t_n - n \cdot t$$

For a real-time system with periodic tasks, we must be able to place an upper bound,  $k$ , on the time-jitter.

$$-k \leq t_n - t \leq +k \text{ for all } n$$

Sometimes it is more important to control the time difference between periodic events rather than the absolute time itself. Let  $t_n$  be the actual time difference between two executions of a periodic task. The desired time difference is  $t$ . For this situation, we define the time-jitter at sample  $n$  to be

$$t_n = t_n - t$$

Again, we must be able to place an upper bound,  $k$ , on the time-jitter. In most situations, the time jitter will be dominated by the time the 9S12 runs with interrupts disabled.

$$-k \leq t_n - t \leq +k \text{ for all } n$$

*Utilization* is defined as the percentage of available resources that are actually used. For example, the 9S12 running at 24 MHz has 24 million bus cycles in a second. If 18 million cycles each second are used performing actual useful work, we can specify the RTOS has a CPU utilization of 75%. If a serial channel can support 960 bytes/sec of data bandwidth and the system actually transmits 96 bytes/sec, then the utilization is 10%. *Breakdown utilization* (BU) is the percentage of resource utilization below which the RTOS can guarantee that all deadlines will be met. For example, a RTOS that can meet all of its deadlines running at 67% CPU utilization is better than a RTOS that needs to run at CPU utilization of 50% in order to meet deadlines. Furthermore, assuming both systems are performing the same useful work, the first RTOS can run at a slower E clock than the second, saving power and money. To see breakdown utilization in another way, consider a package shipping company. Useful work is defined as package volume in  $m^3$  shipped over distance in km per hour. Assume the goal is to ship 1000  $m^3$ -km/hr, and assume also one truck can deliver a maximum of 100  $m^3$ -km/hr. In a perfect situation, 10 trucks could perform the task. In addition to actually shipping packages, the trucking company must maintain its trucks, load and unload packages, and wait around because of scheduling conflicts. All of these support tasks are analogous to the OS tasks that need to run that do not directly perform useful work for the customer. Because of the support tasks, the first trucking company needs 15 or more trucks to guarantee meeting the goal of 1000  $m^3$ -km/hr delivered on time. Their breakdown utilization is 67% (10/15). This means, if they try to deliver 10 trucks worth of work with only 14 trucks (utilization = 10/14 = 71%), they will miss some deadlines. A second trucking company needs 20 or more trucks to guarantee all packages are delivered on time. Their breakdown utilization is 50% (10/20). The first company is clearly more efficient and effective than the second. *Normalized mean response time* (NMRT) is the ratio of the “best case” time interval a task becomes ready to execute and then terminates, and the actual CPU time consumed. *Guaranteed ratio* (GR), used in dynamic scheduling, is the number of tasks whose deadlines can be guaranteed to be met versus the total number of tasks requesting execution.

When optimizing for performance, it is important to understand the user application. A *static* system is one where the needs of the task are fixed and known at design time. This type of system is easiest to optimize, because performance during the design and test phases will match results occurring when the system is deployed. A *deterministic* system has variable requirements, but the requirements are either known or can be predicted in advance. The performance of a deterministic system can be studied using probabilistic theory. A *dynamic* system also has variable requirements, but there may be no way to know or predict these needs in advance. The RTOS for a dynamic system must embed debugging instruments into the delivered system that measure system performance during deployment. In this way, the OS designer and the application engineer (customer) can work together to tune the OS so that it meets or exceeds the performance criteria.

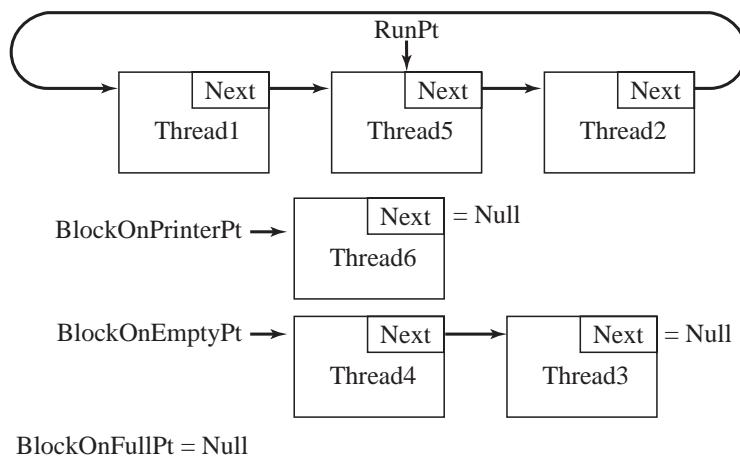
A *hook* is an OS feature that allows the user to attach functions to strategic places in the OS. Examples of places we might want to place hooks include: whenever the OS has finished initialization, the OS is running the scheduler, or whenever a new thread is created. To use a hook, the user writes a function, calls the OS, and passes a function pointer. These programs are extra for the user's convenience and not required by the OS itself. When that event occurs, the OS calls the user function. Hooks are extremely useful for debugging.

Whenever the object code for the OS can be downloaded into ROM, such that the system will launch and run on power up, we classify the OS as *ROMable*. A ROMable OS is essential for embedded systems. The operating systems that run on a personal computer (e.g., Windows, Linux, Tiger) are not ROMable, meaning we need to download the OS into memory each time the system is powered up. Another property important for embedded systems is *certification*. Whenever an OS is deployed in embedded system used in safety-critical applications (e.g., military, nuclear, transportation, or medical) the corresponding governing body must certify the OS satisfies the appropriate specifications. The minimally intrusive debugging instruments such as profiles, dumps, and monitors are necessary to collect data to support the claims that the OS performs according to its specification.

A good implementation for the scheduler uses linked-list data structures to hold the ready and blocked threads. We can create a separate blocked linked list for each reason why the thread can not execute. For example, one blocked list for waiting for the output display to be free, one for full during a call to `Fifo_Put`, and, one for empty during a call to `Fifo_Get`. In general, we will have one blocked list with each blocking semaphore. In Figure 5.4, Thread 5 is running, Threads 1 and 2 are ready to run, Thread 6 is blocked waiting for the printer, and Threads 3 and 4 are blocked because the FIFO is empty.

**Figure 5.4**

Thread 5 is running, threads 1 and 2 are ready to run, and the rest are blocked.

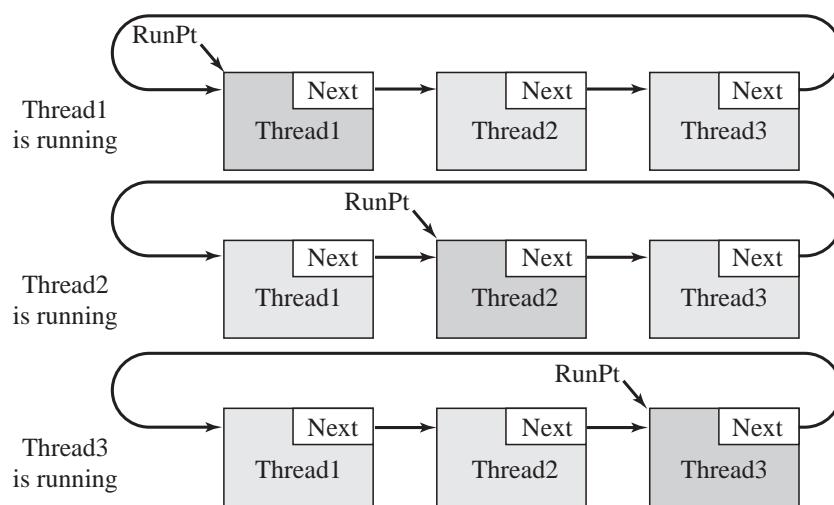


## 5.2 Round-Robin Scheduler

In this section, we will develop a simple multithreaded round-robin scheduler. In particular, there will be three statically allocated threads that are each allowed to execute 1 ms in a round-robin fashion. Even though there are two programs, ProgA and ProgB, there will be three threads (Figure 5.5). Recall that a thread is not simply the software but the execution of the software. In this way, we will have two threads executing the same program, ProgA.

**Figure 5.5**

The circular linked list allows the scheduler to run all three threads equally.



The thread control block (TCB) will store the information private to each thread. There will be a 64-byte TCB structure for each thread. The TCB must contain:

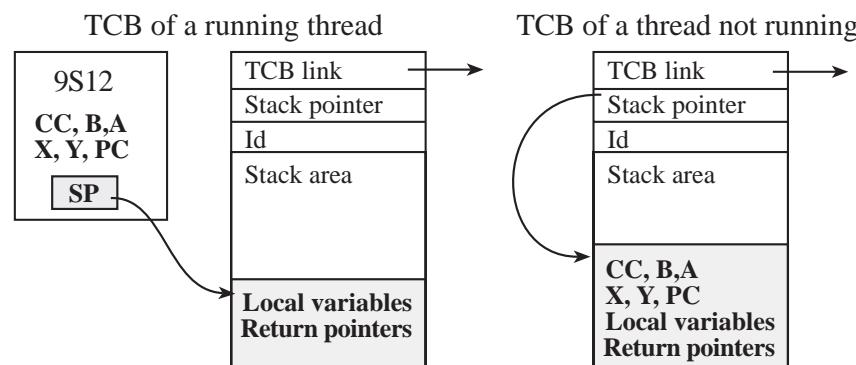
1. A pointer so that it can be chained into a linked list
2. The value of its stack pointer
3. A stack area that includes local variables and other registers

While a thread is running, it uses the actual 9S12 hardware registers CCR, B, A, X, Y, PC, and SP (Figure 5.6). In addition to these necessary components, the TCB might also contain:

4. Thread number, type, or name
5. Age, or how long this thread has been active
6. Priority
7. Resources that this thread has been granted or blocked
8. A sleep counter

**Figure 5.6**

The running thread uses the actual registers, while the other threads have their register values saved on the stack.



To illustrate the concept of a preemptive scheduler, we will implement a round-robin scheduler first in 9S12 assembly then in 9S12 C. The three statically allocated threads are arranged in a circular linked list (Program 5.4). This example illustrates the difference between a program (e.g., ProgA and ProgB) and a thread (e.g., Thread1, Thread2, and Thread3). Notice that Threads 1 and 2 both execute ProgA. There are many applications where the same program is being executed multiple times.

**Program 5.4**

Assembly code that statically allocates three threads.

org \$0800 ;RAM on the 9S12DP512 RunPt rmb 2 ;pointer to thread that is currently running ;TCBs go in RAM, each TCB is 64 bytes		
TCB1 rmb 2 ;next	TCB2 rmb 2 ;next	TCB3 rmb 2 ;next
SP1 rmb 2 ;stackPt	SP2 rmb 2 ;stackPt	SP3 rmb 2 ;stackPt
Id1 rmb 1 ;1	Id2 rmb 1 ;2	Id3 rmb 1 ;Id=4
Stck1 rmb 50	Stck2 rmb 50	Stck3 rmb 50
IS1 rmb 7 ;CCR,D,X,Y	IS2 rmb 7	IS3 rmb 7
IPC1 rmb 2 ;PC	IPC2 rmb 2	IPC3 rmb 2

Even though the thread has not yet been allowed to run, it is created with an initial stack area that “looks like” it had been previously suspended by an OC5 interrupt. Notice that the initial value loaded into the CCR (\$40) when the thread runs for the first time has XIRQ disabled (X=1) and IRQ enabled (I=0). When the thread is launched for the first time, it will execute the program specified by the value in the “initial PC” location. The Id is used to visualize the active thread.

The round-robin preemptive scheduler simply switches to a new thread every 1 ms (Programs 5.5, 5.6).

**Program 5.5**

Assembly code that implements the preemptive thread switcher.

```

        org $4000 ;programs in ROM
Next  equ 0      ;pointer to next TCB
StkPt equ 2     ;Stack pointer for this thread
Id    equ 4     ;Used to visualize which thread is current running
Main  lddaa #$FF
        staa DDTT   ;PTT is which program is executing
        staa DDPt   ;PTP is which thread
        movb #$80,TSCR1 ;enable TCNT, 8 MHz
        bset TIOS,$$20 ;output compare 5
        bset TIE,$$20 ;arm OC5
        movw #TCB2,TCB1 ;create circular list
        movw #TCB3,TCB2 ;next
        movw #TCB1,TCB3 ;round robin
        movw #IS1,SP1   ;initial SP
        movw #IS2,SP2
        movw #IS3,SP3
        movb #1,Id1     ;thread Id
        movb #2,Id2
        movb #4,Id3
        movb #$40,IS1   ;initial CCR
        movb #$40,IS2
        movb #$40,IS3
        ldd #ProgA
        std IPC1       ;initial PC
        std IPC2       ;threads 1 and 2 run ProgA
        ldd #ProgB
        std IPC3       ;thread 3 runs ProgB
        ldx #TCB1      ;First thread to run
        jmp Start
; Suspend thread which is currently running
C5Han ldx RunPt
        sts StkPt,x ;save Stack Pointer in TCB
* launch next thread
        ldx Next,x

```

*continued on p. 260*

**Program 5.5**

Assembly code that implements the preemptive thread switcher.

*continued from p. 259*

```

Start stx RunPt
    movb Id,x,PTP ;visualizes running thread
    lds StkPt,x ;set SP for this new thread
    ldd TCNT
    addd #8000 ;interrupts every 1 ms
    std TC5
    movb #$20,TFLG1 ;acknowledge OC5
    rti

```

i set 0 ProgA leas -2,sp ldd #5 std i,sp ;i=5 loopA movb #2,PTT ;PT1 ldd i,sp jsr sub std i,sp bra loopA	i set 0 ProgB leas -2,sp ldd #6 std i,sp ;i=6 loopB movb #4,PTT ;PT2 ldd i,sp jsr sub std 0,sp bra loopB	;in: j ;out j+1 j set 0 sub std 2,-sp movb #1,PTT ;PT0 ldd j,sp addd #1 leas 2,sp rts
--	--	---

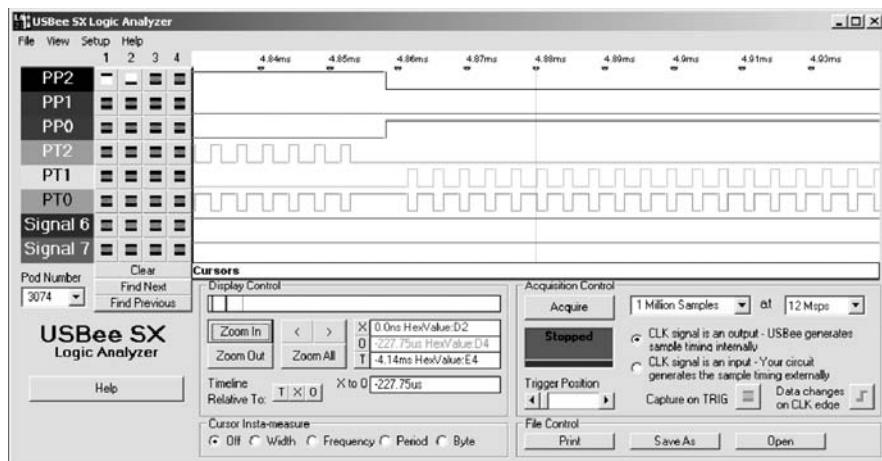
**Program 5.6**

Assembly code for the two main programs and shared subroutine.

Figure 5.7 shows a profile of Programs 5.4 through 5.6 measured on a logic analyzer. This recording was collected on a 9S12DP512 running at 8 MHz. During time before 4.855 ms on this plot, PP2 is high, which means Thread 3 is running. During this time, the oscillations between PT2 (running ProgB) and PT0 (running sub) show the execution pattern for Thread 3. Around time 4.855 ms, the output-compare interrupt runs the scheduler. The fall of PP2 and rise of PP0 mean Thread 3 is suspended and Thread 1 is about to run. For times after 4.855 ms, oscillations between PT1 (running ProgA) and PT0 (running sub) show the execution pattern for Thread 1. Looking at the PT0 trace, we can estimate the thread switch time to be about 8  $\mu$ s, because the repeating pattern on PT0 shows an 8  $\mu$ s gap. The thread switch ISR in Program 5.5 requires 38 cycles to execute. Therefore,

**Figure 5.7**

Profile of the three threads running two main programs.



we expect nine cycles for the interrupt context switch, 19 cycles for the serial monitor, and 38 cycles for the scheduler. These 66 cycles consume 8.25  $\mu$ s on an 8 MHz 9S12, and this estimation is consistent with the measured data in Figure 5.7.

**Checkpoint 5.5:** Do the data in Figure 5.7 show that the subroutine **sub** has been re-entered?

Next, we will implement this same multithreaded RTOS in C. Because the TCBs contain the stack and the saved stack pointer, they must be allocated in RAM. Analogous to Program 5.4, Program 5.7 shows the TCB definitions for this operating system. One tricky part about defining the TCB is to choose the size of `MoreStack` so that all entries are contiguous. Most compilers will try to allocate 16-bit variables (e.g., `InitialRegX`) on an even address. Because the TCB starts at an even address, `MoreStack`, `InitialCCR`, and `InitialRegA` will be at odd addresses while the rest of the TCB entries will be allocated at even addresses. If you change the 50 to 49, then the compiler may leave a 1-byte gap between `InitialRegA` and `InitialRegX`, then the program will not work. Notice that the `Next` fields in the TCB form a circular linked list, as shown in Figure 5.5. One consequence of this approach is there will be two copies of the 192 bytes used for the three TCBs. There will be one copy in ROM having the initial values, and there will be a second copy in RAM that will be used at run time. During initialization, before the `main` program is invoked, the compiler includes software that transfers the ROM copy into the RAM locations. After the

### Program 5.7

C code for the thread control block.

```

struct TCB{
    struct TCB *Next;           // Link to Next TCB
    unsigned char *StackPt;     // Stack Pointer
    unsigned char Id;          // 1, 2, 4,
    unsigned char MoreStack[50]; // additional stack
    unsigned char InitialCCR;
    unsigned char InitialRegB;
    unsigned char InitialRegA;
    unsigned short InitialRegX;
    unsigned short InitialRegY;
    void (*InitialPC)(void); }

typedef struct TCB TCBType;
typedef TCBType * TCBPtr;
TCBType TCBs[3]={
    { &TCBs[1],           // Pointer to Next
        &TCBs[0].InitialCCR, // Initial SP
        1,                  // Thread1, Id=1
        { 0 },
        0x40,0,0,0,0,        // Initial CCR,B,A,X,Y
        ProgA },            // Thread 1 runs ProgA
    { &TCBs[2],
        &TCBs[1].InitialCCR,
        2,                  // Thread2, ID=2
        { 0 },
        0x40,0,0,0,0,
        ProgA },            // Thread2 runs ProgA
    { &TCBs[0],
        &TCBs[2].InitialCCR,
        4,                  // Thread3, ID=4
        { 0 },
        0x40,0,0,0,0,
        ProgB } };          // Thread3 runs ProgB
TCBPtr RunPt; // Pointer to thread currently running

```

thread is launched for the first time, the initial parameters, defined as `InitialCCR` to `InitialPC` in the TCB structure, no longer contain initial parameters, and the entire 59-byte area is used for the stack.

**Checkpoint 5.6:** What does `RunPt` point to?

Analogous to Program 5.5, Program 5.8 shows the ISR implementing the thread switch. The output compare 5 handler implements the round-robin preemptive scheduler, as described in Figure 5.5. This C ISR requires only 44 cycles to execute, so there is just a small speed disadvantage for using C. Most compilers do not like programmers who read and write the stack pointer, so expect compiler warnings when implementing the low-level thread switching. The initialization configures output-compare 5 interrupts, and initializes the `RunPt`. Recall, the initial TCBs were created to mimic how it would look if it had been running and then was suspended. To launch the first thread, the SP is set into the initial stack pointing to the \$40, which is the initial CCR, and a `rti` is executed.

### Program 5.8

C code for the thread switcher.

```
interrupt 13 void threadSwitchISR(void){
    asm ldx RunPt
    asm sts 2,x          // save SP for suspended thread
    RunPt = RunPt->Next; // Round robin
    PTP = RunPt->Id;    // PTP shows which thread is running
    TC5 = TCNT+8000;    // Time slice = 1ms
    TFLG1 = 0x20;        // acknowledge by clearing C5F
    asm ldx RunPt
    asm lds 2,x          // restore SP for next thread to run
}
void main(void) {
    DDRP = 0xFF; // for debugging
    DDRT = 0xFF;
    RunPt = &TCBs[0]; // Specify first thread to run
    TIOS |= 0x20; // activate OC5
    TSCR1 = 0x80; // enable TCNT, 8 MHz
    TIE |= 0x20; // Arm TC5
    TC5 = TCNT+8000;
    TFLG1 = 0x20; // Clear C5F
    PTP = RunPt->Id;
    asm ldx RunPt
    asm lds 2,x
    asm rti // Launch First Thread
}
```

**Checkpoint 5.7:** What is the purpose of the `sts lds` instructions?

**Checkpoint 5.8:** What happens if a thread executes `TC5=TCNT+6;`

Program 5.9 is the C version of Program 5.6, which shows the user software consisting of two programs and a subprogram. A logic analyzer connected to PTT will show which program is running, and PTP will show which thread is running. Except for the extra six cycles it takes the C version to run the output compare ISR, the logic analyzer output for the C version is identical to data collected in Figure 5.7 using the assembly version of the RTOS. When debugging the RTOS, we can open a debugging window and observe the TCB structure, as shown in Figure 5.8. In this system, `TCBs[3]` exists in memory locations 0x0900 to 0x9BF. The first TCB entries form a circular linked list. The second TCB entries are the stack pointers when the thread is in the active state (not running). The third TCB entries are the thread `Id` (1, 2, 4). The three shaded portions in Figure 5.8 are the three stacks. We can visually see that none of the three threads is experiencing stack overflow,

because the top portion of each stack area remains at the initial value of 0. We should expect some stack space will be required to run the serial monitor debugger, which may explain why Thread 2 uses more stack space than Thread 1—even though they are running the same program. After running this system for an hour, the maximum stack size was 20 bytes. It would be reasonable therefore to leave the stack allocation at 59 bytes, which is two to three times the maximum observable size. In a mission-critical application, we could add runtime error checking to see if the stack size gets within 10 bytes of the maximum or generate theoretical proof that the stack cannot overflow. A stack underflow represents a software crash, and this type of error should be removable during testing.

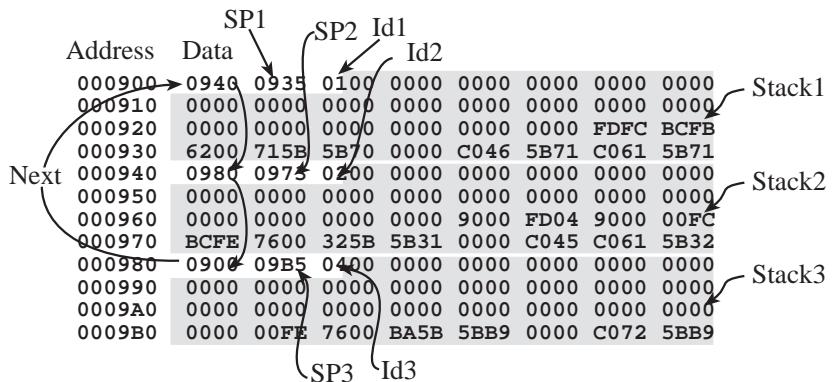
### Program 5.9

C code for the two main programs and shared subroutine.

<code>void ProgA(){ short i; i = 5; while(1){     PTT = 2;     i = sub(i); }</code>	<code>void ProgB(){ short i; i = 6; while(1){     PTT = 4;     i = sub(i); }</code>	<code>// sub-function // input: j // output: j+1 short sub(short j){     PTT = 1;     return(j+1); }</code>
---	---	---

**Figure 5.8**

Metrowerks Codewarrior memory debugging window during execution of Programs 5.7 through 5.9.



We can easily add a cooperative feature to our RTOS. By triggering an output compare interrupt, we cause the current thread to be suspended and a new one run. Program 5.10 can be used to run the cooperative multitasking problem described in Program 5.3. The “6” is selected so the output compare 5 trigger occurs during the `rts` instruction of the `OS_Suspend` function.

### Program 5.10

C function to suspend a new thread.

```
void OS_Suspend(void){  
    TC5 = TCNT+6;  
}
```

In the previous two examples, the threads were specified at assembly/compile time. We can use the compiler’s memory manager to dynamically allocate the deallocate threads. Program 5.11 dynamically allocates space for a new TCB and initializes its contents.

With this new function, the user can create threads at run time. Now, we can remove the `TCBs[3]` definition and replace the line `RunPt=&TCBs[0];` in the main program of Program 5.8 with the following three calls:

```
OS_AddThread(&ProgA,1); // Thread1 is ProgA  
OS_AddThread(&ProgA,2); // Thread2 is ProgA  
OS_AddThread(&ProgB,4); // Thread3 is ProgB
```

**Program 5.11**

C function to create a new thread.

```
void OS_AddThread(void (*program)(void), unsigned char TheId){
    TCBPtr NewPt;           // pointer to new thread control block
    NewPt = (TCBPtr)malloc(sizeof(TCBType)); // space for new TCB
    if(NewPt==0)return;
    NewPt->StackPt = &(NewPt->InitialCCR); // SP when not running
    NewPt->Id = TheId;                // used to visualize active thread
    NewPt->InitialCCR = 0x40;          // Initial CCR, I=0
    NewPt->InitialRegB = 0;            // Initial RegB
    NewPt->InitialRegA = 0;            // Initial RegA
    NewPt->InitialRegX = 0;            // Initial RegX
    NewPt->InitialRegY = 0;            // Initial RegY
    NewPt->InitialPC = program;       // Initial PC
    if(RunPt){
        NewPt->Next = RunPt->Next;
        RunPt->Next = NewPt;           // Link it into circular list
    }else{
        RunPt = NewPt;                // the first and only thread
        NewPt->Next = NewPt;          // circular list of one TCB
    }
}
```

To implement the sleep function, we add a counter to each TCB and call it `Sleep`. If `Sleep` is zero, the thread is not sleeping and can be run, meaning it is either in the run or active state. If `Sleep` is nonzero, the thread is sleeping. We change the scheduler replacing the line `RunPt=RunPt->Next;` in Program 5.8 with

```
RunPt = RunPt->Next; // skip at least one
while(RunPt->Sleep){ // do not run if sleeping
    RunPt = RunPt->Next; // find one not sleeping
}
```

In this way, any thread with a nonzero `Sleep` counter will not be run. The user must be careful not to let all the threads go to sleep, because doing so would crash this implementation. Next, we need to add a periodic task that decrements the `Sleep` counter for any nonzero counter. When a thread wishes to sleep, it will set its `Sleep` counter and invoke the cooperative scheduler, as shown in Program 5.12. The period of this decrementing task will determine the resolution of the parameter `time`.

**Program 5.12**

C function to put a thread to sleep.

```
void OS_Sleep(unsigned short time){
    RunPt->Sleep = time; // how long to not run
    OS_Suspend();         // stop running
}
```

When designing a thread scheduler, it is important to observe the assembly code generated by the compiler. It is important to avoid C code that necessitates creating local variables on the stack, because the thread switcher will allocate the local variables on the stack of the old thread and deallocate them off the stack of the new thread.

## 5.3 Semaphores

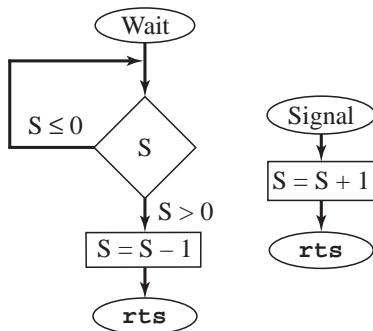
We will use semaphores to implement synchronization, sharing, and communication between threads. A semaphore is a counter and has two atomic functions (methods) apart from initialization that operate on the counter. The operations are called P or wait (derived from the Dutch *proberen*, “to test”), and V or signal (derived from the Dutch *verhogen*,

“to increment”). When we use a semaphore, we usually can assign a meaning or significance to the counter value. A binary semaphore has a global variable that can be 1 for free and 0 for busy. Another name for wait is pend, and another name for signal is post.

### 5.3.1 Spin-Lock Semaphore Implementation

There are many implementations of semaphores, but the simplest is called *spin-lock* (Figure 5.9). If the thread calls *wait* with the counter less than or equal to 0 it will “spin” (do nothing) until the counter goes above zero (Program 5.13). In the context of the previous round-robin scheduler, a thread that is “spinning” will perform no useful work but eventually will be suspended by the OC5 handler, and other threads will execute. It is important to allow interrupts to occur while the thread is spinning so that the computer does

**Figure 5.9**  
Flowcharts of a spin-lock counting semaphore.



<pre> ;assembly implementation S      fcb 1      ;semaphore, initialized to 1  ; spin if zero, otherwise decrement wait sei          ;read-modify-write atomic       ldaa S      ;value of semaphore       tsta       bgt OK        ;available if &gt;0       cli       bra  wait   ;**interrupts occur here** OK   deca       staa S      ;S=S-1       cli       rts  ;increment semaphore signal       inc  S      ;S=S+1, this is atomic       rts   </pre>	<pre> // ***** OS_Wait ***** // decrement and spin if less than 0 // input: pointer to a semaphore // output: none void OS_Wait(short *semaPt){     asm sei          // Test and set is atomic     while(*semaPt &lt;= 0){ // disabled         asm cli          // disabled         asm nop          // enabled         asm sei          // enabled     }     (*semaPt)--;           // disabled     asm cli          // disabled } // ***** OS_Signal ***** // increment semaphore // input: pointer to a semaphore // output: none void OS_Signal(short *semaPt){     unsigned char SaveCCR;     asm tpa     asm staa SaveCCR // save previous     asm sei          // make atomic     (*semaPt)++;     asm ldaa SaveCCR // recall previous     asm tap          // end critical }   </pre>
--	---

#### Program 5.13

A spin-lock counting semaphore.

not hang up. The read-modify-write operation on the global semaphore must be made atomic, because the scheduler might switch threads in between any two instructions that execute with the interrupts enabled.

**Checkpoint 5.9:** What happens if a spin-lock **wait** loops with interrupts disabled?

**Checkpoint 5.10:** What happens if the **sei cli** instructions are removed entirely from the spin-lock wait?

The 9S12 **minm** instruction can be used to implement a binary semaphore without disabling interrupts (Program 5.14). In C, these functions are shown in Program 5.15.

#### Program 5.14

9S12 assembly code for a spin-lock binary semaphore.

```
S      fcb 1      ;semaphore flag initialized to be 1
bWait clra      ;new value
ldx #S ;pointer to S
loop   minm 0,x  ;in either case S is now 0, carry set if S used to be 1
       bcc loop  ;loop until S=1
       rts
bSignal movb #1,S ;S=1
       rts
```

#### Program 5.15

9S12 C code for a spin-lock binary semaphore.

```
void bWait(char *semaphore){
asm tfr D,X // RegX points to semaphore
asm clra // new value for semaphore
asm minm 0,x // test and set
asm bcc *-3
}
void bSignal(char *semaphore){
(*semaphore) = 1; // compiler makes this atomic
}
```

**Observation:** If the semaphores can be implemented without disabling interrupts, then the latency in response to external events will be improved.

We can use three binary semaphores and a counter to implement a counting semaphore again without disabling interrupts (Program 5.16).

#### Program 5.16

C code for a counting semaphore.

```
struct sema4      // counting semaphore based on 3 binary semaphores
{
    short value; // semaphore value
    char s1;     // binary semaphore
    char s2;     // binary semaphore
    char s3;     // binary semaphore
};
typedef struct sema4 sema4Type;
typedef sema4Type * sema4Ptr;
void Wait(sema4Ptr semaphore){
    bWait(&semaphore->s3); // wait if other caller to Wait gets here first
    bWait(&semaphore->s1); // mutual exclusive access to value
    (semaphore->value)--; // basic function of Wait
    if((semaphore->value)<0){
```

```

        bSignal(&semaphore->s1); // end of mutual exclusive access to value
        bWait(&semaphore->s2);   // wait for value to go above 0
    }
}
else
    bSignal(&semaphore->s1); // end of mutual exclusive access to value
    bSignal(&semaphore->s3);      // let other callers to Wait get in
}
void Signal(sema4Ptr semaphore){
    bWait(&semaphore->s1); // mutual exclusive access to value
    (semaphore->value)++; // basic function of Signal
    if((semaphore->value)<=0)
        bSignal(&semaphore->s2); // allow S2 spinner to continue
    bSignal(&semaphore->s1); // end of mutual exclusive access to value
}
void Initialize(sema4Ptr semaphore, short initial){
    semaphore->s1=1; // first one to bWait(s1) continues
    semaphore->s2=0; // first one to bWait(s2) spins
    semaphore->s3=1; // first one to bWait(s3) continues
    semaphore->value=initial;
}

```

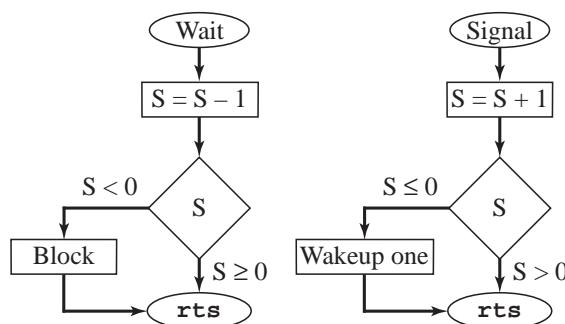
### 5.3.2 Blocking Semaphore Implementation

There are two problems associated with spin-lock semaphores. The first is an obvious inefficiency in having threads spin while there is nothing for them to do. Blocking semaphores will be a means to recapture this lost processing time. Essentially, with blocking semaphores, a thread will not run unless it has useful work it can accomplish. The second problem with spin-lock semaphores is a fairness issue. Consider the case with Threads 1, 2, and 3 running in round-robin order. Assume Thread 1 is the one calling *Signal*, and threads 2 and 3 call *Wait*. If Threads 2 and 3 are both spinning waiting on the semaphore and then Thread 1 signals the semaphore, which thread (2 or 3) will be allowed to run? Because of its position in the 1, 2, and 3 cycle, Thread 2 will always capture the semaphore ahead of Thread 3. It seems fair when the status of a resource goes from busy to available that all threads waiting on the resource get equal chance. Many businesses have solved this fairness problem by issuing numbered tickets, creating queues, or having the customers sign a log when they enter the business looking for service (e.g., when waiting for a checkout clerk at the grocery store, we know to get in line, and we think it is unfair for pushy people to cut in line). We define *bounded waiting* as the condition where once a thread begins to wait on a resource (the call to *Wait* does not return right away), there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed. Bounded waiting does not guarantee a minimum time before *wait* will return; it just guarantees a finite number of other threads will go before this thread. For example, it is Christmas time, I want to mail a package to my mom, I walk into the post office and take a number, the number on the ticket is 251, I look up at the counter and the display shows 212, and I know there are 39 people ahead of me in line. We will implement bounded waiting with blocking semaphores by placing the blocked threads on a list, which are sorted by the order in which they blocked. When we wake up a thread off the blocked list, we wake up the one that has been waiting the longest.

The priority schedulers described in Section 5.1 require use the blocking semaphores. That is, we cannot use a priority scheduler with spin-lock semaphores. One problem with a priority scheduler is *priority inversion*, a condition where a high-priority thread is waiting on a resource owned by a low-priority thread. For example, consider the case where both a high-priority and low-priority thread need the same resource. Assume the low-priority thread asks for and is granted the resource, and then the high-priority thread asks for it and blocks. During the time the low-priority thread is using the resource, the high-priority thread essentially becomes low priority. One solution to priority inversion is *priority inheritance*. With priority inheritance, once a high-priority thread blocks on a resource, the thread holding the resource is granted temporarily a high priority.

For this implementation, each semaphore has an integer global variable (an up/down counter) and a blocked **tcb** linked list. This linked list contains the threads that are blocked (not ready to run) because they called the `Wait` function when the semaphore counter was less than or equal to 0. Our semaphore “object” also has three “methods”: `Initialize`, `Wait`, and `Signal` (Figure 5.10). The proper way to make a multistep sequence atomic is to first save the interrupt status (I bit) on the stack, then disable interrupts. At the end of the atomic sequence, we restore the I bit back to its original value rather than simply assuming it was enabled to begin with. This procedure to disable/enable interrupts handles the situation of nested critical sections. We must be careful to block and wake up the correct number of threads.

**Figure 5.10**  
Flowcharts of a blocking counting semaphore.



#### Initialize (Program 5.17):

1. Set the counter to its initial value
2. Clear the associated blocked **tcb** linked list

#### Wait:

1. Disable interrupts to make atomic:

```

tpa
psha    save old interrupt status
sei

```

2. Decrement the semaphore counter,  $S=S-1$
3. If the semaphore counter is less than 0, then

Block this thread onto the associated **tcb** blocked linked list by executing SWI

The SWI operation performs stack operations similar to an OC5 interrupt

The SWI handler suspends the current thread

Moves the **tcb** of this thread from the active list to the blocked list

Launches another thread from the active list

4. Restore interrupt status:

```

pula
tap

```

#### Signal:

1. Disable interrupts to make atomic:

```

tpa
psha    save old interrupt status
sei

```

2. Increment the semaphore counter,  $S=S+1$

3. If the semaphore counter is less than or equal to 0, then
  - Wake up one thread from the **tcb** linked list
  - Do not suspend execution of this thread
  - Simply move the **tcb** of the “wake-up” thread from the blocked list to the active list
4. Restore interrupt status:

```
pula
tap
```

### Program 5.17

Assembly code to initialize a blocking semaphore.

```
S      rmb  1    ;semaphore counter
BlockPt rmb  2    ;Pointer to threads blocked on S
Init   tpa
       psha           ;Save old value of I
       sei            ;Make atomic
       ldaa #1
       staa S          ;Init semaphore value
       ldx  #Null
       stx  BlockPt    ;empty list
       pula
       tap            ;Restore old value of I
       rts
```

The implementation in Program 5.18 assumes the system only has one semaphore. In a more general implementation, there would be multiple semaphores, each with its own counter and blocked list. In this case, a pointer (call by reference) to the semaphore structure (counter and blocked list) would be passed to the SWI handler.

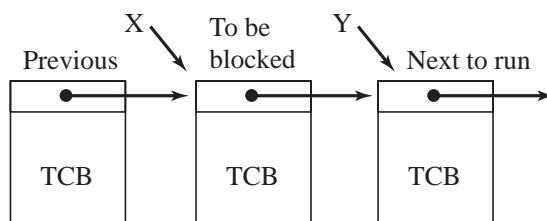
### Program 5.18

Assembly code helper function to block a thread, used to implement a blocking semaphore.

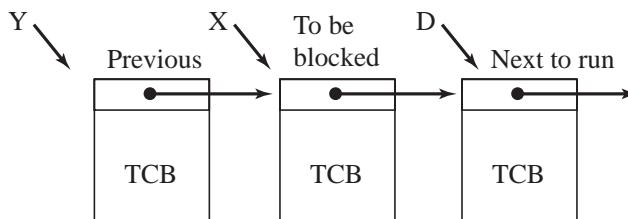
```
;To block a thread, execute SWI
;RunPt points to the active thread, which will be blocked
;BlockPt points to the blocked list for this semaphore
SWIhan ldx RunPt    ;running process "to be blocked"
        sts SP,x    ;save Stack Pointer in its TCB
; Unlink "to be blocked" thread from RunPt list
        ldy Next,x  ;find previous thread (see Figure 5.11)
        sty RunPt    ;next one to run
look   cpx Next,y  ;search to find previous
        beq found
        ldy Next,y
        bra look
found ldd RunPt    ;one after blocked (see Figure 5.12)
        std Next,y  ;link previous to next to run
; Put "to be blocked" thread on block list
        ldy BlockPt ;(see Figure 5.13)
        sty Next,x  ;link "to be blocked"
        stx BlockPt
; Launch next thread
        ldx RunPt
        lds SP,x    ;set SP for this new thread
        ldd TCNT    ;Next thread gets a full 1ms time slice
        addd #8000  ;interrupt after 1 ms
        std TC5
        ldaa #$20    ;($20 on the 6812)
        staa TFLG1   ;clear OC5F
        rti
```

**Figure 5.11**

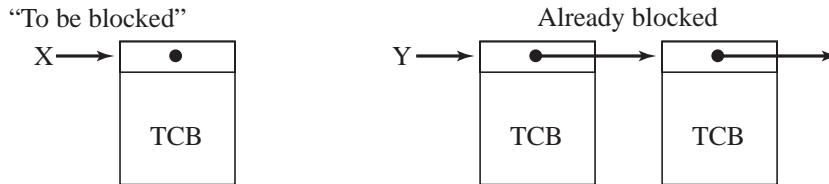
Linked list of threads ready to run before the thread is blocked.

**Figure 5.12**

Linked list at this point in the program.

**Figure 5.13**

TCBs after the thread is blocked (ready-to-run TCBs are not shown).



## 5.4 Thread Synchronization and Communication

This section can be used in two ways. First, it provides a short introduction to the kinds of problems that can be solved using semaphores. In other words, if you have a problem similar to one of these examples, then you should consider a thread scheduler with blocking semaphores as one possible implementation. Second, this section provides the basic approach to solving these particular problems. As stated in the introduction to this chapter, we will generally find that large software systems with more-or-less independent modules can use semaphores when synchronization is needed. On the other hand, for smaller systems with more tightly coupled modules, then the overhead of a thread scheduler will probably not be justified. An important design step when using semaphores is to ascribe a meaning to each semaphore and to each value that semaphore can be.

### 5.4.1 Thread Synchronization or Rendezvous

The objective of this example is to synchronize threads 1 and 2. In other words, whichever thread gets to this part of the code first will wait for the other. Initially semaphores  $S_1$  and  $S_2$  are both 0. The two threads are said to *rendezvous* at the code following the signal and wait calls. The significance of the two semaphores is illustrated in the following table:

$S_1$	$S_2$	Meaning
0	0	Neither thread has arrived at the rendezvous location or both have passed
-1	+1	Thread 2 arrived first and is waiting for thread 1
+1	-1	Thread 1 arrived first and is waiting for thread 2
<b>Thread 1</b>		<b>Thread 2</b>
<code>Signal(&amp;S1);</code>	<code>Signal(&amp;S2);</code>	
<code>Wait(&amp;S2);</code>	<code>Wait(&amp;S1);</code>	

### 5.4.2 Resource Sharing, Nonreentrant Code or Mutual Exclusion

The objective of this example is to share a common resource on a one-at-a-time basis. The critical section (or vulnerable window) of nonreentrant software is that region that should be executed only by one thread at a time. Mutual exclusion means that once a thread has begun executing in its critical section (the `printf()` ; in this example), then no other thread is allowed to execute its critical section. In other words, whichever thread starts to print first will be allowed to finish printing. Either a binary or a counting semaphore can be used. Initially, the semaphore is 1. If  $S=1$ , it means the printer is free. If  $S=0$ , it means the printer is busy and no thread is waiting. If  $S<0$ , it means the printer is busy and one or more threads are blocked.

Thread 1	Thread 2	Thread 3
<code>bWait(&amp;S);</code>	<code>bWait(&amp;S);</code>	<code>bWait(&amp;S);</code>
<code>printf("bye");</code>	<code>printf("tchau");</code>	<code>printf("ciao");</code>
<code>bSignal(&amp;S);</code>	<code>bsignal(&amp;S);</code>	<code>bSignal(&amp;S);</code>

### 5.4.3 Thread Communication Between Two Threads Using a Mailbox

The objective of this example is to send mail from thread 1 to thread 2. The `send` semaphore allows the producer to tell the consumer that new mail is available. The `ack` semaphore is a mechanism for the consumer to tell the producer that the mail was received. Initially, semaphores `send` and `ack` are both 0. The significance of the two semaphores is illustrated in the following table. In this example, the two threads will rendezvous at this point. The “4” represents any data being sent.

send	ack	Meaning
0	0	No mail available, consumer is not waiting
-1	0	No mail available, consumer is waiting for mail
+1	-1	Mail available and producer is waiting for acknowledgment

Producer thread	Consumer thread
<code>Mail=4;</code>	<code>Wait(&amp;send);</code>
<code>Signal(&amp;send)</code>	<code>read(Mail);</code>
<code>Wait(&amp;ack)</code>	<code>Signal(&amp;ack)</code>

### 5.4.4 Thread Communication Between Many Threads Using a FIFO Queue

In the *bounded buffer* problem, we can have multiple threads putting data into a finite-size FIFO and multiple threads getting data out of this FIFO. Bounded buffer is simply another name for the standard FIFO queue we used in Chapter 4 to solve the producer/consumer application. We need two counting semaphores called `CurrentSize` and `RoomLeft` that contain the number of entries currently stored in the FIFO and the number of empty spaces left in the FIFO, respectively. `CurrentSize` is initialized to zero, and `RoomLeft` is initialized to the maximum allowable number of elements in the FIFO. The `Fifo_Put` routine executes the following steps:

```
Wait(&RoomLeft);           /* This will block the thread when FULL */
asm sei                   /* mutually exclusive access to FIFO */
Enter information into the FIFO structure
asm cli
Signal(&CurrentSize);    /* Wake up a thread if blocked on empty */
```

The `Fifo_Get` routine executes the following steps:

```
Wait(&CurrentSize);        /* This will block the thread when EMPTY */
asm sei                   /* mutually exclusive access to FIFO */
Remove information from the FIFO structure
asm cli
Signal(&RoomLeft);       /* Wake up a thread if blocked on full */
```

Another implementation of this bounded buffer problem adds the semaphore called `Mutex` to implement the mutually exclusive access to the FIFO. `Mutex` is initialized to 1. The significance of this semaphore is illustrated in the following table:

Mutex	Meaning
1	No thread is currently entering or removing data
0	One thread is entering or removing data, none are blocked
-1	One thread is entering or removing data, one is blocked

The `Fifo_Put` routine executes the following steps:

```
Wait(&RoomLeft);           /* This will block the thread when FULL */
Wait(&Mutex);             /* Access to FIFO is critical code */
Enter information into the FIFO structure
Signal(&Mutex);
Signal(&CurrentSize);    /* Wake up a thread blocked on empty? */
```

The `Fifo_Get` routine executes the following steps:

```
Wait(&CurrentSize);        /* This will block the thread when EMPTY */
Wait(&Mutex);              /* Access to FIFO is critical code */
Remove information from the FIFO structure
Signal(&Mutex);
Signal(&RoomLeft);         /* Wake up a thread blocked on full? */
```

**Checkpoint 5.11:** On average over the long term, what is the relationship between the number of times `Wait` is called compared to the number of times `Signal` is called?

## 5.5 Fixed Scheduling

In the round-robin scheduler of Section 5.2, the threads were run one at a time and each was given the same time slice. When using semaphores, the thread scheduler dynamically runs or blocks threads depending on conditions at that time. There is another application of thread scheduling sometimes found in real-time embedded systems, which involves a fixed scheduler. In this scheduler, the thread sequence and the allocated time-slices are determined a priori, during the design phase of the project. This class of problems is like creating the city bus schedule, or routing packages through a warehouse. What we do first is create a list of tasks to perform, as follows:

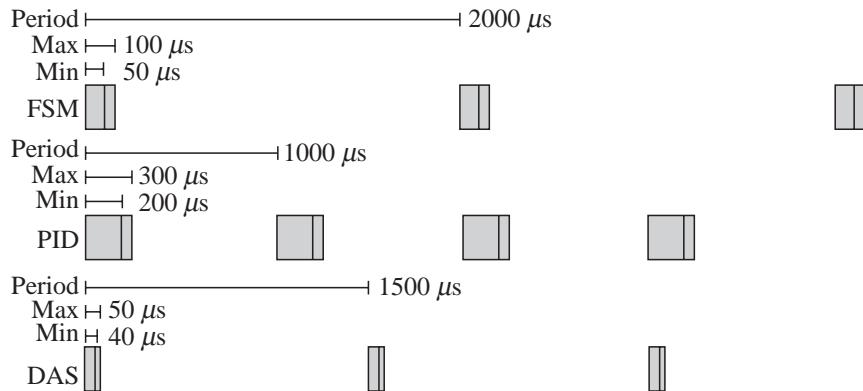
1. Assigning a priority to each task,
2. Defining the resources required for each task,
3. Determining how often each task is to run, and
4. Estimating how long each task will require to complete.

Next, we consider the available resources. Since this chapter deals with thread scheduling, the only resource we will consider here is processor cycles. In more complex systems, we could consider other resources such as memory and I/O channels. For real-time tasks we want to guarantee performance, so we must consider the worst-case estimate of how long each task will take, so the schedule can be achieved 100% of the time. On the other hand, if it is acceptable to meet the scheduling requirement most of the time, we could consider the average time it takes to perform each task. Lastly, we schedule the run times for each task by assigning times for the highest priority tasks first and then shuffle the assignments like placing pieces in a puzzle until all real-time tasks are scheduled as required. The tasks that are not real-time can be scheduled in the remaining slots. If all real-time tasks cannot be scheduled, then a faster microcontroller

will be required. The design of this type of fixed scheduler is illustrated with a design example.

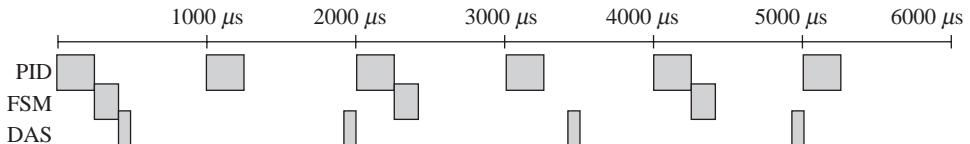
The goal of this design example is to schedule three real-time tasks: a finite state machine (FSM), a proportional-integral-derivative controller (PID), and a data acquisition system (DAS). There will also be one non-real-time task, PAN, which will input/output with the front panel. Figure 5.14 shows that each real-time task in the example has a required period of execution, a maximum execution time, and a minimum execution time.

**Figure 5.14**  
Real-time specifications  
for these three tasks.



Because we wish to guarantee that tasks will always be started on time, we will consider the maximum times. If a solution were to exist, then we will be able find one with a repeating 6000 μs pattern, because 6000 is the least common multiple of 2000, 1000, and 1500. The basic approach to scheduling periodic tasks is to time-shift the second and third tasks so that when the three tasks are combined, there are no overlaps, as shown in Figure 5.15. We start with the most frequent task, which in this example is the PID controller; then we schedule the FSM task immediately after it. In this example, about 41% of the time is allocated to real-time tasks. A solution is possible for this case, because the number of tasks is small and there is a simple 1/1.5/2 relationship between the required periods. Then, we schedule non-real-time tasks in the remaining intervals.

**Figure 5.15**  
Repeating pattern to  
schedule these three  
real-time tasks.



Program 5.19 shows the four threads for this system. The real-time threads execute `os_sleep` when they complete their task, which will suspend the thread and run the non-real-time thread. In this way, each thread will run one time through the `for` loop at the period requirement specified in Figure 5.14. When the threads explicitly release control (in this case by calling `os_sleep`), the system is called **cooperative multitasking**. The non-real-time thread (`PAN`) will be suspended by the timer interrupt, in a manner similar to the preemptive schedule described previously in Section 5.2.

Program 5.20 creates the four thread control blocks. The initial CCR has `I=0`, enabling interrupts. In this system the TCBs are not linked together, but rather exist as a table of

**Program 5.19**

Four user threads.

```

//*****FSM*****
void FSM(void){ StatePtr Pt;
    Pt = SA;                      // Initial State
    DDRT = 0x03;                  // PT1,PT0 outputs, PT3,PT2 inputs
    PTT = Pt->Out;               // Output depends on the current state
    for(;;) {
        OS_Sleep();                // Runs every 2ms
        Pt = Pt->Next[PTT>>2]; // Next state depends on the input
        PTT = Pt->Out;             // Output depends on the current state
    }
}
//*****PID*****
void PID(void){ unsigned char speed,power;
    PID_Init();                   // Initialize
    for(;;) {
        OS_Sleep();                // Runs every 1ms
        speed = PID_In();          // read tachometer
        power = PID_Calc(speed);
        PID_Out(power);           // adjust power to motor
    }
}
//*****DAS*****
void DAS(void){ unsigned char raw;
    DAS_Init();                   // Initialize
    for(;;) {
        OS_Sleep();                // Runs every 1.5ms
        raw = DAS_In();             // read ADC
        Result = DAS_Calc(raw);
    }
}
//*****PAN*****
void PAN(void){ unsigned char input;
    PAN_Init();                   // Initialize
    for(;;) {
        input = PAN_In();          // front panel input
        if(input){
            PAN_Out(input);        // process
        }
    }
}

```

four entries, one for each thread. Each thread will have a total of 100 bytes of stack, and the stack itself exists inside the TCB. The `RunPt` will point to the TCB of the currently running thread.

Program 5.21 defines the data structure containing the details of the fixed scheduler. This structure is a circular linked list, because the schedule repeats. In particular, the 22 entries explicitly define the schedule drawn in Figure 5.15. The front panel thread (`PAN`) is assigned to run in the gaps when no real-time thread requires execution.

The thread scheduler is shown in Program 5.22. The software interrupt creates the cooperative multitasking and is used by the real-time threads when their task is complete. In this example, there is only one non-real-time thread, but it would be straightforward to implement a round-robin scheduler for these threads in the software interrupt handler.

**Program 5.20**

The thread control blocks.

```
struct TCB{
    unsigned char *StackPt;          // Stack Pointer
    unsigned char MoreStack[91];     // 100 bytes of stack
    unsigned char InitialReg[7];     // initial CCR,B,A,X,Y
    void (*InitialPC)(void);        // starting location
};

typedef struct TCB TCBType;
TCBType *RunPt;                  // thread currently running
#define TheFSM &sys[0]           // finite state machine
#define ThePID &sys[1]            // proportional-integral-derivative
#define TheDAS &sys[2]             // data acquisition system
#define ThePAN &sys[3]             // front panel
TCBType sys[4]={
    { TheFSM.InitialReg[0],{ 0 },{0x40,0,0,0,0,0,0},FSM },
    { ThePID.InitialReg[0],{ 0 },{0x40,0,0,0,0,0,0},PID },
    { TheDAS.InitialReg[0],{ 0 },{0x40,0,0,0,0,0,0},DAS },
    { ThePAN.InitialReg[0],{ 0 },{0x40,0,0,0,0,0,0},PAN }
};
```

**Program 5.21**

The scheduler defines both the thread and the duration.

```
struct Node{
    struct Node *Next;              // circular linked list
    TCBType *ThreadPt;             // which thread to run
    unsigned short TimeSlice;      // how long to run it
};

typedef struct Node NodeType;
NodeType *NodePt;
NodeType Schedule[22]={
    { &Schedule[1], ThePID, 300}, // interval      0,   300
    { &Schedule[2], TheFSM, 100}, // interval    300,   400
    { &Schedule[3], TheDAS,  50}, // interval    400,   450
    { &Schedule[4], ThePAN, 550}, // interval    450,  1000
    { &Schedule[5], ThePID, 300}, // interval   1000,  1300
    { &Schedule[6], ThePAN, 600}, // interval   1300,  1900
    { &Schedule[7], TheDAS,  50}, // interval   1900,  1950
    { &Schedule[8], ThePAN,  50}, // interval   1950,  2000
    { &Schedule[9], ThePID, 300}, // interval   2000,  2300
    { &Schedule[10],TheFSM, 100}, // interval   2300,  2400
    { &Schedule[11],ThePAN, 600}, // interval   2400,  3000
    { &Schedule[12],ThePID, 300}, // interval   3000,  3300
    { &Schedule[13],ThePAN, 100}, // interval   3300,  3400
    { &Schedule[14],TheDAS,  50}, // interval   3400,  3450
    { &Schedule[15],ThePAN, 550}, // interval   3450,  4000
    { &Schedule[16],ThePID, 300}, // interval   4000,  4300
    { &Schedule[17],TheFSM, 100}, // interval   4300,  4400
    { &Schedule[18],ThePAN, 500}, // interval   4400,  4900
    { &Schedule[19],TheDAS,  50}, // interval   4900,  4950
    { &Schedule[20],ThePAN,  50}, // interval   4950,  5000
    { &Schedule[21],ThePID, 300}, // interval   5000,  5300
    { &Schedule[0], ThePAN, 700}  // interval   5300,  6000
};
```

**Program 5.22**

The scheduler defines both the thread and the duration.

```

void OS_Sleep(void){ // cooperative multitasking
    asm swi           // suspend this tread and run another
}
interrupt 4 void swiISR(void){
asm ldx RunPt      // cooperative multitasking
asm sts 0,x        // thread goes to sleep when it is done
    RunPt = ThePAN; // non-real time thread
asm ldx RunPt
asm lds 0,x
}
interrupt 11 void threadSwitchISR(void){
asm ldx RunPt
asm sts 0,x
    NodePt = NodePt->Next;
    RunPt = NodePt->ThreadPt; // which thread to run
    TC3 = TC3+NodePt->TimeSlice; // Thread runs for a unit of time
    TFLG1 = 0x08;                // acknowledge by clearing TC3F
asm ldx RunPt
asm lds 0,x
}
void main(void) {
    NodePt = &Schedule[0]; // first thread to run
    RunPt = NodePt->ThreadPt;
    TIOS |= 0x08;          // activate OC3
    TSCR1 = 0x80;          // enable TCNT
    TSCR2 = 0x02;          // usec TCNT
    TIE |= 0x08;           // Arm TC3
    TC3 = TCNT+NodePt->TimeSlice; // Thread runs for a unit of time
    TFLG1 = 0x08;           // Clear C3F
asm ldx RunPt
asm lds 0,x
asm rti    // Launch First Thread
}

```

**Checkpoint 5.12:** Why does the thread switch Program 5.22 execute `sts 0,x`, while the thread switches in Programs 5.5 and 5.8 execute `sts 2,x`?

We could have attempted to implement this system with regular periodic interrupts. In particular, we could have created three independent periodic interrupts and performed each task in a separate ISR. Unfortunately, there would be situations when one or more tasks would overlap. In other words, one interrupt might be requested while we are executing one of the other two ISRs. Although all tasks would run, some would be delayed. This delay is called **time-jitter**, which is defined as the difference between the time a thread is supposed to run (see comments of Program 5.21) and the time it does run. This solution presented in Programs 5.19 to 5.22 will create a situation in which the interrupts are always enabled when the C3F flag gets set; therefore, no real-time tasks are delayed. The measured time-jitter for this system is always less than 1  $\mu$ sec. The origin of the error is the variability in which instruction of `PAN()` is being executed at the time of the output compare interrupt request.

**Checkpoint 5.13:** What would be the effect on time-jitter by activating the PLL in the MC9S12C32 so that the computer runs six times faster (24MHz) and multiplying the TimeSlice constants in Program 5.21 by six?

According to the Rate Monotonic theorem, we should have been able to schedule these tasks because

$$\sum_{i=0}^{n-1} \frac{E_i}{T_i} = \sum \frac{100}{2000} + \frac{300}{1000} + \frac{50}{1500} = 0.38 \quad n(2^{1/n} - 1) = (3^{1/3} - 1) = 0.78$$

## 5.6. OS Considerations for I/O Devices

**5.6.1 Board Support Package** The entire book deals with interfacing I/O devices to build embedded systems. However, in this section, we will study two considerations of how the OS can manage I/O. As presented earlier in Section 2.6, it is good design practice to provide an abstraction for the I/O layer. Names for this abstraction include hardware abstraction layer (HAL), device driver, and board support package (BSP). From an operating system perspective, the goal is to make it easier to port the system from one hardware platform to another. The system becomes more portable if we create a BSP for our hardware devices. A BSP could allow you to encapsulate the following functionality:

- Timer initialization
- ISR handlers
- LED output functions
- Switch input functions
- Setting up the interrupt controller
- Setting up communication channel
- CAN, I2C, ADC, DAC, SPI, serial, graphics

**Example 5.1** Design a BSP for using a periodic interrupt.

**Solution** In any abstraction, we need to separate what the system does from how it does it. What we use a periodic interrupt for is to run a task at a fixed rate. How we do it on the 9S12 is to enable the timer and configure one of the output compare interrupts, as presented previously in Section 4.14. What the user needs is an OS function that he or she can call specifying their task and how often it should run. We can abstract the periodic interrupt, by defining the function in Program 5.23, which is essentially Program 4.31 with the flexibility to specify the task to run and the period with which to run it. We have hidden from the user the fact that we are running on a 9S12 at 24 MHz. To run the function Task once a second, the user calls `OS_AddPeriodicTask(1000, &Task);`

### Program 5.23

RTOS function to run a periodic task.

```
unsigned short static volatile Count;
unsigned short static Period;
void (*CallBack)(void); // call back function
interrupt 11 void TC3handler(void){ // executes at 1000 Hz
    TFLG1 = 0x08; // acknowledge OC3
    Count++;
    TC3 = TC3+1500; // 1 ms
    if(Count==Period){
        Count = 0;
        (*CallBack)(); // execute call back process
    }
}
```

*continued on p. 278*

**Program 5.23**

RTOS function to run a periodic task.

*continued from p. 277*

```
----- OS_AddPeriodicTask -----
// Input: thePeriod is a time period in ms
//          fp is a function to be executed at this period
// Output: none
// Example: to toggle PT0 once a second, we can
// void toggle(void){PTT^=0x01;};
// OS_AddPeriodicTask(1000,&toggle);
void OS_AddPeriodicTask (unsigned short thePeriod, void(*fp)(void)){

    asm sei      // make ritual atomic
    Period = thePeriod;
    CallBack = fp;
    Count = 0;
    TIOS |= 0x08;    // Activate TC3 as output compare
    TSCR1 = 0x80;    // Enable TCNT at 1.5 MHz, PLL active so E=24MHz
    TSCR2 = 0x04;    // Divide by 16 TCNT prescale, TOI disarm
    TIE |= 0x08;    // Arm OC3
    TC3 = TCNT+50; // First interrupt right away
    asm cli
}
```

**Example 5.5** Design a BSP for the LEDs in Example 1.1.

**Solution** Again, we need to separate what the system does from how it does it. We can turn LEDs on and off. In this example, the four LEDs constitute one 4-bit device, so we will organize the solution in that manner, as shown in Program 5.24. Again, we have hidden from the user the fact that we are running on a 9S12 using Port T.

**Program 5.24**

BSP for four LEDs.

```
-----OS_LEDInit -----
// Initialize the set of 4 LEDs
// Input: none
// Output: none
void OS_LEDInit(void){
    DDRT |= 0x0F; // make PT3-0
}
-----OS_LED_Out -----
// Output to the 4 LEDs
// Input: number from 0 to 15, specifying which LEDs are on/off
// Output: none
void OS_LEDOut(unsigned char number){
    PTT = (PTT&0xF0)|(number&0x0F); // friendly access
}
```

## Path Expression

**5.6.2 Path expression** is a formal mechanism to specify the correct calling order in a group of related functions. Consider a SCI device driver with four functions (similar to Program 3.10); the prototypes are:

```

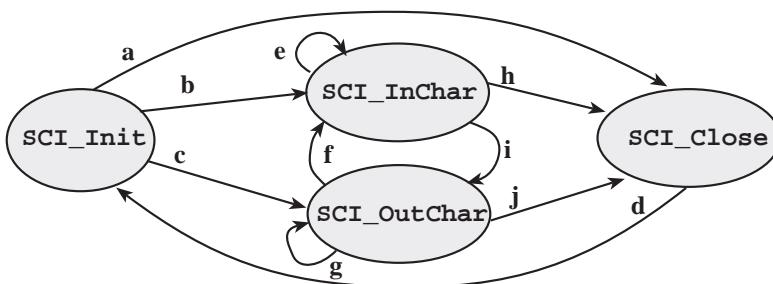
void SCI_Init(void);           // Initialize Serial port
char SCI_InChar(void);        // Wait for new serial port input
void SCI_OutChar(char data);   // Output 8-bit to serial port
void SCI_Close(void);         // Shut down serial port

```

It is obvious that you should not attempt to input/output until the SCI is initialized. In this problem, we will go further and actually prevent the user from executing `SCI_InChar` and `SCI_OutChar` before executing `SCI_Init`. A directed graph (Figure 5.16) is a general method to specify the valid calling sequences. An arrow represents a valid calling sequence within the path expression. The system “state” is determined by the function it called last. For this example, we begin in the closed state, because the SCI is initially disabled. The tail of an arrow touches the function we called last, and the head of an arrow points to a function that we are allowed to call next.

**Figure 5.16**

Directed graph showing path expression for the serial port driver.



In this method, a calling sequence is valid if there is sequence of arrows to define it. For example,

Init InChar InChar OutChar Close	d b e i j
Init OutChar OutChar OutChar OutChar	d c g g g
Init Close Init InChar Close	d a d b h

On the other hand, the following calling sequences are illegal because each has no representative sequence of arrows.

Init InChar Init OutChar Close	Can't initialize twice
Close	Can't close because already disabled
OutChar OutChar OutChar	Can't output without initialization

A fast, but memory-inefficient, method to represent a directed graph uses a square matrix. Since there are four functions, the matrix will be 4 by 4. The row number (0,1,2,3) will specify the current state (the function called last), and the column number (0,1,2,3) will specify the function that might be called next. The values in the matrix are true(1)/false(0) specifying whether or not the next function call is legal. Since there are 10 arrows in the directed graph, there will be exactly 10 true values in the matrix—one for each arrow. The remaining values will be false(0). Program 5.25 shows the data structure for the directed graph. At the beginning of each call to the serial port driver, the OS checks to verify the user has permission to execute that function. The global variable `State` defines the current state. For example, `Path[3][0]` will be true, signifying it is okay to call `SCI_Init`, if the SCI is disabled. We assume there is an operating system function called `OS_Kill()`, which should be called if a thread makes an illegal function call, destroying the thread because it has made a serious programming error.

**Program 5.25**

Directed graph showing path expression for the serial port driver.

```

int State=3; // start in the Closed state
int Path[4][4]={ /* Init InChar OutChar Close */
/* column 0 1 2 3 */
/* Init row 0 */ { 0 , 1 , 1 , 1 },
/* InChar row 1 */ { 0 , 1 , 1 , 1 },
/* OutChar row 2 */ { 0 , 1 , 1 , 1 },
/* Close row 3 */ { 1 , 0 , 0 , 0 } }
void SCI_Init(void){
    if(Path[State][0]==0) OS_Kill(); // kill if illegal
    State = 0; // perform valid Init
    SCIBD = 13;
    SCICR1 = 0;
    SCICR2 = 0x0C;
}
char SCI_InChar(void){
    if(Path[State][1]==0) OS_Kill(); // kill if illegal
    State = 1; // perform valid InChar
    while((SCISR1 & RDRF) == 0){};
    return(SCIDRL);
}
void SCI_OutChar(char data){
    if(Path[State][2]==0) OS_Kill(); // kill if illegal
    State = 2; // perform valid OutChar
    while((SCISR1 & TDRE) == 0){};
    SCIDRL = data;
}
void SCI_Close(void){
    if(Path[State][3]==0) OS_Kill(); // kill if illegal
    State = 3; // perform valid Close
    SCICR2 = 0x00;
}

```

## 5.7 Exercises

- 5.1** Select the best term from the chapter that describes each definition.
- A technique to periodically increase the priority of low-priority threads so that low-priority threads occasionally get run. The increase is temporary.
  - A situation that can occur in a priority thread scheduler where a low-priority thread never runs.
  - The condition where Thread 1 is waiting for a unique resource held by Thread 2, and Thread 2 is waiting for a unique resource held by Thread 1.
  - The condition where a thread is not allowed to run because it needs something that is unavailable.
  - The condition where once a thread blocks, there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed.
  - An operation that once started will run to completion without interruption
  - An implementation using a FIFO or mailbox that separates data input from data processing.
  - A technique that could be used to prevent the user from executing I/O on a driver until after the user calls the appropriate initialization.
  - A scheduling algorithm that assigns priority linearly related to how often a thread needs to run. Threads needing to run more often have a higher priority.
  - An OS feature that allows the user to run user-defined software at specific places within the OS. These programs are extra for the user's convenience and not required by the OS itself.
  - An OS feature that allows you to use the OS in safety-critical applications.

- l)** A scheduling algorithm with priority but varying time slice. If a thread blocks on I/O, its time slice is reduced. If it runs to completion of a time slice, its time slice is increased.
- m)** The condition where (at most) one thread is allowed access to a resource that can not be shared. If a second thread wishes access to the resource while the first thread is using it, the second thread is made to wait until the first thread is finished.
- n)** The condition a function has that allows it to be simultaneously executed by multiple threads.
- o)** A thread scheduling algorithm that has the threads themselves decide when the thread switches should occur.
- p)** A situation that can occur in a priority thread scheduler where a high-priority thread is waiting on a resource owned by a low-priority thread.
- q)** A type of semaphore implemented with a busy-wait loop.
- r)** A type of thread scheduler where each thread has equal priority and all threads are executed in a circular sequence.

**5.2** For each of the following terms give a definition in 32 words or less

- |                                 |                                     |                                 |
|---------------------------------|-------------------------------------|---------------------------------|
| <b>a)</b> active                | <b>j)</b> deadlock                  | <b>r)</b> starvation            |
| <b>b)</b> aging                 | <b>k)</b> earliest slack time first | <b>s)</b> preemptive scheduler  |
| <b>c)</b> atomic                | <b>l)</b> exponential queue         | <b>t)</b> producer-consumer     |
| <b>d)</b> blocked               | <b>m)</b> hook                      | <b>u)</b> rate monotonic        |
| <b>e)</b> bounded buffer        | <b>n)</b> maximum latency           | <b>v)</b> reentrant             |
| <b>f)</b> bounded waiting       | <b>o)</b> nonreentrant              | <b>w)</b> rendezvous            |
| <b>g)</b> breakdown utilization | <b>p)</b> normalized mean           | <b>x)</b> round-robin scheduler |
| <b>h)</b> certification         | response time                       | <b>y)</b> sleeping              |
| <b>i)</b> critical section      | <b>q)</b> path expression           | <b>z)</b> spin lock             |

**5.3** Consider the following implementation of a spinlock semaphore used with a round-robin preemptive scheduler.

```
void OS_Wait(char *semaPt){
    asm sei           // Test and set is atomic
    while(*semaPt <= 0){ // disabled
        asm cli         // disabled
        asm nop         // enabled
        asm sei         // enabled
    }
    (*semaPt->Counter)--; // disabled
    asm cli         // disabled
}
void OS_Signal(char *semaPt){
    (*semaPt)++; // enabled
}
```

**a)** The compiler generated this code for os\_Signal

0000 b745	[1]	TFR	D,X
0002 6200	[3]	INC	0,X
0004 3d	[5]	RTS	

Are there any critical sections in os\_Signal? That is, can two threads both call os\_Signal? Justify your answer.

**b)** Design a new op-code for the 9S12, and use it to rewrite os\_Wait so the spin lock semaphore can be executed without critical sections and without disabling interrupts. Be VERY specific about what the new instruction does (e.g., addressing modes, condition code bits). The 16-bit pointer, semaPt, will be in Register D when os\_Wait is called.

**5.4** We can use semaphores to limit access to resources. In the following example, both threads need access to a printer and a SPI port. The binary semaphore sPrint provides mutual exclusive

access to the printer, and the binary semaphore `sSPI` provides mutual exclusive access to the SPI port. The following scenario has a serious flaw:

**Thread 1**

```
bwait(&sPrint);
bwait(&sSPI);
printf("bye");
OutSPI(6);
bsignal(&sPrint);
bsignal(&sSPI);
```

**Thread 2**

```
bwait(&sSPI);
bwait(&sPrint);
OutSPI(5);
printf("tchau");
bsignal(&sSPI);
bsignal(&sPrint);
```

- a) Describe the sequence of events that would lead to the condition where both threads are stuck forever.
- b) How could you change the above program to prevent the deadlock?

- D5.5** Some computers have a `swap` instruction. Modify the subroutines shown in Program 5.14, written in 9S12 assembly language, to implement a binary spin-lock semaphore using this fictional instruction. The system must provide for mutual exclusion. You may not disable interrupts. Do not use the `minm` instruction. The syntax of this fictional atomic operation is

`swap Operand, r`

where `Operand` is any standard 9S12 addressing mode and `r` is any of the 9S12 registers. This instruction exchanges the 8- or 16-bit contents of the `Operand` and data register. It does not set any condition code bits.

- D5.6** In this exercise you will extend the preemptive scheduler to support priority. This system should support three levels of priority: 1 will be the highest. You can solve this problem using either assembly or C.

- a) Redesign the TCB to include a 16-bit integer for the priority (although the values will be restricted to 1, 2, 3). Show the static allocation for the three threads from the example in this chapter, assuming the first two are priority 2 and the last is priority 3. There are no priority 1 threads in this example, but there might be in the future.
- b) Redesign the scheduler to support this priority scheme.
- c) In the chapter it said “*Normally, we add priority to a system that implements blocking semaphores and not to one that uses spin-lock semaphores.*” What specifically will happen here if the system is run with spin-lock semaphores?
- d) Even when the system supports blocking semaphores, starvation might happen to the low-priority threads. Describe the sequence of events that causes starvation.
- e) Suggest a solution to the starvation problem.

- D5.7** You are given four identical I/O ports to manage on the MC68HC12A4: Port A, Port B, Port C, and Port D. You may assume there is a preemptive thread scheduler and blocking semaphores.

- a) Look up the address of each port and its direction register.
- b) Create a data structure to hold an address of the port and the address of the data direction register. Assume the type of this structure is called `PortType`.
- c) Design and implement a manager that supports two functions. The first function is called `NewPort`. Its prototype is

```
PortType *NewPort(void);
```

If a port is available when a thread calls `NewPort`, then a pointer to the structure, defined in part b, is returned. If no port is available, then the thread will block. When a port becomes available, this thread will be awakened and the pointer to the structure will be returned. You may define and use blocking semaphores without showing the implementation of the semaphore or scheduler. The second function is called `FreePort`, and its prototype is

```
void FreePort(PortType *pt);
```

This function returns a port so that it can be used by the other threads. Include a function that initializes the system, where all five ports are free. (*Hint:* The solution is very similar to the FIFO queue example shown in Section 5.4.4.)

**D5.8** Consider a problem of running three foreground threads using a preemptive scheduler with semaphore synchronization. Each thread has a central body( ) containing code that should be executed together. The basic shell of this system is given. Define one or more semaphores, then add semaphore function calls to implement a *three-thread rendezvous*. Basically, each time through the while loop, the first two threads to finish their start( ) code will wait for the last thread to finish its start( ) code. Then, all three threads will be active at the same time as they execute their corresponding body( ). You may call any if the semaphore functions is defined in Section 5.3. You will allocate one or more semaphores and add calls to semaphore functions, otherwise no other changes are allowed. You may assume thread1 runs first. For each semaphore you add, explain what it means to be 0, 1, etc.

```
void thread1(void){    void thread2(void){    void thread3(void){
    init1();            init2();            init3();
    while(1){
        start1();          start2();          start3();
        body1();           body2();           body3();
        end1();            end2();            end3();
    }
}
```

**D5.9** The goal of this problem is to design a cooperative thread switcher that runs Program 5.3. There will be no interrupts whatsoever, just the SWI instruction that causes a software interrupt. The main program creates three threads and launches the first one, similar to Program 5.8, but without the output-compare interrupt. The threads are chained in a circle using the Next pointers in the TCB. All threads will cooperate by calling your OS\_Suspend( ) function regularly. The thread control block described in Section 5.2 will be used.

- Write the function OS\_Suspend, which issues a SWI.
- Write the SWI interrupt handler that suspends the current thread and runs the next thread in the circular linked list.

**D5.10** Consider a problem of deadlocks that can occur with semaphore synchronization. The following is a classic example that might occur if two threads need both the disk and the printer. In this example, the disk has a binary semaphore DiskFree, which is 1 if the disk is available. Similarly the printer has a binary semaphore PrinterFree, which is 1 if the printer is available. A deadlock occurs if each thread gets one resource then waits (on each other) for the other resource. In this example, we assume there is one disk and one printer.

```
void thread1(void){                void thread2(void){
    OS_bWait(&DiskFree);          OS_bWait(&PrinterFree);
    OS_bWait(&PrinterFree);      OS_bWait(&DiskFree);

    // use disk and printer          // use printer and disk

    OS_bSignal(&DiskFree);        OS_bSignal(&PrinterFree);
    OS_bSignal(&PrinterFree);    OS_bSignal(&DiskFree);
}
```

In this problem, we will develop a graphical method (called a *resource allocation graph*) to visualize/recognize the deadlock. Draw each thread in your system as an oval and each binary semaphore as a rectangle. If a thread calls OS\_bWait and returns, then draw an arrow (called an *allocation edge*) from the semaphore to the thread. An arrow from a semaphore to a thread means that thread owns the resource. If a thread calls OS\_bSignal, then erase the previously drawn allocation edge. If a thread calls OS\_bWait and spins or blocks because the semaphore is not free, then draw an arrow from the thread to the semaphore (called a *request edge*). An arrow from a thread to a semaphore means that thread is waiting for the resource associated with the semaphore.

- Draw the resource allocation graph that occurs with the deadlock sequence
  - Thread1 executes OS\_bWait(&DiskFree);
  - Thread2 executes OS\_bWait(&PrinterFree);

- 3) Thread2 executes `OS_bWait(&DiskFree);`  
 4) Thread1 executes `OS_bWait(&PrinterFree);`
- b) This method can be generalized to detect that a deadlock has occurred with an arbitrary number of binary semaphores and threads. What shape in the resource allocation graph defines a deadlock? In other words, generalize the use of this method such that you can claim  
*"There is a deadlock if and only if the resource allocation graph contains a shape in the form of a \_\_\_\_\_."*
- c) Justify your answer by giving a deadlock example with three threads and three binary semaphores. In particular, give
- 1) The C code;
  - 2) The execution sequence;
  - 3) The resource allocation graph.

**D5.11** Consider a system with two LCD message displays in the context of a preemptive thread scheduler with blocking semaphores. To display a message, the OS can call either `LCD1_OutString` or `LCD2_OutString`, passing it an ASCII string. These routines have critical sections but must run with interrupts enabled. The foreground threads will not call `LCD1_OutString` or `LCD2_OutString` directly; rather, the threads call a generic OS routine `OS_Display`. If an LCD is free, the OS passes the message to the free LCD. If both LCDs are busy, the thread will block. There are many threads that wish to display messages, and the threads do not care or know onto which LCD their message will be displayed. You are given the `LCD1_OutString` or `LCD2_OutString` routines, the OS, and the blocking semaphores with the following prototypes.

```
void LCD1_OutString(char *string); // up to 20ms to complete
void LCD2_OutString(char *string); // up to 20ms to complete
int OS_InitSemaphore(Sema4Type *semaPt, short value);
void OS_Wait(Sema4Type *semaPt);
void OS_Signal(Sema4Type *semaPt);
```

- a) List the semaphores and private global variables needed for your solution. For each semaphore, define what it means and what initial value it should have. Give the meaning and initial values for any private global variables you need. The threads will not directly access these semaphores or variables.
- b) Write the generic OS display routine that the foreground threads will call (you may not disable interrupts or call any other functions other than the five functions shown above).

```
void OS_Display(char *string){
```

**D5.12** Consider a system that employs a preemptive real-time OS. There are multiple threads that need to update a shared LCD display. Consider this example with two foreground threads (`thread1` `thread2`) and one background thread (`isr`) that all output to a three-line LCD. `Free` is a global variable, initialized to 1.

```
void thread1(void){ void thread2(void){ interrupt 15
unsigned short data; unsigned short data; void isr(void){
  init1();           init2();           unsigned short data;
  for(;;){          for(;;){          data = calc3();
    data = calc1();   data = calc2();   Display(3,data);
    Display(1,data);  Display(2,data); TFLG1 = 0x80;
  }                  }                  }
}                  }
```

The first parameter of `Display` is the line number, and the second parameter is a 16-bit number. You may call the two `LCD_GoTo` `LCD_OutDec` functions without writing them. Your `Display` function will effectively perform the following (this program has a critical section).

```

unsigned char Free=1;
void Display(int line, unsigned short num){
    if(Free){Free=0; LCD_GoTo(line,1); LCD_OutDec(num); Free = 1;}
}

```

Rewrite this `Display` function to remove the critical sections. You CANNOT disable interrupts at all. You should not introduce new critical sections. You CANNOT allow threads to block or spin. If the LCD is busy, then the output is simply skipped. You may not change the thread code or the `Display` function prototype. Basically you will add software to this existing `Display` function, but not `thread1`, `thread2`, or `isr`.

- D5.13** This problem investigates the design of an adaptive priority scheduler with exponential time slices. This is also called an *exponential Queue* or *multi-level feedback queue*. The CTSS system (MIT, early 1960s) was the first to use exponential queues. One of the difficulties in a priority scheduler is the assignment of priority. Typically, one wishes to assign a high priority to threads doing I/O (which block a lot) so that the response to I/O is short, and assign a low priority to threads not doing I/O (which do not block a lot). However, in a complex system, a particular thread may sometimes exhibit I/O bound behavior but later exhibit CPU bound behavior. An adaptive scheduler will adjust the priority according to the current activity of the thread. Priority 1 threads will run with a time slice of 4000 (1ms), priority 2 threads will run with a time slice of 8000 (2ms), and priority 3 threads will run with a time slice of 16000 (4ms). Consider this blocking round-robin scheduler, with two new entries, shown in **bold**, added to the TCB.

```

struct TCB{
    struct TCB *Next;      // Link to Next TCB
    char *StackPt;         // Stack Pointer
    Sema4Type *BlockPt;   // 0 if not blocked, pointer if blocked
    short Priority;     // 1 (highest), 2, or 3 (lowest)
    unsigned short TimeSlice; // 4000,8000, or 16000
    char Stack[100];       // stack, size determined at runtime
};

typedef struct TCB TCBType;
typedef TCBType * TCBPtr;

```

- a)** Rewrite the `OS_Wait` function so that if a priority 2 or 3 thread blocks, its priority will be raised (decrement by 1) and its time slice will be halved. No changes to `OS_Signal` will be needed.

```

void OS_Wait(Sema4Type *semaPt){
    asm sei           // Test and set is atomic
    (semaPt->Value)--;
    if(semaPt->Value <=0){
        RunPt->BlockPt = semaPt; // block this thread
        TC3=TCNT+15;           // stop running this thread
    }
    asm cli
}

```

- b)** Rewrite the `threadSwitch` ISR so that if a priority 1 or 2 thread runs to the end of its time slice without blocking, its priority will be lowered (increment by 1), and its time slice will be doubled. In addition, implement priority scheduling with variable time slices.

```

// this is a round robin scheduler with fixed timeslices
interrupt 11 void threadSwitch(void){
asm ldx RunPt
asm sts 2,x
    RunPt = RunPt->Next;
    while(RunPt->BlockPt){
        RunPt = RunPt->Next; // don't run blocked threads
    }
}

```

```

TC3 = TCNT+4000;           // Thread runs for a unit of time
TFLG1 = 0x08;               // acknowledge by clearing TC3F
asm ldx RunPt
asm lds 2,x
}

```

**D5.14** Consider the implementation of OS\_AddThread, shown in Program 5.11. Redesign the system so that if the user program finishes, the OS will run the user program again. For example, this user function executes stuff1, stuff2, and stuff3 once and quits.

```
void user(void){ stuff1(); stuff2(); stuff3();}
```

If the user calls this system function to activate user,

```
OS_AddThread(&user);
```

then with your updated system stuff1, stuff2, and stuff3 will be repeated over and over again. You are allowed to make changes to the struct and to OS\_AddThread but not to user or other OS functions. You can however add additional OS functions. In particular, show changes to the struct and rewrite OS\_AddThread in its entirety.

## 5.8 Lab Assignments

**Lab 5.1** The overall objective is to create a preemptive thread scheduler with blocking semaphores. There is a preemptive scheduler with spinlock semaphores on the website for the book, in the Lab17 folder. You will first adjust the FIFO sizes so no data is lost. Then, you will redesign the system to implement blocking semaphores.

**Lab 5.2** The overall objective is to create a fixed thread scheduler for three real-time tasks and two non-real-time tasks. An implementation of the fixed scheduler developed in Section 5.5 can be found as the FixedScheduler example at <http://www.ece.utexas.edu/~valvano/metrowerks/>. First, you will create a second non-real-time thread similar to PAN, but using other I/O pins. Next, you will calculate the maximum time to execute each loop of the real-time threads. Then, you will develop a fixed scheduler with the following specification:

FSM period is 100 ms  
 PID period is 75 ms  
 DAS period is 10 ms

The swi handler should alternate starting the two non-real-time threads. You can use switches for inputs and LEDs for outputs so that you can interact with the system.

**Lab 5.3** The overall objective is to write an algorithm to search for the best schedule for a fixed-rate scheduler like the one presented in Section 5.5. Assume there are four tasks. Task A runs every  $T_A$  with a maximum time of  $E_A$ . Task B runs every  $T_B$  with a maximum time of  $E_B$ . Task C runs every  $T_C$  with a maximum time of  $E_C$ . Task D runs every  $T_D$  with a maximum time of  $E_D$ . You may assume all the time specifications are integers with units of 0.1 ms. For example, if Task A runs every 10 ms with a maximum time of 1.2ms, then  $T_A$  is 100 and EA is 12. The basic approach is to define slide times i, j, and k such that we schedule

Task A at times  $n*T_A$   
 Task B at times  $n*T_B + j$   
 Task C at times  $n*T_C + k$   
 Task D at times  $n*T_D + l$

with the desire that no two tasks be scheduled at the same time. If two tasks overlap one of them is delayed by 0.1ms, causing jitter. The total jitter for the scheduling algorithm (choice of i, j, k) is the total number of 0.1 time-quanta a task is delayed. In essence this lab will search the space of all (i, j, k) to find values with minimum jitter.

First test case:

Task A runs every 1.0 ms, maximum time is 0.1ms

Task B runs every 1.5 ms, maximum time is 0.1ms

Task C runs every 2.5 ms, maximum time is 0.1ms

Task D runs every 3.0 ms, maximum time is 0.1ms

Pattern repeats every 15ms

$$E_A/T_A + E_B/T_B + E_C/T_C + E_D/T_D = 0.24$$

This first test case has a solution with i=1, j=2, and k=3 that has no jitter.

Second test case:

Task A runs every 0.4 ms, maximum time is 0.1ms

Task B runs every 0.6 ms, maximum time is 0.1ms

Task C runs every 1.0 ms, maximum time is 0.1ms

Task D runs every 1.5 ms, maximum time is 0.1ms

Pattern repeats every 9ms

$$E_A/T_A + E_B/T_B + E_C/T_C + E_D/T_D = 0.58$$

This second test case has the best solution with i=1, j=1, and k=14, but has a jitter of 5 time-quanta.

# 6 Timing Generation and Measurements

## Chapter 6 objectives are to use:

- ❖ Input capture to generate interrupts and measure period or pulse width
- ❖ Output compare to create periodic interrupts, generate square waves, and measure frequency
- ❖ Both input capture and output compare to make flexible and robust measurement systems
- ❖ Pulse accumulator to measure frequency and period
- ❖ Pulse-width modulator to generate waveforms

The timer systems on the state-of-the-art microcontrollers are very versatile. Over the last 30 years, the evolution of these timer functions has paralleled the growth of new applications possible with these embedded computers. In other words, inexpensive yet powerful embedded systems have been made possible by the capabilities of the timer system. In this chapter we will introduce these functions, then use them throughout the remainder of the book. If we review the applications introduced in the first chapter (see Section 1.1), we will find that virtually all of them make extensive use of the timer functions developed in this chapter.

## 6.1 Input Capture

### 6.1.1 Basic Principles of Input Capture

We can use input capture to measure the period or pulse width of digital logic signals. The input capture system can also be used to trigger interrupts on rising or falling transitions of external signals. TCNT is a 16-bit counter incremented at a fixed rate. See Figure 6.1. Each input capture module has

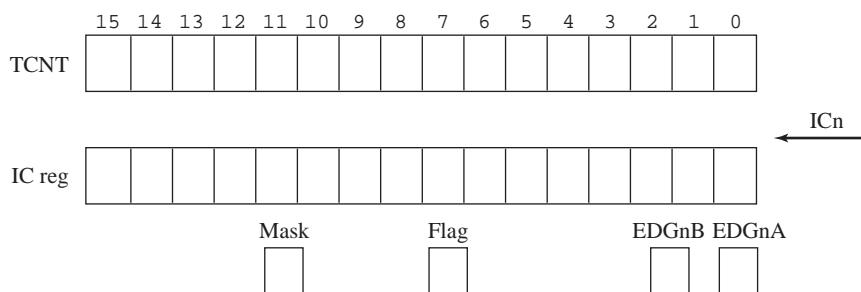
- An external input pin, IC<sub>n</sub>,
- A flag bit,
- Two edge control bits, EDGnB EDGnA,
- An interrupt mask bit (arm), and
- A 16-bit input capture register.

The various members of the Freescale 9S12 family have from two to sixteen input capture modules. However, most 9S12 microcontrollers have eight channels. In this book we use the term **arm** to describe the bit that allows or denies a specific flag from requesting an interrupt. The Freescale manuals refer to this bit as a **mask**. That is, the device is armed when the mask bit is 1. Typically, there is a separate arm bit for every flag that can request an interrupt.

An external input signal is connected to the input capture pin. The EDGnB, EDGnA bits specify whether the rising, falling, or both rising and falling edges of the external signal will trigger an input capture event. Two or three actions result from an input capture event: 1) the current TCNT value is copied into the input capture register, 2) the input

**Figure 6.1**

Basic components of input capture.



capture flag is set, and 3) an interrupt is requested if the mask bit is 1. This means an interrupt can be requested on a capture event. The input capture mechanism has many uses. Three common applications are:

1. An interrupt service routine is executed on the active edge of the external signal,
2. Perform two rising edge input captures and subtract the two measurements to get the period,
3. Perform a rising edge then a falling edge capture and subtract the two measurements to get the pulse width.

The flag bit does not behave like a regular memory location. In particular, the flag cannot be set by software. Rather, an input capture or output compare hardware event will set the flag. The other peculiar behavior of the flag is that the software must write a one to the flag in order to clear it. If the software writes a zero to the flag, no change will occur.

**Checkpoint 6.1:** When does an input capture event occur?

**Checkpoint 6.2:** What happens during an input capture event?

**Observation:** The TCNT timer is very accurate because of the stability of the crystal clock.

**Observation:** When measuring period or pulse width, the measurement resolution will equal the TCNT period.

## 6.1.2 Input Capture Details

Next we will overview the specific input capture functions on the 9S12. This section is intended to supplement rather than replace the Freescale manuals. When designing systems with input capture, please refer to the reference manual of your specific Freescale microcomputer.

Table 6.1 shows the 9S12 registers used for input capture. The entries shown in bold will be used in this section. **TCNT** is a 16-bit unsigned counter that is incremented at a rate determined by three bits (PR2, PR1, and PRO) in the **TSCR2** register. To use any of the features involving TCNT, such as input capture, we have to set the **TEN** bit in the **TSCR1** register. Table 6.2 shows the available TCNT periods for three different E clock periods. The fundamental approach to input capture involves connecting an external TTL-level signal to one of the eight input capture pins on Port T, as shown in Figure 6.2. The pin is selected as input capture by placing a 0 in the corresponding bit of the **TIOS** register. There is a direction register, **DDRT**, for port T, which means we should clear the corresponding bits for the input capture inputs. The input capture event occurs on the rising, falling, or both rising and falling edge of this external signal. The 9S12 will set the input capture flag (**CnF** in the **TFLG1** register) and latch the current 16-bit TCNT value into the input capture latch (TCn). If the input capture flag is armed (**CnI** in the **TIE** register), then an interrupt will be requested when the flag is set. There is a separate 16-bit input capture register for each of the eight input capture modules. We specify the active edge (i.e., the edge that latches TCNT and sets the flag) by initializing the **TCTL3** and **TCTL4** registers, as described in

Address	msb																lsb	Name
\$0044	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TCNT	
\$0050	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC0	
\$0052	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC1	
\$0054	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC2	
\$0056	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC3	
\$0058	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC4	
\$005A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC5	
\$005C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC6	
\$005E	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC7	

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
\$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
\$0046	TEN	TSWAI	TSBCK	TFFCA	0	0	0	0	TSCR1
\$004D	TOI	0	0	0	TCRE	PR2	PR1	PR0	TSCR2
\$0040	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0	TIOS
\$004C	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I	TIE
\$004E	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F	TFLG1
\$004F	TOF	0	0	0	0	0	0	0	TFLG2
\$004A	EDG7B	EDG7A	EDG6B	EDG6A	EDG5B	EDG5A	EDG4B	EDG4A	TCTL3
\$004B	EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B	EDG0A	TCTL4

**Table 6.1**

9S12 registers used for input capture.

**Table 6.2**

Three control bits define the TCNT clock period.

PR2	PR1	PR0	Divide by	E = 4 MHz	E = 8 MHz	E = 24 MHz
				TCNT Period	TCNT Period	TCNT Period
0	0	0	1	250 ns	125 ns	42 ns
0	0	1	2	500 ns	250 ns	83 ns
0	1	0	4	1 µs	500 ns	167 ns
0	1	1	8	2 µs	1 µs	333 ns
1	0	0	1	6	4 µs	667 ns
1	0	1	3	2	8 µs	1.33 µs
1	1	0	64	16 µs	8 µs	2.67 µs
1	1	1	128	32 µs	16 µs	5.33 µs

**Figure 6.2**

Input capture interface on the 9S12.

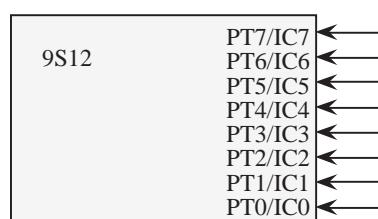


Table 6.3. We can arm or disarm the input capture interrupts by initializing the TIE register. Our software can determine whether an input capture event has occurred by reading the TFLG1 register. Every time the TCNT register overflows from \$FFFF to 0, the TOF flag in the TFLG2 register is set. The TOF flag will cause an interrupt if the mask TOI equals 1, which is described in Section 4.14.

**Table 6.3**

Two control bits define the active edge used for input capture.

EDGnB	EDGnA	Active edge
0	0	None
0	1	Capture on rising
1	0	Capture on falling
1	1	Capture on both rising and falling

**Observation:** The phase-lock-loop (PLL) on the 9S12 will affect the TCNT period.

**Checkpoint 6.3:** List the steps to initialize PT0 as an armed rising edge input capture.

**Checkpoint 6.4:** List the steps to initialize PT3 as an armed falling edge input capture.

Fast clear (TFFCA) is a mode that attempts to automatically clear flag bits. We will not use fast clear mode, in order to make our software clearer and to make it friendly when using multiple timer features. TCNT can be reset with a special mode of output compare 7 (TCRE=1). But in this book, we will consider TCNT as a simple 16-bit counter clocked at a continuous rate (we will always set TCRE=0).

The flags in the TFLG1 and TFLG2 registers are cleared by writing a 1 into the specific flag bit we wish to clear. For example, writing a \$FF into TFLG1 will clear all eight flags. The following is a valid method for clearing C3F. That is, this acknowledge sequence clears the C3F flag without affecting the other seven flags in the TFLG1 register.

```
TFLG1 = 0x08;
```

**Checkpoint 6.5:** Write assembly or C code to clear C6F.

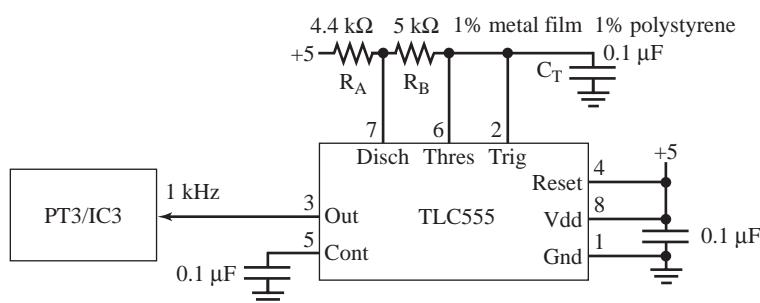
**Common error:** Executing `TFLG1 |= 0x08;` will mistakenly clear all the bits in the TFLG1 register.

**Example 6.1** Design a system that counts the number of rising edges on a digital signal.

**Solution** We solved a similar problem in Example 4.1 using key wake-up. However, in this solution we will use input capture. Any of the eight input-capture pins could have been used, but we selected PT3 as shown in Figure 6.3. The initialization sets the direction register bit 3 to 0, so PT3 is an input. Bit 3 in TI0S is cleared, making timer channel 3 an input capture. Bits 7 and 6 in TCTL4 specify we want to capture on the rising edge of PT3. We arm the input-capture channel by setting bit 3 in TIE. It is good design practice to clear trigger flags in the initialization, so the first interrupt is due to a rising edge on the input occurring after the initialization and not due to events occurring during power up. In this example, we will test the system by attaching the input to an astable multivibrator. The clock frequency of a TLC555 timer is determined by the resistor and capacitor values. The period of a TLC555 timer is  $0.693 \cdot C_T \cdot (R_A + 2R_B)$ . In our circuit,  $R_A$  is  $4.4\text{ k}\Omega$ ,  $R_B$  is  $5\text{ k}\Omega$ , and

**Figure 6.3**

An external signal is connected to the input capture.



$C_T$  is 0.1  $\mu F$ . With these values, the signal on PT3 will be a 1-kHz square wave. By adjusting the resistor values, we can test how fast a signal this system can count.

An input capture interrupt occurs on each rise of the square wave generated by the TLC555. The latency of the system is defined as the time delay between the rise of the input capture signal to the increment of Counter. Assuming there are no other interrupts, and assuming the Main program does not disable interrupts, the delay includes the time to (1) finish the current instruction, (2) process the interrupt, and (3) execute the interrupt handler up to and including changing Counter (Table 6.4).

**Table 6.4**

Components of latency and their values for the assembly language implementation.

Component	9S12
Longest instruction (cycles, $\mu s$ )	13=3.25 $\mu s$
Process the interrupt (cycles, $\mu s$ )	9=2.25 $\mu s$
Execute handler (cycles, $\mu s$ )	11=2.75 $\mu s$
Max latency ( $\mu s$ )	8.25

The latency may be larger if there are other sections of code that execute with the interrupts disabled, like other interrupt handlers. The ritual `Init` sets input capture to interrupt on the rise and initializes the global Counter (Program 6.1).

<pre>;external signal to PT3 Count rmb 2           ;number of events Init sei              ;make atomic     bclr TIOS,#\$08   ;PT3=input capture     bclr DDRT,#\$08   ;PT3 is input     movb #\$80,TSCR1  ;enable TCNT     bclr TCTL4,#\$80   ;EDG3BA =01     bset TCTL4,#\$40   ;on rise of IC3     bset TIE,#\$08     ;Arm C3F     movw #0,Count     ;init global     movb #\$08,TFLG1   ;clear C3F     cli               ;enable     rts C3Han movb #\$08,TFLG1 ;ack C3F [4]     ldx Count         [3]     inx               [1]     stx Count         [3]     rti     org \$FFE8        ;timer channel 3     fdb C3Han</pre>	<pre>// PT3 input = external signal unsigned short Count; // incremented void Init(void){     asm sei          // make atomic     TIOS &amp;=~0x08; // PT3 input capture     DDRT &amp;=~0x08; // PT3 is input     TSCR1 = 0x80; // enable TCNT     TCTL4 = (TCTL4&amp;0x3F) 0x40;     TIE  = 0x08; // Arm IC3, rising     TFLG1 = 0x08; // initially clear     Count = 0;     asm cli } void interrupt 11 C3Han(void){     TFLG1 = 0x08; // acknowledge     Count++; }</pre>
---	---

### Program 6.1

Counting interrupt using input capture.

The interrupt software acknowledges the interrupt, and increments the global variable. The interrupt handlers cannot “poll for 0s and 1s” because none of the bits in the status register is guaranteed to be zero.

**Common Error:** When two software modules both need to set the same configuration register, a poorly written initialization by one software module undoes the initialization performed by the other.

### 6.1.3 Period Measurement

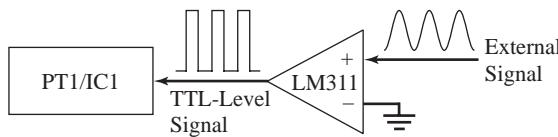
Before one implements a system that measures period, it is appropriate to consider the issues of resolution, precision, and range. The *resolution* of a period measurement is defined as the smallest change in period that can reliably be detected. In the first example, if the period increases by 500 ns, then there will be one more TCNT clock between the first rising edge and the second rising edge. In this situation, the period calculated by TIC1-First will increase by 1; therefore, the period measurement resolution is 500 ns. The resolution is also the significance or the units of the measurement. In this first example, if the calculation of Period results in 1000, then it represents a period of  $1000 \cdot 500 \text{ ns}$ , or 500  $\mu\text{s}$ . In the example presented later in Example 6.9, the period must increase at least 1 ms for the measurement to be able to reliably detect the change. In the example of Section 6.5.1, the period measurement resolution is 1 ms. The *precision* of the period measurement is defined as the number of separate and distinguishable measurements. In the first example, a 16-bit counter is used, so there are about 65,536 different periods that can be measured. We can specify the precision in alternatives (e.g., 65536) or in bits (e.g., 16 bits). The precision of Example 3.3 is 32 bits. The last issue to consider is the *range* of the period measurement, which is defined as the minimum and maximum values that can reliably be measured. We are concerned what happens if the period is too small or too large. A good measurement system should be able to detect overflows and underflows. In addition, we would not like the system to crash, or hang up if the input period is out of range. Similarly, it is desirable if the system can detect when there is no period.

**Example 6.2** Design a system that measures period with a precision of 16 bits and a resolution of 500 ns.

**Solution** In this example, the TTL-level input signal is connected to an input capture pin. Each rising edge will generate an input capture interrupt (Figure 6.4). The period is calculated as the difference in TIC1 latch values from one rising edge to the other (Figure 6.5). For example, if the period is 8192  $\mu\text{s}$ , the IC1 interrupts will be requested every 16,384 cycles, and the difference between TIC1 latch values will be  $16384 = \$4000$ . This subtraction remains valid even if the TCNT overflows and wraps around in between IC1 interrupts. On the other hand, this method will not operate properly if the period is larger than 65,535 cycles, or 32,767  $\mu\text{s}$ .

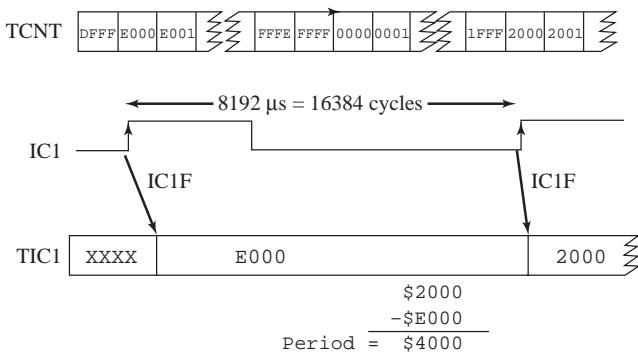
**Figure 6.4**

To measure period we connect the external signal to an input capture.



**Figure 6.5**

Timing example showing counter rollover during 16-bit period measurement.



The resolution is 500 ns because the period must increase by at least this amount before the difference between TIC1 measurements will reliably change. Even though a

16-bit counter is used, the precision is a little less than 16 bits, because the shortest period that can be handled with this interrupt-driven approach is shown in Table 6.5. This factor is determined by counting all the cycles of the `IC1han` and adding the time needed for the microcomputer to process the interrupt. In other words, if the interrupts are requested at a rate faster than the minimum period specified in Table 6.5, then some interrupts will be lost. Also notice that as the period approaches this minimum, a higher and higher percentage of the computer execution is utilized just in the `IC1han` itself. For example, on a 9S12 running at 4 MHz, Table 6.6 shows that if the period is 20  $\mu$ s, then 50% of the time is used executing the ISR.

**Table 6.5**

Calculation of the smallest period that can be handled by the input capture interrupt.

Component	9S12 Assembly	C
Process the interrupt (cycles, $\mu$ s)	$9 = 2.25 \mu\text{s}$	$9 = 2.25 \mu\text{s}$
Execute entire handler (cycles, $\mu$ s)	$31 = 7.75 \mu\text{s}$	$30 = 7.50 \mu\text{s}$
Minimum period (cycles, $\mu$ s)	$40 = 10 \mu\text{s}$	$39 = 9.75 \mu\text{s}$

**Table 6.6**

Calculation of the percentage time in the 9S12 interrupt handler as a function of input period.

Period	Cycles/Interrupt	Percentage Time in Handler
10 $\mu$ s	40	100%
20 $\mu$ s	40	50%
100 $\mu$ s	40	10%
P( $\mu$ s)	40	1000/P (%)

Depending on the complier, C code may execute faster or slower than the assembly, but the general trend (percentage overhead approaches 100% as the period approaches zero) still holds. Metrowerks CodeWarrior creates object code that runs in 30 cycles, one cycle faster than the assembly version. As mentioned earlier, this implementation cannot detect if the period is larger than 32 ms. For example, a signal with period of 40,960  $\mu$ s gives the same result as a signal with a period of 8192  $\mu$ s. This limitation will be corrected in the other two period measurement examples presented later in this chapter.

The period measurement system is presented in Program 6.2. The 16-bit subtraction calculates the number of TCNT clocks between rising edges. Since the ritual does not wait for the first edge, the first period measurement will be incorrect and should be neglected. Because the input capture interrupt has a separate vector, the software does not poll. An interrupt is requested on each rising edge of the input signal.

The code `Period = TIC1-First;` calculates the number of TCNT clocks between rising edges. The first period measurement will be incorrect and should be neglected.

<pre>:external signal to PT1/IC1 Period rmb 2 ;units 500 ns First rmb 2 ;TCNT at first edge Done rmb 1 ;set each rising Init sei ;make atomic bclr TIOS,#\$02 ;PT1=input capture bclr DDRT,#\$02 ;PT1 is input movb #\$80,TSCR1 ;enable TCNT movb #\$01,TSCR2 ;500ns clk bclr TCTL4,#\$08 ;EDG1BA =01 bset TCTL4,#\$04 ;on rise of PT1 movw TCNT,First ;init global clr Done movb #\$02,TFLG1 ;clear C1F</pre>	<pre>// PT1/IC1 input = external signal // rising edge to rising edge // resolution = 500ns // Range = 20 us to 32 ms, // no overflow checking unsigned short Period; // 500 ns units unsigned short First; // TCNT first edge unsigned char Done; // Set each rising void Init(void){     asm sei // make atomic     TIOS &amp;= ~0x02; // PT1 input capture     DDRT &amp;= ~0x02; // PT1 is input     TSCR1 = 0x80; // enable TCNT     TSCR2 = 0x01; // 500ns clock</pre>
--	--

```

bset TIE,#$02    ;Arm C1F
cli             ;enable
rts
IC1Han movb #$02,TFLG1 ;clear C1F [4]
    ldd TC1          [3]
    subd First        [3]
    std Period        [3]
    movw TC1,First     [6]
    movb #$FF,Done      [4]
    rti              [8]

org $FFEC ;timer channel 1
fdb IC1Han

```

	TCTL4 = (TCTL4&0xF3) 0x04; // rising First = TCNT; // first will be wrong Done = 0; // set on subsequent TFLG1 = 0x02; // Clear C1F TIE  = 0x02; // Arm IC1 asm cli } void interrupt 9 TC1handler(void){ Period = TC1-First; // 500ns resolution First = TC1; // Setup for next TFLG1 = 0x02; // ack by clearing C1F Done = 0xFF; }
--	---

**Program 6.2**

16-bit period measurement.

**Checkpoint 6.10:** How would you modify Program 6.2 to implement a 2  $\mu$ s measurement resolution?

We can adjust the three bits (PR2, PR1, and PR0) in the TSCR2 register to specify one of eight possible period measurement resolutions varying from 250 ns to 32  $\mu$ s.

**Example 6.3** Design a system that measures period with a precision of 32 bits and a resolution of 500 ns.

**Solution** In this example, we measure the period of an external signal with a precision of 32 bits and a resolution of 500 ns. The TTL-level signal is connected to an input capture pin (Figure 6.4). Every time the TCNT register overflows from \$FFFF to 0, the TOF flag is set. We can increase the precision of the period measurement as well as implementing period-too-long error checking by counting the number of TOF flag setting events during one period. To implement the period measurement task in the background, we will arm both the input capture and the timer overflow interrupts.

If we use a 16-bit counter, Count, for the number of times the TOF is set during one period, the precision of the period measurement is 32 bits. Let  $T_1$  be the 32-bit time of the first rising edge of the input signal. The high 16 bits of  $T_1$  are 0, and the low 16 bits are the input capture latch value at the time of the first rising edge. Let  $T_2$  be the 32-bit time of the second rising edge of the input signal. The high 16 bits of  $T_2$  are derived from the value of Count, and the low 16 bits are the input capture latch value at the time of the second rising edge. The period is calculated from the 32-bit subtraction:

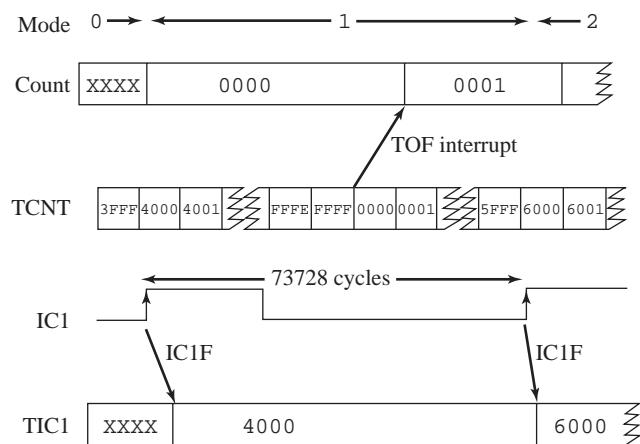
$$\text{Period} = T_2 - T_1$$

Figure 6.6 illustrates a typical 32-bit measurement of period that is  $73728 = \$00012000$  cycles long. On the first rising edge of IC1, the TIC1 value of \$4000 is copied into the global variable First, and Count is cleared. In this situation,  $T_1$  is \$00004000 cycles. Next, the TCNT overflows, causing a TOF interrupt. The TOhandler() increments Count to 1. On the second rising edge of IC1, the time  $T_2$  is measured as Count concatenated with TIC1. In this case,  $T_2 = \$00016000$ . The period is the difference between  $T_2$  and  $T_1$ , that is,  $73728 = \$12000$ . Notice how the C program below handles the borrow from the least significant word into the most significant word during the 32-bit subtraction, using the statement if (TIC1<First) MsPeriod--;

The tricky part about implementing this 32-bit precision measurement is when the TOF and IC1F flags are both set approximately at the same time. At the time of the first rising

**Figure 6.6**

Simple illustration of the 32-bit period measurement.



edge of IC1, if TOF is not set, then the time  $T_1$  is simply TIC1. If TOF is set, then it could have occurred just before IC1, in which case the next TOF interrupt should not increment Count or it could have occurred just after IC1, in which case the next TOF interrupt should increment Count.

Figure 6.7 illustrates the situation when the TOF is set just before the first rising edge of IC1. Even though the TOF flag is set before the IC1F flag, they both could occur during the same instruction. Since IC1F is a higher-priority interrupt than TOF, the `TIC1handler()` will be executed before the `TOFhandler()`. If the TOF flag setting occurred just before the IC1, then the most significant bit of TIC1 will be 0. In this situation, the statement `if (((TIC1&0x8000)==0)&&(TFLG2&0x80))Count--;` will initialize Count to \$FFFF so that the first TOF interrupt will effectively not be counted. In this case,  $T_1$  is \$00000001 cycles and  $T_2$  is \$00011006 cycles, and the period is \$11005, or 69,637 cycles.

**Figure 6.7**

Situation when TOF is set just before the first rising edge of IC1.

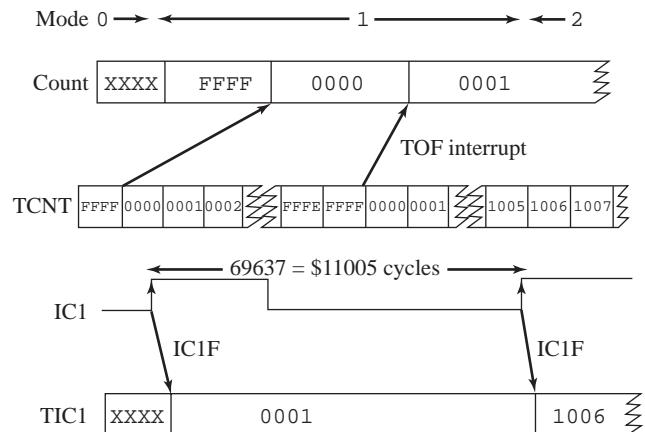
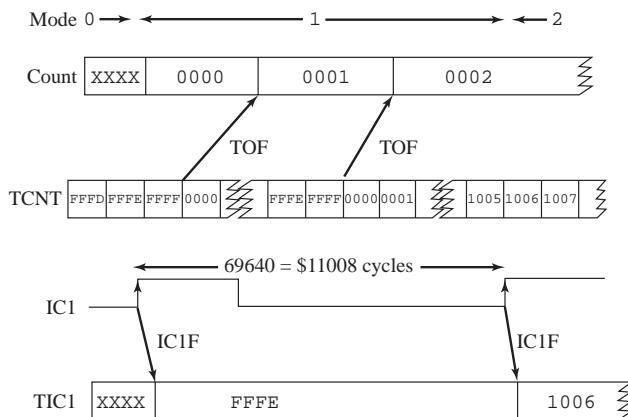


Figure 6.8 illustrates the situation when a TOF occurs just after the IC1F. We assume the TOF flag is set before the line `if (((TIC1&0x8000)==0)&&(TFLG2&0x80))Count--;` is executed. If the TOF flag is set after this line, then we have a case like Figure 6.6, and there is no problem. Since the TOF flag setting occurred just after the IC1, then the most significant bit of TIC1 will be 1. In this situation, the statement `if (((TIC1&0x8000)==0)&&(TFLG2&0x80))Count--;` will leave Count at 0 so that the first TOF interrupt will be properly counted. In this case,  $T_1$  is \$0000FFFE cycles and  $T_2$  is \$000021006 cycles, and the period is \$11008, or 69,640 cycles.

**Figure 6.8**

Situation when TOF is set just after the first rising edge of IC1.



There is a similar problem that exists if the TOF is set just before or just after the second setting of the IC1F. In this case, we will count the TOF by incrementing Count if the TOF occurs just before IC1F. The statement `if(((TIC1&0x8000)==0)&&(TFLG2&0x80)) Count++;` will correct for this situation.

Notice also that if the Count reaches 65535 either in Mode=0 (searching for first rise) or in Mode=1 (search for second rise), the measurement is terminated. This mechanism allows the system to detect a period-too-long. This measurement system does not measure every period, but rather it could measure every other one. This fairly complicated solution was derived from a 6811 assembly language program found in the Freescale 68HC11 Reference Manual, Example 10-3.

Program 6.3 measures period from rising edge to rising edge with a resolution of 500 ns (TCNT clock). The precision is 32 bits. The range varies from 0 to 35 min (TCNT period\*4billion). An input capture interrupt is generated on each rising edge. The signal is connected to IC1 (Figure 6.4). The timer overflow interrupt counts the global variable Count. The globals MsPeriod and LsPeriod compose the 32-bit period measurement. First is the TCNT at the first rising edge. The global Mode means:

- 0 means looking for first edge of IC1
- 1 means looking for second edge
- 2 means measurement is done

### Program 6.3

C language 32-bit period measurement.

```
// MC9S12C32, CodeWarrior C
unsigned short MsPeriod,LsPeriod;
unsigned short First;
unsigned short Count;
unsigned char Mode;
void interrupt 16 TOhandler(void){
    TFLG2 = 0x80; // ack
    Count++;
    if(Count==65535){ // 35 minutes
        MsPeriod=LsPeriod=65535;
        TIE=0x00; TSCR2=0x00; // Disarm
        Mode = 2; // done
    }
}
void interrupt 9 TIC1handler(void){
    if(Mode==0){ // first edge
        First = TC1; Count=0;
        Mode=1;
        if(((TC1&0x8000)==0)
            &&(TFLG2&0x80)) Count--;
    }
}
```

*continued on p. 298*

**Program 6.3**

C language 32-bit period measurement.

*continued from p. 297*

```

else{           // second edge
    if(((TC1&0x8000)==0)
        &&(TFLG2&0x80)) Count++;
    Mode = 2; // measurement done
    MsPeriod = Count;
    LsPeriod = TC1-First;
    if(TC1<First){
        MsPeriod--; // borrow
    }
    TIE=0x00; TSCR2=0x00; // Disarm
}
TFLG1 = 0x02; // ack, clear C1F
}
void Init(void){
    asm sei           // make atomic
    TIOS &=~0x02; // PT1 input capture
    DDRT &=~0x02; // PT1 is input
    TSCR2 = 0x81; // Arm, TOF 30.517Hz
    TSCR1 = 0x80; // enable counter
    TFLG1 = 0x02; // Clear C1F
    TIE |= 0x02; // Arm IC1, C1I=1
    TCTL4 = (TCTL4&0xF3)|0x04; // rising
    TFLG2 = 0x80; // Clear TOF
    Mode = 0; // searching for first
    asm cli
}

```

### 6.1.4 Pulse-Width Measurement

The basic idea of pulse-width measurement is to cause an input capture event on both the rising and falling edges of an input signal. The difference between these two times will be the pulse width. Just like period measurement, the resolution is determined by the rate at which TCNT is incremented.

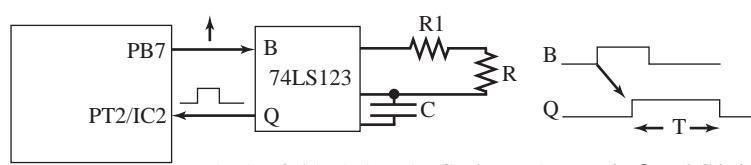
**Example 6.4** Design a system to measure resistance and use it to interface a joystick.

**Solution** The objective is to use *input capture pulse-width measurement* to measure resistance (Figure 6.9). This basic approach is employed by most joystick interfaces. The resistance measurement range is  $0 \leq R \leq 1 \text{ M}\Omega$ . The desired resolution is  $1 \text{ k}\Omega$ . We will use busy-wait synchronization.

Figure 6.9 shows the hardware interface between the unknown resistance  $R$  and the input capture pin of the microcomputer. A rising edge on PB7 causes a monostable positive logic pulse on the “Q” pin of the 74LS123. We choose  $R_1$  and  $C$  so that the resistance resolution maps into a pulse-width measurement resolution of 500 ns, and the resistance range  $0 \leq R \leq 1 \text{ M}\Omega$  maps into  $500 \leq T \leq 1000 \mu\text{s}$ . The following equation describes

**Figure 6.9**

To measure resistance using pulse width we connect the external signal to an input capture.



$$T(\text{sec}) = 0.45 \cdot (R_1 + R) \cdot C \text{ where } R_1, R \text{ are in } \Omega \text{ and } C \text{ is in F}$$

the pulse width generated by the 74LS123 monostable as a function of the resistances and capacitance.

$$T = 0.45 \cdot (R + R_1) \cdot C$$

For a linear system, with  $x$  as input and  $y$  as output, we can use calculus to relate the measurement resolution of the input and output.

$$\Delta y = \frac{y}{x} \Delta x$$

Therefore, the relationship between the pulse-width measurement resolution  $\Delta T$  and the resulting resistance measurement resolution is determined by the value of the capacitor.

$$\Delta T = 0.45 \Delta R \cdot C$$

To make a  $\Delta T$  of 500 ns correspond to a  $\Delta R$  of 1 k $\Omega$ , we choose

$$C = \Delta T / (0.45 \cdot \Delta R) = 500 \text{ ns} / 0.45 \text{ k}\Omega = 1111 \text{ pF}$$

To study the range of pulse widths, we look at  $R=0$ ,

$$T = 0.45 \cdot R_1 \cdot C$$

so

$$R_1 = T / (0.45 \cdot C) = 500 \mu\text{s} / (0.45 \cdot 1111 \text{ pF}) = 1 \text{ M}\Omega$$

As a check, notice at  $R=1 \text{ M}\Omega$ ,  $T = 0.45 \cdot (2 \text{ M}\Omega) \cdot 1111 \text{ pF} = 1 \text{ ms}$ .

The measurement subroutine, including ritual, returns in RegD the current resistance  $R$  in k $\Omega$ . For example, if the resistance  $R$  is 123 k $\Omega$ , then RegD will be 123. We will not worry about resistances  $R$  greater than 1 M $\Omega$  or if  $R$  is disconnected.

The Init function is the ritual that will configure the system in Program 6.4. Next, the Meas subroutine used to perform the input capture measurement. Again the function returns resistance in units of k $\Omega$ .

<pre>; B=PB7, Q=PT2/IC2 Init    bclr TIOS,#\$04 ;PT2=input capture         movb #\$80,TSCR1 ;enable         movb #\$01,TSCR2 ;500 ns clock         clr TIE          ;gadfly, C2I=0         bset DDRB,#\$80         rts ; return Reg D as R in kohm Meas    movb #\$10,TCTL4 ;Rising edge         movb #\$04,TFLG1 ;C2F=0         bclr PORTB,#\$80 ;PB7=0         bset PORTB,#\$80 ;PB7=1 First   brclr TFLG1,#\$04,First ;Wait for first rising edge         ldy TC2 ;TCNT at rising         movb #\$04,TFLG1 ;C2F=0         movb #\$20,TCTL4 ;Falling edge         pshy      ;Save on stack Second  brclr TFLG1,#\$04,Second ;Wait for next falling edge         ldd TC2 ;TCNT at falling         subd 2,SP+ ;RegD=pulse width 1000 to 2000 cyc         subd #1000 ;0&lt;=R&lt;=1000 kohm         rts </pre>	<pre>// return resistance in kohms // 0 to 1000 kohm unsigned short Measure(void) {     unsigned short Rising;     TCTL4 = (TCTL4&amp;0xCF) 0x10; // Rising     TFLG1 = 0x04; // clear C2F     PORTB &amp;=~0x80;     PORTB  = 0x80; // rising edge on PB7     while((TFLG1&amp;0x04)==0){}; // wait for rise     Rising = TC2; // TCNT at rising edge     TFLG1 = 0x04; // clear C2F     TCTL4 = (TCTL4&amp;0xCF) 0x20; // Falling     while((TFLG1&amp;0x04)==0){}; // wait for fall     return(TC2-Rising-1000); } void Init(void){     DDRB  = 0x80; // PB7 is output     TIOS &amp;=~0x04; // clear bit 2     DDRT &amp;=~0x04; // PT2 is input capture     TSCR1 = 0x80; // enable     TSCR2 = 0x01; // 500 ns clock     TIE = 0x00; // no interrupts }</pre>
---	---

## Program 6.4

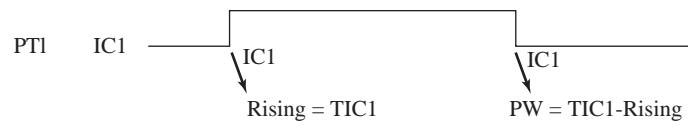
Measuring resistance using pulse-width measurement.

**Example 6.5** Design a system to measure pulse width using interrupts, with a precision of 16 bits and a resolution of 500 ns.

**Solution** In this example, the digital logic signal is connected to an input capture (Figure 6.4). Both the rising and falling edges will generate an input capture interrupt. The pulse width is calculated as the difference in TIC1 latch values from a rising edge to the next falling edge (Figure 6.10). In this example the background thread sets the global variable `Rising` on the rising edge and calculates the pulse width `PW` on the falling edge. The interrupt handler simply reads the current value on the input capture pin to determine if this interrupt is a rising or falling edge. If the first interrupt is a falling edge, then the first pulse width measured will be inaccurate.

**Figure 6.10**

Pulse width is the time between the rising and falling edges.



The pulse-width measurement is performed from rising edge to falling edge. The resolution is 500 ns (determined by TCNT clock). The range is about 20  $\mu$ s to 32 ms, with no overflow checking. The low end of the range is determined by the software overhead to process the interrupt. IC1 interrupts occur on both the rising and falling edges. The global `PW` contains the most recent measurement. `Rising` is a private<sup>1</sup> global variable containing the TCNT value at the rising edge.

`Done` is set at the falling edge of IC1, signifying a new measurement is available. The main program reads `Done` and `PW` to process the measurements. The main program can clear `Done` after it has processed the measurement. The interrupt handler could (but doesn't in this simple example) check `Done` and trigger an error if a new measurement is complete, but the previous measurement has not been processed yet. Because the `Init()` sets `Rising` equal to the current time (and not the time of a rising edge), if the first interrupt is falling edge, then the first measurement will be incorrect. If the first interrupt after the `Init()` is executed is a rising edge, then the first measurement will be correct (Program 6.5).

### Program 6.5

C language pulse-width measurement.

```
// MC9S12C32, CodeWarrior C
unsigned short PW;          // units of 500 ns
unsigned short Rising;      // TCNT at rising
unsigned char Done;         // Set each falling
void interrupt 9 TC1handler(void){
    if(PTT&0x02){        // PTT=1 if rising
        Rising = TC1;    // Setup for next
    }
    else{
        PW = TC1-Rising; // measurement
        Done = 0xFF;
    }
    TFLG1 = 0x02;        // ack, clear C1F
}
```

<sup>1</sup>A global variable only accessed by one module.

```

void Init(void) {
    asm sei      // make atomic
    TIOS &= ~0x02; // clear bit 1
    DDRT &= ~0x02; // PT1 is input capture
    TSCR1 = 0x80; // enable
    TSCR2 = 0x01; // 500 ns clock
    TCTL4 |= 0x0C; // Both edges IC1
    TIE |= 0x02; // arm IC1
    TFLG1 = 0x02; // clear C1F
    Rising = TCNT;
    Done = 0;
    asm cli
}

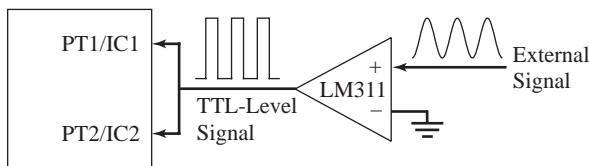
```

**Checkpoint 6.7:** What is the relationship between the resolution and the maximum pulse width that can be measured?

**Example 6.6** Design a system to measure pulse width with a minimum period of 500 ns, a precision of 16 bits, and a resolution of 500 ns.

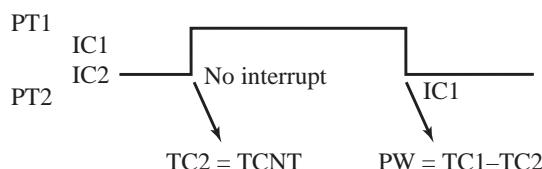
**Solution** One of the limitations of the previous measurement technique is that the lower bound on the range is determined by the software overhead to process the input capture interrupt. If the pulse-width is below that minimum, the error goes undetected. To solve this problem, the digital logic signal is connected to both IC1 and IC2 (Figure 6.11). The rising-edge time will be measured by IC2 without the need of an interrupt, and the falling-edge interrupts will be handled by IC1.

**Figure 6.11**  
To measure pulse we could connect the external signal to two input captures.



The pulse width is calculated as the difference in  $TC1 - TC2$  latch values. In this example the IC1 interrupt handler simply sets the global variable `PW` at the time of the falling edge. Because no software is required to process the IC2 measurement, there is no minimum pulse width. On the other hand, software processing is required to handle the IC1 signal, so there is a minimum period (e.g., there must be more than 20  $\mu$ s from one falling edge to the next falling edge). Again, the first measurement may or may not be accurate (Figure 6.12).

**Figure 6.12**  
The rising edge is measured with IC2, and the falling edge is measured with IC1.



The pulse-width measurement is performed from rising edge to falling edge. The resolution is 500 ns (determined by TCNT). The range is about 500 ns to 32 ms, with no overflow checking. IC1 interrupts only occur on the falling edges. The global PW contains the most recent measurement. Done is set at the falling edge of IC1, signifying a new measurement is available. If the first edge after the `Init()` is executed is a falling edge, then the first measurement will be incorrect (because TC2 is incorrect). If the first edge after the `Init()` is executed is a rising edge, then the first measurement will be correct. Compared to the last example, notice how little software overhead is required to perform these measurements (Program 6.6).

Notice how the C code in Program 6.6 sets four bits of TCTL4 without modifying the other six bits. We call this a “friendly” ritual because it does not undo other initializations. On the other hand, the rituals all have to get together and agree on a common value for the TSCR2.

### Program 6.6

C language pulse-width measurement using two input captures.

```
// MC9S12C32, CodeWarrior C
unsigned short PW; // units of 500 ns
unsigned char Done; // Set each falling

void interrupt 9 TIC1handler(void){
    TFLG1 = 0x02; // ack C1F
    PW = TC1-TC2; // from rise to fall
    Done = 0xFF;
}

void Init(void) {
    asm sei // make atomic
    TIOS &= ~0x06; // clear bits 2,1
    DDRT &= ~0x06; // PT2,PT1 input captures
    TSCR1 = 0x80; // enable
    TSCR2 = 0x01; // 500 ns clock
    TCTL4 = (TCTL4&0xCF)|0x10; // IC2 Rise
    TCTL4 = (TCTL4&0xF3)|0x08; // IC1 Fall
    Done = 0; // set on the falling edge
    TIE |= 0x02; // arm IC1, not IC2
    TFLG1 = 0x02; // clear C1F
    asm cli
}
```

## 6.2 Output Compare

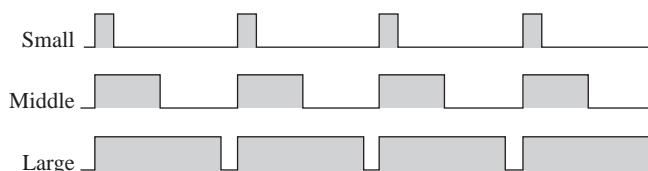
### 6.2.1 General Concepts

As with our introduction to input capture, we begin our discussion of output compare with some general comments. Output compare will be used to create squarewaves, generate pulses, implement time delays, and execute periodic interrupts. A common output technique used in computer-based control systems is a variable duty-cycle actuator. Another name for this is pulse-width modulation (PWM). In Figure 6.13, the shaded areas represent times when power is applied. The computer generates a squarewave with variable duty-cycle using the output compare interface. To apply a large amount of power to the control system, it issues a squarewave that is mostly high. To apply a small amount of power, it outputs a squarewave that is mostly low.

We will also use output compare together with input capture to measure frequency. Output compare and input capture can also be combined to measure period and frequency over a wide range of ranges and resolutions. The same 16-bit TCNT, incremented at a fixed

**Figure 6.13**

The output power is controlled by varying the duty cycle of a squarewave.

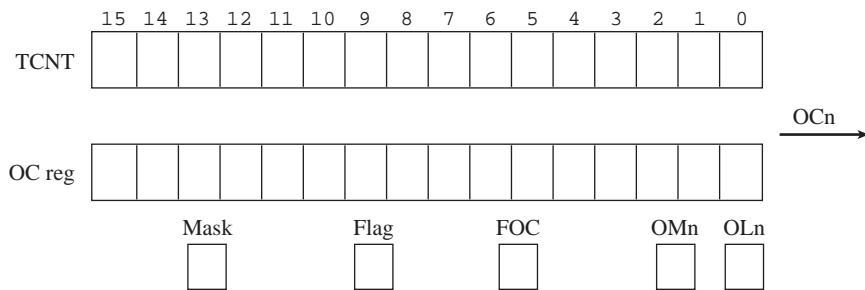


rate, is used for input capture and output compare. As shown in Figure 6.14, each output compare module has

- An external output pin, OC<sub>n</sub>
- A flag bit
- A force compare control bit, FOC<sub>n</sub>
- Two control bits, OM<sub>n</sub> OL<sub>n</sub>
- An interrupt mask bit (arm bit)
- A 16-bit output compare register

**Figure 6.14**

The basic components of output compare.



The various microcomputers from Freescale have from zero to fifteen output-compare modules. However, most 9S12 microcontrollers have eight channels, which can be configured either as input capture or output compare.

The output compare pin is an output of the computer; hence it will be used to control an external device. An output compare event occurs, setting the flag bit, when either

1. The 16-bit TCNT matches the 16-bit OC register, or
2. The software writes a one to the FOC bit.

The OM<sub>n</sub>, OL<sub>n</sub> bits specify what effect the output compare event will have on the output pin. Two or three actions result from an output compare event: 1) the OC<sub>n</sub> output bit changes, 2) the output compare flag is set, and 3) an interrupt is requested if the mask bit is 1. Just like the input capture, the output compare flag is cleared by writing a one to it. The module is armed when the mask bit is 1. One simple application of output compare is to create a fixed time delay. Let `delay` be the number of cycles you wish to wait. The steps to create the delay are as follows:

1. Read the current 16-bit TCNT
2. Calculate `TCNT+delay`
3. Set the 16-bit output compare register to `TCNT+delay`
4. Clear the output compare flag,
5. Wait for the output compare flag to be set

If we perform the addition `TCNT+delay` with 16-bit integer math (e.g., `addd` instruction), then this approach will function properly even if the counter rolls over from \$FFFF to 0. For example, assume the current TCNT is \$F000, and we wish to delay 8192 (\$2000) cycles. The 16-bit addition \$F000+\$2000 will result in a sum of \$1000. If we put the \$1000

into the output compare register, then it will take the correct number (i.e., 8192 cycles), for the TCNT to go from \$F000 to \$1000. The time for the software to execute Steps 1–4 will determine the minimum delay that can be created with this approach. For example, if the value of `delay` is so small that the TCNT hits `TCNT+delay` before the flag is cleared in Step 4, then the delay will be a  $65536 + \text{delay}$  delay. For obvious reasons, the maximum delay with this simple method is 65536 cycles.

One of the output compare modules, PT7/OC7 on most 9S12 microcontrollers, can be configured such that an output compare event on it will cause changes on some or all of the other output compare pins. This coupled behavior can be used to create synchronized signals. For example, we can create pulses that start together or end together.

**Checkpoint 6.8:** When does an output compare event occur?

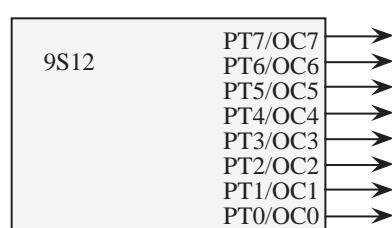
**Checkpoint 6.9:** What happens during an output compare event?

## 6.2.2 Output Compare Details

In the following subsections we overview the specific output compare functions on particular Freescale microcomputers. This section is intended to supplement rather than replace the Freescale manuals. When designing systems with output compare, please refer to the reference manual of your specific Freescale microcomputer.

Because the 9S12 input capture and output compare modules share I/O pins (Figure 6.15) and many registers, we have already introduced much of the output compare components. Table 6.7 summarizes the 9S12 registers used for output compare. The entries shown in bold will be used in this section. The TEN bit in the TSCR1 register must be enabled. TCNT is a 16-bit unsigned counter that is incremented at a rate determined by three bits (PR2, PR1, and PR0) in the TSCR2 register. The fundamental approach to output compare involves connecting one of the eight output compare pins on Port T to an external TTL-level device. The pin is selected as output compare by placing a 1 in the corresponding bit of the TIOS register. With input capture, we must set the direction bit in DDRT to zero, but selecting output compare in the TIOS register automatically makes the bit an output. An output compare event occurs when one of the Output Compare Registers (TCn) matches the TCNT register. This event will set the corresponding output compare flag (CnF) in the TFLG1 register. If the output pin is activated, as specified in the TCTL1 and TCTL2 registers, then this output compare event can also modify the output value (set to one, clear to zero, or toggle.)

**Figure 6.15**  
Available signals for  
output compare on the  
9S12.



We can arm or disarm the individual output compare interrupts by initializing the TIE register. Our software can determine whether an output compare event has occurred by reading the TFLG1 register. Recall the flags in the TFLG1 and TFLG2 registers are cleared by writing a one into the specific flag bit we wish to clear. For example, writing a \$FF into TFLG1 will clear all eight flags. The following is a valid method for clearing C3F. That is, these acknowledge sequences clear the C3F flag without affecting the other seven flags in the TFLG1 register.

`TFLG1=0x08 ;`

**Checkpoint 6.10:** Write assembly or C code to clear the C4F flag.

**Checkpoint 6.11:** Explain why `TFLG1 |= 0x20 ;` is unfriendly.

Address	msb															lsb	Name
\$0044	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TCNT
\$0050	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC0
\$0052	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC1
\$0054	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC2
\$0056	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC3
\$0058	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC4
\$005A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC5
\$005C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC6
\$005E	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC7

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
\$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
\$0046	TEN	TSWAI	TSBCK	TFFCA	0	0	0	0	TSCR1
\$004D	TOI	0	0	0	TCRE	PR2	PR1	PR0	TSCR2
\$0040	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0	TIOS
\$004C	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I	TIE
\$004E	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F	TFLG1
\$004F	TOF	0	0	0	0	0	0	0	TFLG2
\$0048	OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4	TCTL1
\$0049	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0	TCTL2
\$0041	FOC7	FOC6	FOC5	FOC4	FOC3	FOC2	FOC1	FOC0	CFORC
\$0042	OC7M7	OC7M6	OC7M5	OC7M4	OC7M3	OC7M2	OC7M1	OC7M0	OC7M
\$0043	OC7D7	OC7D6	OC7D5	OC7D4	OC7D3	OC7D2	OC7D1	OC7D0	OC7D
\$0047	TOV7	TOV6	TOV5	TOV4	TOV3	TOV2	TOV1	TOV0	TTOV

**Table 6.7**

9S12 registers used for output compare.

The output compare event (when the 16-bit output compare register equals the 16-bit TCNT) can be configured to affect an output pin. The TCTL1 and TCTL2 registers determine what effect each of the four possible output compare events will have (none, toggle, clear, or set) on the output pin, as shown in Table 6.8.

**Table 6.8**

Two bits determine the action caused by 9S12 output compare event.

OMn	OLn	Effect of When TOCn = TCNT
0	0	does not affect OCn
0	1	Toggle OCn
1	0	Clear OCn=0
1	1	Set OCn=1

There is one output compare, OC7, which operates differently. On a successful OC7 event ( $TC7=TCNT$ ), the 9S12 can be programmed to set or clear any of the other output compare pins. The OC7M register selects which pin(s) will be affected by the OC7 event. Clear the corresponding bit(s) of the OC7M register to zero to disconnect OC7 from the output pin(s). Conversely, set the corresponding bit(s) of the OC7M register to one to attach the OC7 event to the output pin(s). For every bit in the OC7M register that is one, the corresponding bit in the OC7D register will specify the resulting value of the output pin(s) after an OC7 event.

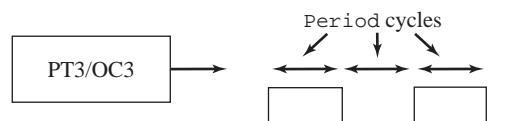
**Observation:** The phase-lock-loop (PLL) on the 9S12 will affect the TCNT period.

**Checkpoint 6.12:** List the steps to initialize PT0 as an armed output compare with toggled output.

**Example 6.7** Design a system that generates a 50% duty cycle square wave.

**Solution** This example generates a 50% duty cycle square wave using output compare. The output is high for *Period* cycles, then low for *Period* cycles. Output compare interrupts will be requested at a rate twice as fast as the resulting square-wave frequency. One interrupt is required for the rising edge on the output compare pin and another for the falling edge. Toggle mode is used to create the 50% duty cycle square wave (Figure 6.16). In this mode, the output compare pin is toggled whenever the output compare latch matches TCNT.

**Figure 6.16**  
Square-wave generation  
using output compare.



The output compare interrupt handler simply acknowledges the interrupt and calculates the time for the next signal transition. This implementation will create an exact square wave independent of software execution delays as long as the interrupt is serviced within *Period* cycles. If the interrupt latency (i.e., the time between the setting of the output compare flag and the running of the ISR) is more than *Period* cycles, then the next edge will not occur for another 65,536 cycles.

Program 6.7 contains the ritual and the interrupt handlers.

<pre> Period rmb 2      ;units usec Init   sei         ;make atomic         bset TIOS,#\$08  ;OC3         bset DDRT,#\$08  ;PT3 output         movb #\$80,TSCR1 ;enable         movb #\$01,TSCR2 ;500 ns clock         bset TIE,#\$08   ;Arm OC3         bset TCTL2,#\$40 ;OL3=1, toggle         bclr TCTL2,#\$80 ;OM3=0         movb #\$08,TFLG1 ;clear C3F         ldd  TCNT      ;current time         addd #50       ;first in 25us         std   TC3         cli           ;enable         rts OC3Han movb #\$08,TFLG1 ;Ack      [4]         ldd  TC3          [3]         addd Period ;next [3]         std   TC3          [2]         rti             [8]         org  \$FFE8         fdb   OC3Han </pre>	<pre> unsigned short Period; // in usec // Number of cycles Low and High void Init(void) {     asm sei           // make atomic     TIOS  = 0x08; // enable OC3     DDRT  = 0x08; // PT3 is output     TSCR1 = 0x80; // enable     TSCR2 = 0x01; // 500 ns clock     TCTL2 = (TCTL2&amp;0x3F) 0x40; // toggle     TIE  = 0x08; // Arm output compare 3     TFLG1 = 0x08; // Initially clear C3F     TC3 = TCNT+50; // First one in 25 us     asm cli } void interrupt 11 TC3handler(void){     TFLG1 = 0x08; // ack C3F     TC3 = TC3+Period; // calculate Next } </pre>
--	--

**Program 6.7**

Software to generate a squarewave using output compare.

To determine the fastest square wave that can be created, we count the time it takes to process an interrupt. The fastest square wave will occupy 100% of the computer execution and request interrupts at twice the frequency (Table 6.9).

**Table 6.9**

Total time in the handler calculated for different implementations.

Component	Assembly	C
Process the interrupt (cycles, $\mu$ s)	9 = 2.25 $\mu$ s	9 = 2.25 $\mu$ s
Execute entire handler (cycles, $\mu$ s)	20 = 5 $\mu$ s	19 = 4.75
Total time ( $\mu$ s)	7.25	7

The following is the assembly listing output of the `TC3handler()` generated by Metrowerks CodeWarrior.

```
Function: TC3handler
0000 c608      [ 1 ]    LDAB   #8          ; TFLG1 = 0x08;
0002 5b00      [ 2 ]    STAB   _TFLG1
0004 dc00      [ 3 ]    LDD    _TC3         ; TC3 = TC3+Period;
0006 f30000    [ 3 ]    ADDD   Period
0009 5c00      [ 2 ]    STD    _TC3
000b 0b        [ 8 ]    RTI
```

The total time to process this C language OC3 interrupt includes (1) 9 cycles for the 9S12 to process the interrupt and (2) 19 cycles to complete the `TC3handler()`. Therefore, each OC3 interrupt requires 28 cycles to process. In this case, we did not include the time to finish the current instruction because we will calculate the overhead required to generate this square wave (Table 6.10). Since an OC3 interrupt is requested every `Period` cycle and each one takes exactly 28 cycles to complete, the overhead required to produce the square wave can be calculated as a percentage of the total available 9S12 execution cycles.

**Table 6.10**

Percentage overhead calculated for a 9S12 C language implementation versus frequency running at 4MHz.

Frequency	Period	Interrupt every (cycles)	Time to process (cycles)	Overhead (%)
100 Hz	10 ms	40,000	28	0.28
1 kHz	1 ms	4000	28	2.8
5 kHz	200 $\mu$ s	800	28	14
1/P	P ( $\mu$ s)	P	28	2800/P

Just like the previous example of the real-time clock, as long as no other software executes with the interrupts disabled for longer than `Period` cycles, this solution will create the perfect square wave. Remember the toggle event (i.e., when `PT3` changes) occurs automatically in hardware at the time when `TC3=TCNT`, and not during the software execution of the `TC3handler()`.

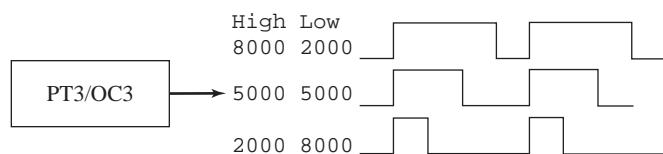
The slowest square wave that can be generated by this implementation has a period of 65,535  $\mu$ s—that is, about 15.3 Hz. On the MC9S12C32 you can use a 32  $\mu$ s `TCNT` to create a squarewave with a 4-second period. Less accurate square waves can be made using an interrupt real-time clock and a software counter. Using the real-time clock, we could interrupt at a fixed interval `T` and decrement a software counter. When that counter hits 0, we could toggle an output bit and reset the counter to `N`. The period of the resulting square wave would be  $2 \cdot N \cdot T$ . These functions could be added to the implementations presented earlier in Programs 6.7.

### 6.2.3 Pulse-Width Modulation

This example generates a variable duty cycle square wave using output compare (Figure 6.17). Pulse-width modulation is an effective and thus popular mechanism for embedded microcomputers to control external devices. The output is 1 for `High` cycles, then 0 for `Low` cycles. Output compare interrupts will again be requested at a rate twice as fast as the

**Figure 6.17**

Pulse-width modulation using output compare.



resulting square-wave frequency. One interrupt is required for the rising edge and another for the falling edge. Toggle mode is used to create the variable duty cycle square wave. In the examples below, High plus Low will always equal 10,000, so in each case the square-wave period will be 10,000 cycles, or 5 ms. By adjusting the ratio of

$$\text{Duty cycle} = \frac{\text{High}}{\text{High} + \text{Low}}$$

the software can control the duty cycle. This implementation cannot generate waves close to 0 or 100% duty cycle. The upper and lower limits can be calculated by counting the cycles required to process the output compare interrupt like in the previous example. If  $T$  is the maximum number of cycles to process the output compare interrupt, then both High and Low must be greater than  $T$ .

The ritual and the interrupt handler are in Program 6.8 (there will be an interrupt for both the rise and fall).

<pre> High    rmb  2 ;number of cycles high Low     rmb  2 ;number of cycles low Init   sei           ;make atomic        bset TIOS,#\$08 ;OC3        bset DDRT,#\$08 ;PT3 output        movb #\$80,TSCR1 ;enable        movb #\$01,TSCR2 ;500 ns clock        bset TIE,#\$08   ;Arm OC3        bset TCTL2,#\$40 ;OL3=1, toggle        bclr TCTL2,#\$80 ;OM3=0        movb #\$08,TFLG1 ;clear C3F        ldd  TCNT      ;current time        addd #50        ;first in 25us        std   TC3        cli            ;enable       rts OC3Han movb #\$08,TFLG1 ;Ack      [4][4]         ldaa PTT      ;rise/fall? [3][3]         bita #\$08     ;PT3      [1][1]         beq zero      [1][3] one    ldd  TC3          [3]         addd High     ;now PT3 is 1 [3]         std   TC3          [2]         bra  done      [3] zero   ldd  TC3          [3]         addd Low      ;now PT3 is 0 [3]         std   TC3          [2] done   rti             [8][8] org   \$FFE8 fdb   OC3Han </pre>	<pre> unsigned short High; // Cycles High unsigned short Low; // Cycles Low // Period is High+Low cycles void Init(void){     asm sei           // make atomic     TIOS  = 0x08; // enable OC3     DDRT  = 0x08; // PT3 is output     TSCR1 = 0x80; // enable     TSCR2 = 0x01; // 500 ns clock     TIE  = 0x08; // Arm output compare 3     TFLG1 = 0x08; // Initially clear C3F     TCTL2 = (TCTL2&amp;0x3F) 0x40; // toggle     TC3 = TCNT+50; // first right away     asm cli }  void interrupt 11 TC3handler(void){     TFLG1 = 0x08; // ack C3F     if(PTT&amp;0x08){ // PT3 is now high         TC3 = TC3+High; // 1 for High cyc     }     else{ // PT3 is now low         TC3 = TC3+Low; // 0 for Low cycles     } } </pre>
---	--

## Program 6.8

Software to generate a pulse-width modulated square wave using output compare.

To determine the minimum and maximum pulse widths that can be created we count the time it takes to process an interrupt. The cycle times shown in brackets, e.g., [2], in Program 6.8 are executed for each interrupt. The two columns in Table 6.11 represent the two possible branch patterns of the interrupt execution. This table presents the **T** parameter discussed previously.

**Table 6.11**

Total time in the handler calculated for different implementations.

Component	Assembly	C
Process the interrupt (cycles)	9	9
Execute entire handler (cycles)	27–28	24–27
Total time <b>T</b> ( $\mu$ s)	9.25 $\mu$ s	9 $\mu$ s

To determine the **T** parameter for the C implementation, you need to observe the assembly listing from the compiler, count cycles, and perform a calculation as shown in Table 6.11.

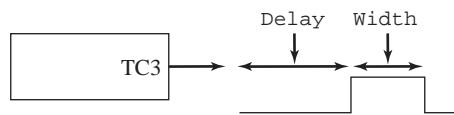
**Checkpoint 6.13:** How do you change Program 6.8 so that the period is 20 ms instead of 5 ms?

**Observation:** The 9S12 ISR produced by Metrowerks executes in 24 to 27 cycles, which is actually faster than the assembly program written in Program 6.8.

## 6.2.4 Delayed Pulse Generation

One application of the coupled output compare mechanism is a delayed output pulse (Figure 6.18). One pulse will be generated each time the function `Pulse()` is called. We will supply two parameters to this pulse generation function, `Delay` and `Width`. The first parameter is the delay (in cycles), and the second is the width of the pulse. Delayed pulse generation could have been created with a single output compare module, but the use of two output compare modules allows the `Width` to be as short as 1 cycle. We can set TC7 to the time we want the pulse to go high and set TC3 to the time we want the pulse to go low. The TC3 interrupt is disarmed in the interrupt handler so that the output pulse occurs only once and is not repeated every 65,536 counts of TCNT. Although the `Width` can be as short as 1 cycle, the `Delay` must be big enough so that TCNT does not reach the TC3 value before the C3F flag is cleared (i.e., the execution time of this function must be less than the `Delay` time).

**Figure 6.18**  
Delayed pulse generation using output compare.



We set both OC7M3 and OC7D3 to 1. When TCNT equals TC7, PT3/OC3 will go high. We set OM3 to 1 and OL3 to 0. When TCNT=TC3, PT3/OC3 will go low (Program 6.9).

**Observation:** We acknowledge an interrupt (clear its flag) within the interrupt handler when we are interested in subsequent interrupts, and we disarm an interrupt (clear its mask) when we are not interested in any more interrupts from this source.

**Program 6.9**

C language delayed pulse output using output compare.

```
// MC9S12C32
void Pulse(unsigned short Delay,
           unsigned short Width){
    asm sei           // make atomic
    TIOS |= 0x08;    // enable OC3
    DDRT |= 0x08;   // PT3 is output
    TSCR1 = 0x80;   // enable
    TSCR2 = 0x01;   // 500 ns clock
    TC7 = TCNT+Delay;
    TC3 = TC7+Width;
    OC7M = 0x08;    // connect OC7 to PT3
    OC7D = 0x08;    // PT3=1 when TC7=TCNT
    TCTL2=(TCTL2&0x3F)|0x80;
    // PT3=0 when TC3=TCNT
    TFLG1 = 0x08;   // Clear C3F
    TIE |= 0x08;    // Arm C3F
    asm cli
}
void interrupt 11 TC3handler(void){
    OC7M = 0;      // disconnect OC7 from PT3
    OC7D = 0;
    TCTL2 &=~0xC0; // disable OC3
    TIE &= ~0x08;  // disarm C3F
}
```

## 6.3 Frequency Measurement

### 6.3.1 Frequency Measurement Concepts

The direct measurement of frequency involves counting input pulses for a fixed amount of time. The basic idea is to use Input Capture to count pulses and use Output Compare to create the fixed time interval. For example, we could initialize Input Capture to interrupt on every rising edge of our input signal. During the Input Capture handler, we could increment a Counter. At the beginning of our fixed time interval, the Counter is initialized to zero, and at the end of the interval, we can calculate frequency:

$$f = \frac{\text{counter}}{\text{fixed time}}$$

The frequency resolution,  $\Delta f$ , is defined to be the smallest change in frequency that can be reliably measured by the system. For the system to detect a change, the frequency must increase (or decrease) enough so that there is one more (or one less) pulse during the fixed time interval. Therefore, the frequency resolution is

$$\Delta f = \frac{1}{\text{fixed time}}$$

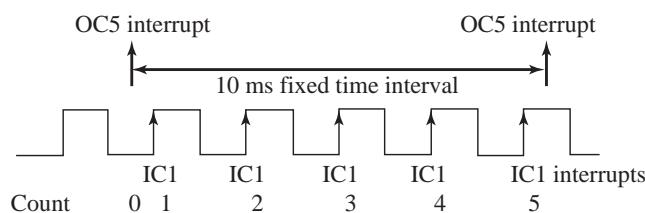
This frequency resolution also specifies the units of the measurement.

**Example 6.8** Design a system that measures frequency with a resolution of 100 Hz.

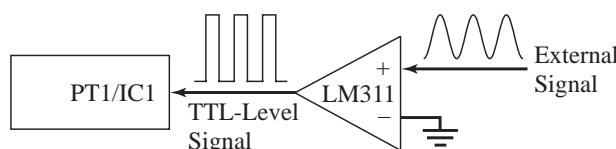
**Solution** If we count pulses in a 10-ms time interval, then the number of pulses represents the signal frequency with units 1/10 ms, or 100 Hz. For example, if there are 5 pulses during the 10-ms interval, then the frequency is 500 Hz. For this system, the measurement resolution is 100 Hz, so the frequency would have to increase to 600 Hz (or decrease to 400 Hz) for the change to be reliably detected (Figure 6.19).

**Figure 6.19**

Basic timing involved in frequency measurement using both input capture and output compare.

**Figure 6.20**

Frequency measurement using both input capture and output compare.



In C, the frequency measurement software is as follows. The frequency measurement counts the number of rising edges in a 10-ms interval. The measurement resolution is 100 Hz (determined by the 10-ms interval). An IC1 interrupt occurs on each rising edge, and an OC5 interrupt occurs every 10 ms. The digital logic signal is connected to PT1, and the foreground/background threads communicate via two shared globals. The background thread will update `Freq` with a new measurement and set `Done`. When `Done` is set, the foreground thread will read the global `Freq` and clear `Done`.

```
unsigned short Freq; /* Frequency with units of 100 Hz */
unsigned char Done; /* Set each measurement, every 10 ms */
```

There is a private global (only accessed by the background threads and not the foreground).

```
unsigned short Count; /* Number of rising edges */
```

If we assume the fastest period is 50 µs, then the largest frequency will be 20 kHz. Therefore, the maximum values of `Count` and `Freq` will be 200, so 8-bit variables could have been used. A frequency of 0 will result in no input capture interrupts, and the system will properly report the frequency of 0. The implementation is shown in C in Program 6.10.

### Program 6.10

C language frequency measurement.

```
// MC9S12C32, CodeWarrior C
void interrupt 9 TC1handler(void){
    Count++; // number of rising edges
    TFLG1 = 0x02; // ack, clear C1F
}
#define Rate 20000 // 10 ms
void interrupt 13 TC5handler(void){
    TFLG1= 0x20; // Acknowledge
    TC5 = TC5+Rate; // every 10 ms
    Freq = Count; // 100 Hz units
    Done = 0xff;
    Count = 0; // Setup for next
}
```

*continued on p. 312*

**Program 6.10**

C language frequency measurement.

*continued from p. 311*

```
void Init(void) {
    asm sei           // make atomic
    TIOS |= 0x20;    // enable OC5
    TSCR1 = 0x80;   // enable
    TSCR2 = 0x01;   // 500 ns clock
    TIE |= 0x22;    // Arm OC5 and IC1
    TC5 = TCNT+Rate; // First in 10 ms
    TCTL4 = (TCTL4&0xF3)|0x04;
    /* C1F set on rising edges */
    Count = 0;      // Set up for first
    Done = 0;        // Set on measurements
    TFLG1 = 0x22;   // clear C5F, C1F
    asm cli
}
```

## 6.4 Conversion Between Frequency and Period

### 6.4.1 Using Period Measurement to Calculate Frequency

Period and frequency are obviously related, so when faced with a problem that requires frequency information we could measure period and calculate frequency from the period measurement.

$$f = \frac{1}{p}$$

Assume we use the 16-bit period measurement interface described earlier as Example 6.2. In this system a global variable `Period` contains the measured period, with a range from 20  $\mu$ s to 32 ms and a resolution of 500 ns. This corresponds to a frequency range of 31 to 50,000 Hz. If we add another global variable, `Freq`, that will hold the calculated frequency in hertz, then a 32-bit by 16-bit divide can be used to calculate

$$\text{Freq} = \frac{2000000}{\text{Period}}$$

It is easy to see how the 20- $\mu$ s to 32-ms period range maps into the 31- to 50,000 Hz frequency range, but mapping the 500-ns period resolution into an equivalent frequency resolution is a little more tricky. If the frequency is  $f$ , then the frequency must change to  $f + \Delta f$  such that the period changes by at least  $\Delta p = 500$  ns.  $1/f$  is the initial period, and  $1/(f + \Delta f)$  is the new period. These two periods differ by 500 ns. In other words,

$$\Delta p = \frac{1}{f} - \frac{1}{f + \Delta f}$$

We can rearrange this equation to relate  $\Delta f$  as a function of  $\Delta p$  and  $f$ .

$$\Delta f = \frac{1}{(1/f) - \Delta p} - f$$

This very nonlinear relationship, shown in Table 6.12, illustrates that although the period resolution is fixed at 500 ns, the equivalent frequency resolution varies from 500 Hz to 0.0005 Hz. If the signal frequency is restricted to values below 1413 Hz, then we can say the frequency resolution will be better than 1 Hz.

**Table 6.12**

Relationship between frequency resolution and frequency when calculated using period measurement.

Frequency (Hz)	Period ( $\mu$ s)	$\Delta f$ (Hz)
31,250	32	500
20,000	50	200
10,000	100	50
5,000	200	13
2,000	500	2
1,000	1,000	0.5
500	2,000	0.13
200	5,000	0.02
100	10,000	0.005
50	20,000	0.001
31.25	32,000	0.0005

## 6.4.2 Using Frequency Measurement to Calculate Period

Similarly, when faced with a problem that requires a period measurement, we could measure frequency and calculate period from the frequency measurement.

$$p = \frac{1}{f}$$

A similar nonlinear relationship exists between the frequency resolution and period resolution. In general, the period measurement approach will be faster, but the frequency measurement approach will be more robust in the face of missed edges or extra pulses. See Exercise 6.3 for a comparison between frequency and period measurement.

**Example 6.9** Design a system that measures period with a resolution of 1 ms.

**Solution** The objective is to measure period with a resolution of 1 ms. The digital logic signal is connected to an input capture. Each rising edge will generate an input capture interrupt. In addition, output compare is used to increment a software counter, `Time`, every 1 ms. The period is calculated as the number of 1-ms output compare interrupts between one rising edge of the input capture pin to the other rising edge of the input capture pin. For example, if the period is 8192  $\mu$ s, there will be eight output-compare interrupts between successive input capture interrupts (Figure 6.25).

A C language implementation is presented as Program 6.11. The period measurement counts the number of 1-ms intervals between successive rising edges. The period measurement resolution is 1 ms, because the period must increase by at least 1 ms for there to be a different number of counts. The range is 0 to 65 s, and the precision of 16 bits is determined by the size of the `Time` counter. If the gadfly loop in the ritual is removed, the first measurement will be inaccurate, but the subsequent measurements will be okay. There is an IC1 interrupt each period, and the external signal is connected to IC1. A real-time clock with a 1-ms rate is created with OC3. The foreground/background threads communicate via three shared globals. The background thread will update `Period` with a new measurement and set `Done`. When `Done` is set, the foreground thread will read the global `Period` and clear `Done`. `OverFlow` is set by the background and read by the foreground if the period is larger than 65 s. The ritual will wait for the first rising edge. In this way the first measurement will be correct. The only problem is that if no signal exists (the input capture pin is a constant level), this ritual software will hang at this wait.

```
unsigned short Period;      /* Period in msec */
unsigned char OverFlow;    /* Set if Period is too big*/
unsigned char Done;        /* Set each rising edge of IC1 */
```

There is a private global (only accessed by the background threads and not the foreground).

```
unsigned short Cnt; /* number of msec in one period */
```

The output compare handler simply counts 1-ms time intervals. The input capture interrupt occurs on the external input. This ISR occurs when a new measurement is ready (Program 6.11).

### Program 6.17

C language implementation of frequency measurement.

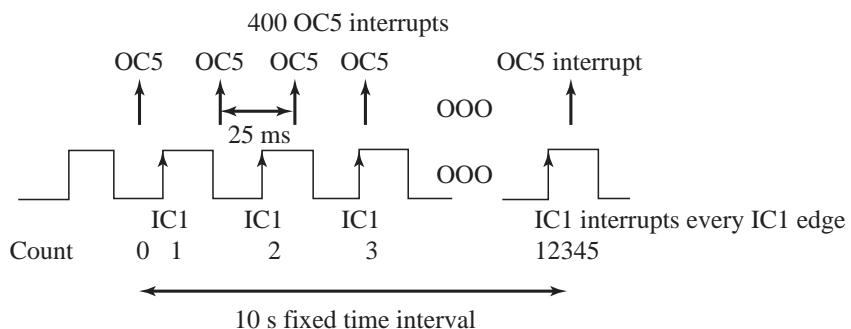
```
// MC9S12C32, CodeWarrior C
#define RESOLUTION 2000
void interrupt 11 TC3handler(void){
    TFLG1 = 0x08; // Acknowledge
    TC3 = TC3+RESOLUTION; // every 1 ms
    Cnt++;
    if(Cnt==0) OverFlow=0xFF;
}
void interrupt 9 TC1handler(void){
    TFLG1 = 0x02; // ack, clear C1F
    if(OverFlow){
        Period = 65535; // greater than 65535
        OverFlow = 0;
    }
    else
        Period = Cnt;
    Cnt = 0; // start next measurement
    Done = 0xFF;
}
void Ritual(void){
    asm sei // make atomic
    TIOS |= 0x08; // enable OC3
    TSCR1 = 0x80; // enable
    TSCR2 = 0x01; // 500 ns clock
    TFLG1 = 0x0A; // Clear C3F,C1F
    TFLG1 = 0x0A; // Clear C3F,C1F
    TIE = 0x0A; // Arm OC3 and IC1
    TCTL4 = (TCTL4&0xF3)|0x04;
/* C1F set on rising edges */
    while((TFLG1&0x02)==0); // wait rising
    TFLG1 = 0x02; // Clear C1F
    TC3 = TCNT+RESOLUTION;
    Cnt=0; OverFlow=0; Done=0;
    asm cli
}
```

**Example 6.10** Design a system that measures frequency with a resolution of 0.1 Hz.

**Solution** If we count pulses in a 10-s time interval, then the number of pulses represents the signal frequency with units 1/10 s, or 0.1 Hz (e.g., if there are 12,345 pulses during the 10-s interval, then the frequency is 1234.5 Hz). For this system, the measurement resolution is 0.1 Hz. For example, the frequency would have to increase to 1234.6 Hz (or decrease to 1234.4 Hz) for the change (a different number of IC1 interrupts) to be reliably detected. OC5 interrupts every 25 ms, and it takes 400 OC5 interrupts to create the 10-s time delay. The number of IC1 interrupts in the 10-s interval represents the input frequency with units of 0.1 Hz (Figure 6.21).

**Figure 6.21**

Basic timing involved in frequency measurement.



The highest frequency that can be measured will be determined by how fast the Input Capture interrupt handler can count pulses. We should select a counter precision (e.g., 8, 16, or 24 bits) so that the highest frequency does not overflow the counter. The digital logic signal is connected to IC1. PT1 Each rising edge will generate an input capture interrupt.

A C language implementation is presented as Program 6.12. To simplify the implementation, we will assume the maximum frequency is 6.5 kHz so that a 16-bit counter can be used. The frequency measurement is the number of rising edges in a 10-s interval. A periodic interrupt is generated with OC5. The foreground/background threads communicate via two shared globals. The background thread will update `Freq` with a new measurement and set `Done`. When `Done` is set, the foreground thread will read the global `Freq` and clear `Done`. Just like the other frequency measurement system, a frequency of 0 will result in no input capture interrupts, and the system will properly report the frequency of 0.

```
unsigned short Freq; /* Frequency with units of 0.1 Hz */
unsigned char Done; /* Set each measurement, every 10 s */
```

There are two private globals (only accessed by the background threads and not the foreground.)

```
unsigned short FourHundred; // Used to create 10sec interval
unsigned short Count; // Number of rising edges
```

The input capture interrupt counts the number of cycles, and the output compare interrupt is used to mark the end of the fixed time measurement (Program 6.12).

### Program 6.12

C language implementation of frequency measurement.

```
// MC9S12C32, CodeWarrior C
#define PERIOD 50000 // 25 ms
void interrupt 9 TC1handler(void){
    Count++; // number of rising edges
    TFLG1 = 0x02; // ack, clear C1F
}
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20; // Acknowledge
    TC5 = TC5+PERIOD; // every 25 ms
    if (++FourHundred==400){
        Freq = Count; // 0.1 Hz units
        FourHundred = 0;
        Done = 0xff;
        Count = 0; // Setup for next
    }
}
```

*continued on p. 316*

**Program 6.12**

C language implementation of frequency measurement.

*continued from p. 315*

```
void Init(void) {
    asm sei           // make atomic
    TIOS |= 0x20;    // enable OC5
    TSCR1 = 0x80;   // enable
    TSCR2 = 0x01;   // 500 ns clock
    TIE = 0x22;     // Arm OC5 and IC1
    TCTL4 = (TCTL4&0xF3)|0x04; // rising
    Count = 0;      // Set up for first
    Done = 0;       // Set on measurement
    FourHundred = 0;
    TC5 = TCNT+PERIOD; // First in 25 ms
    TFLG1 = 0x22;   // Clear C5F,C1F
    asm cli
}
```

## 6.5 Pulse Accumulator

### 6.5.1 9S12 Pulse Accumulator Details

The 9S12 pulse accumulator is a 16-bit read/write counter that can operate in either of two modes. External event counting mode can be used for counting events or frequency measurement. We will use gated time accumulation mode for pulse width measurement. The I/O ports involved in the 9S12 pulse accumulator are shown in Table 6.13.

Address	msb														lsb	Name
Address	Bit 7	6	5	4	3	2	1	0	Bit 0	Name						
\$0062	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	PACNT
\$0046	TEN	TSWAI	TSBCK	TFFCA	0	0	0	0	0	0	0	0	0	0	0	TSCR1
\$0060	0	PAEN	PAMOD	PEDGE	CLK1	CLK0	PAOVI	PAI	PAOVI	PAI	PAIF	PAIF	PAIF	PAIF	PAIF	PACTL
\$0061	0	0	0	0	0	0	PAOVF	PAIF	PAOVF	PAIF	PAFLG	PAFLG	PAFLG	PAFLG	PAFLG	PAFLG
\$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PT1	PT0	PTT	PTT	PTT	PTT	PTT	PTT
\$0242	DDRT7	6	5	4	3	2	1	Bit 0	Bit 0	Bit 0	DDRT	DDRT	DDRT	DDRT	DDRT	DDRT

**Table 6.13**

9S12 I/O ports used by the pulse accumulator.

**DDRT7** is the Data Direction bit for PT7. Normally, the DDRT7 bit is cleared so that PT7 is an input, but even if it is configured for output, PT7 still drives the pulse accumulator. **PAEN** is the Pulse Accumulator System Enable bit. We set PAEN to one in order to activate the pulse accumulator. The PAMOD and PEDGE bits select the operation mode, as shown in Table 6.14.

PAMOD	PEDGE	Mode	Action on Clock	Sets PAIF
0	0	Event counting	PT7 falling edge increments PACNT	Falling edge PT7
0	1	Event counting	PT7 rising edge increments PACNT	Rising edge PT7
1	0	Gated time accumulation	Counts when PT7=1	Falling edge PT7
1	1	Gated time accumulation	Counts when PT7=0	Rising edge PT7

**Table 6.14**

9S12 pulse accumulator operation modes.

In the event counting mode, the 16-bit counter (**PACNT**) is incremented on either the rising edge or the falling edge of PT7. The maximum clocking rate for the external event counting mode is the E clock frequency divided by two. Event counting mode does not require the timer to be enabled.

In the gated time accumulation mode, a free-running clock (E clock divided by 64) increments the 16-bit counter. In this mode, the E clock divided by 64 increments PACNT while the PT7 input is active. Gated accumulation mode does require the TEN in the TSCR1 register to be set.

The **PAOVF** status bit is set each time the pulse accumulator count rolls over from \$FFFF to \$0000. To clear this status bit, we write a one to the PAFLG register bit 1. The **PAOVI** will arm the device so that a pulse accumulator interrupt is requested when PAOVF is set. When **PAOVI** is zero, pulse accumulator overflow interrupts are disarmed. The **PAIF** status bit is automatically set each time a selected edge is detected at the PT7 pin (PEDGE=0 means falling edge, and PEDGE=1 means rising edge). To clear this status bit, write to the PAFLG register bit 1. The **PAII** will arm the device so that a pulse accumulator interrupt is requested when PAIF is set. When PAII is zero, pulse accumulator input interrupts are disarmed.

**Observation:** The PACNT input and timer channel 7 use the same pin PT7. To use the pulse accumulator, disconnect PT7 from the output compare logic by clearing bits OM7 and OL7. Also clear the channel 7 output compare 7 mask bit, OC7M7.

## 6.5.2 Frequency Measurement

The goal is to measure frequency in Hz, which in this case is defined as the number of falling edges that occur in one second. The signal to be measured will be connected to the pulse accumulator input, which is PT7. The frequency measurement function, shown in Program 6.13, enables the pulse accumulator and selects event counting mode. When measuring frequency it usually doesn't matter whether we count rising or falling edges. However, in this case, falling edges will be counted. The approach will be to initialize the pulse accumulator to event counting, clear the count, wait one second, then read the counter. Since frequency is defined as the number of edges in one second, the value in the PACNT after the one-second time delay will be frequency in Hz. The 9S12 can measure 0 to 65535 Hz. In both cases, the frequency resolution (which is the smallest change in frequency that can be distinguished) will be 1 Hz. In general, the frequency resolution will be one divided by the fixed time during which counts are measured. The PAOVF bit will be set if the input frequency exceeds the measurement range.

### Program 6.13

Frequency measurement using the pulse accumulator.

```
;9S12 measures 0 to 65535 Hz
;returns Reg D = freq in Hz
Freq bclr DDRT,#$80 ;PT7 is input
    movb #$40,PACTL ;count falling
    movw #0,PACNT
    movb #$02,PAFLG ;clear PAOVF
    ldy #100
    bsr Timer_Wait10ms ;Program 2.6
    brclr PAFLG,#$02,ok ;check PAOVF
bad ldd #65535          ;too big
    bra out
ok   ldd PACNT ;units in Hz
out  rts
```

**Checkpoint 6.14:** What is the frequency resolution of the system implemented in Program 6.13? What does it mean?

**Checkpoint 6.15:** How do you modify Program 6.13 so that it measures frequency with a resolution of 1 kHz?

### 6.5.3 Pulse-Width Measurement

The goal is to measure pulse width, which in this case will be defined as the time the input signal is high. Again, the input signal will be connected to the pulse accumulator input, which is PT7. The pulse-width measurement function, shown in Program 6.14, enables the pulse accumulator and selects gated accumulation mode. In this case, PEDGE is set to zero, so the PACNT will accumulate when the input is high. With PEDGE equal to zero, the PAIF will be set on the falling edge of the input, signaling that the pulse-width measurement is complete. The approach will be to initialize the pulse accumulator to gated accumulation mode, clear the count, wait for PAIF to be set, and then read the counter. Since PACNT counts while the input is high, the value in this counter will represent the width of the pulse. The pulse-width resolution is the smallest change in pulse width that can be distinguished. In general, the pulse-width resolution will be the period of the free-running clock used to increment the counter. Assuming the 9S12 E clock is set to the default value of 4 MHz, its pulse-width resolution will be 16  $\mu$ s. The 9S12 can measure 16  $\mu$ s to 1.05 s. The PAOVF bit will be set if the input pulse width exceeds the measurement range. If the input signal has a pulse width of 1 ms, then the function will give a result of 1000/16 or 62.

#### Program 6.14

Pulse-width measurement using the pulse accumulator.

```
;9S12 measures 16us to 1.05s
;returns Reg D = pulse width in 16us
Puls bclr DDRT,#$80 ;PT7 is input
    movb #$60,PACTL ;measure high
    movw #0,PACNT
    movb #$02,PAFLG ;clear PAOVF
loop brclr PAFLG,#$01,loop
    brclr PAFLG,#$02,ok ;check PAOVF
bad ldd #65535          ;too big
    bra out
ok   ldd PACNT ;units in 16us
out  rts
```

**Checkpoint 6.16:** What is the pulse-width resolution of the system implemented in Program 6.14? What does it mean?

**Checkpoint 6.17:** What will be the output of Program 6.14 if the pulse width is 1234.5  $\mu$ sec?

## 6.6 Pulse-Width Modulation on the MC9S12C32

Earlier in the chapter we created a pulse-width modulated output using output compare. The disadvantages of that approach include the software overhead to service the periodic interrupts and its inability to create 0% and 100% signals. Appreciating the importance of pulse-width modulation, Freescale added dedicated hardware to handle PWM, not previously available in the 6811. Table 6.15 shows the MC9S12C32 registers used to create pulse-width modulated outputs. Each of three PWM devices can be configured either as two 8-bit channels or one 16-bit channel. In particular, each of the 16-bit registers in Table 6.15 could be considered as two separate 8-bit registers. For example, the 16-bit register **PWMPER01** could be considered as the two 8-bit registers **PWMPER0** (at address

\$00F2) and **PWMPER1** (at address \$00F3). Some versions of the MC9S12C32 do not have all the Port P pins available, as described in Table 1.14 back in Chapter 1. The PWM channel 5 always uses output PP5 (which is available on all MC9S12C32 packages), but the other channels can be connected to either Port P or Port T. If a bit in the **MODRR** register is 1, the corresponding Port T pin is connected to the PWM system. If the bit is 0, the Port P pin is connected to the PWM and the corresponding Port T pin is connected to the timer system.

Address	msb																lsb	Name
\$00F2	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PWMPER01	
\$00F4	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PWMPER23	
\$00F6	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PWMPER45	
\$00F8	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PWMMDTY01	
\$00FA	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PWMMDTY23	
\$00FC	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PWMMDTY45	

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0247	0	0	0	MODRR4	MODRR3	MODRR2	MODRR1	MODRR0	MODRR
\$00E0	0	0	PWME5	PWME4	PWME3	PWME2	PWME1	PWME0	PWME
\$00E1	0	0	PPOL5	PPOL4	PPOL3	PPOL2	PPOL1	PPOL0	PWMPOL
\$00E2	0	0	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0	PWMCLK
\$00E3	0	PCKB2	PCKB1	PCKB0	0	PCKA2	PCKA1	PCKA0	PWMPRCLK
\$00E4	0	0	CAE5	CAE4	CAE3	CAE2	CAE1	CAE0	PWMCAE
\$00E5	0	CON45	CON23	CON01	PSWAI	PFRZ	0	0	PWMCTL
\$00E8	Bit 7	6	5	4	3	2	1	Bit 0	PWMSCLA
\$00E9	Bit 7	6	5	4	3	2	1	Bit 0	PWMSCLB

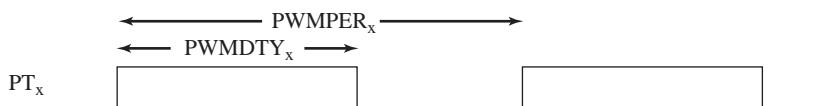
**Table 6.15**

MC9S12C32 registers used to configure pulse-width modulated outputs.

The **PWME** register allows you to enable/disable individual PWM channels. The **PWMCTL** register is used to concatenate two 8-bit channels into one 16-bit PWM. For example, if the **CON23** is 1, then channels 2 and 3 become one 16-bit channel with the output generated on PP3 (if MODRR3=0) or PT3 (if MODRR3=1). Concatenated channels are controlled using the higher of the two channels. For example, concatenated channel 23 is enabled with bit PWME3. The **PWMPOL** register specifies the polarity of the output. Figure 6.22 shows a PWM output for the case when the **PPOL<sub>x</sub>** bit is 1. The output will be high for the number of counts in the **PWMMDTY** register. The **PWMPER** register contains the number of counts in one complete cycle. The duty cycle is defined as the fraction of time the signal is high, calculated as a percent, depending on PWMPER and PWMMDTY.

$$\text{DutyCycle} = 100\% * \text{PWMMDTY}_x / \text{PWMPER}_x$$

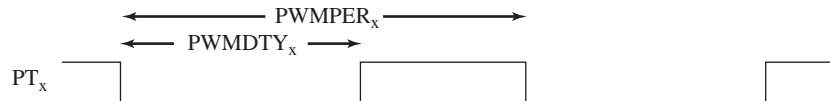
**Figure 6.22**  
PWM output generated when **PPOL<sub>x</sub>**=1.



If the **PPOL<sub>x</sub>** bit is 0, the output will be low for the number of counts in the **PWMDTY<sub>x</sub>** register, as illustrated in Figure 6.23. The duty cycle, defined as a fraction of time the signal is high, is

$$\text{DutyCycle} = 100\% * (\text{PWMPER}_x - \text{PWMDTY}_x) / \text{PWMPER}_x$$

**Figure 6.23**  
PWM output generated when PPOL=0.



There are many possible choices for the clock. The base clock is derived from the E clock. Activating the PLL affects the E clock and hence will affect the PWM generation. Channels 0, 1, 4, and 5 use either clock A or clock SA. Channels 2 and 3 use either clock B or clock SB. The six bits in the **PWMPCRCLK** register, as shown in Table 6.16, determine the relationship of clocks A and B with the E clock.

**Table 6.16**  
Clock A and Clock B prescale in PWMCLK.

PCKB2	PCKB1	PCKB0	Clock B	PCKA2	PCKA1	PCKA0	Clock A
0	0	0	E	0	0	0	E
0	0	1	E/2	0	0	1	E/2
0	1	0	E/4	0	1	0	E/4
0	1	1	E/8	0	1	1	E/8
1	0	0	E/16	1	0	0	E/16
1	0	1	E/32	1	0	1	E/32
1	1	0	E/64	1	1	0	E/64
1	1	1	E/128	1	1	1	E/128

It is possible to divide the A and B clocks further using the **PWMSCLA** and **PWMSCLB** registers. The period of the SA clock is the period of the A clock divided by two times the value in the PWMSCLA register. Similarly, the period of the SB clock is the period of the B clock divided by two times the value in the PWMSCLB register. If the value in PWMSCLA(B) is 0, then a divide by 512 is selected. The clock used for each channel is determined by the **PWMCLK** register. The period of the PWM output is the period of the selected clock times the value in the **PWMPER** register.

PCLK5	=1 Clock SA is the clock source for PWM channel 5 =0 Clock A is the clock source for PWM channel 5
PCLK4	=1 Clock SA is the clock source for PWM channel 4 =0 Clock A is the clock source for PWM channel 4
PCLK3	=1 Clock SB is the clock source for PWM channel 3 =0 Clock B is the clock source for PWM channel 3
PCLK2	=1 Clock SB is the clock source for PWM channel 2 =0 Clock B is the clock source for PWM channel 2
PCLK1	=1 Clock SA is the clock source for PWM channel 1 =0 Clock A is the clock source for PWM channel 1
PCLK0	=1 Clock SA is the clock source for PWM channel 0 =0 Clock A is the clock source for PWM channel 0

Let **n** be the 3-bit value for PCKA2-0 in the PWMCLK register. Let the E clock period be **Period<sub>E</sub>**. Then if the A clock is selected for channel x, the periods of the A clock and PWM output will be

$$\begin{aligned} \text{Period}_A &= 2^n * \text{Period}_E \\ \text{Period}_{PTx} &= 2^n * \text{PWMPER}_x * \text{Period}_E \end{aligned}$$

If the SA clock is selected for channel x, the periods of the SA clock and PWM output will be

$$\begin{aligned} \text{Period}_{SA} &= 2^n * 2 * \text{PWMSCLA} * \text{Period}_E \\ \text{or} \quad \text{Period}_{SA} &= 2^n * 512 * \text{Period}_E \text{ (if PWMSCLA equals 0)} \end{aligned}$$

$$\begin{aligned} \text{Period}_{PTx} &= 2^n * 2 * \text{PWMSCLA} * \text{PWMPER}_x * \text{Period}_E \\ \text{or} \quad \text{Period}_{PTx} &= 2^n * 512 * \text{PWMPER}_x * \text{Period}_E \text{ (if PWMSCLA equals 0)} \end{aligned}$$

The design of a PWM system considers three factors. The first factor is the period of the PWM output. Most applications choose a period, initialize the waveform at that period, and adjust the duty cycle dynamically. The second factor is precision, which is the total number of duty cycles that can be created. An 8-bit PWM channel may have up to 256 different outputs, while a 16-bit channel can potentially create up to 65536 different duty cycles. More specifically, since the duty cycle register must be less than or equal to the period register (e.g.,  $\text{PWMDTY}_x \leq \text{PWMPER}_x$ ), the precision of the system will equal  $\text{PWMPER}_x + 1$  in alternatives. The final consideration is the number of channels. The MC9S12C32 supports up to six 8-bit channels or three 16-bit channels. It is possible to mix and match, creating, for example, four 8-bit channels and one 16-bit channel. Different versions of the 9S12 will have from 0 to 8 channels of PWM.

In this first design example, we will create a 10ms period 8-bit PWM output using channel 0 generated on the PT0 output. In order to maximize precision, it is best to create the 10 ms period using as large a value in PWMPER0 as possible. We have the limitation that the prescale and PWMPER0 factors will be integers. Since 10 ms/256 equals 39.0625  $\mu$ s, we need a clock just larger than 39  $\mu$ s. The fastest clock that can be used is 40  $\mu$ s, resulting in PWMPER0 equal to 250. Assuming the E clock period is 250 ns, the prescale needs to be 40/0.25 or 160. There are a number of ways to make this happen, but one way is to select Clock A to be E/16, create SA=A/10, then select the SA clock for channel 0, as shown in Program 6.15.

<pre>;MC9S12C32 assembly PWM_Init0          ;10ms PWM on PT0     bset MODRR,#\$01 ;PT0 with PWM     bset PWME,#\$01  ;enable chan 0     bset PWMPOL,#\$01 ;high then low     bset PWMCLK,#\$01 ;Clock SA     ldaa PWMPRCLK     anda #\$F8     oraa #\$04     staa PWMPRCLK      ;A=E/16     movb #5,PWMSCLA     ;SA=A/10     movb #250,PWMPERO    ;10ms period     clr PWMDTY0          ;initially off     rts     PWM_Duty0           ;RegA is duty cycle     staa PWMDTY0        ;0 to 250     rts</pre>	<pre>// MC9S12C32 C // 10ms PWM on PT0 void PWM_Init(void){     MODRR  = 0x01; // PT0 with PWM     PWME  = 0x01; // enable channel 0     PWMPOL  = 0x01; // PT0 high then low     PWMCLK  = 0x01; // Clock SA     PWMPRCLK = (PWMPRCLK&amp;0xF8) 0x04; // A=E/16     PWMSCLA = 5; // SA=A/10, 0.25*160=40us     PWMPERO = 250; // 10ms period     PWMDTY0 = 0; // initially off } // Set the duty cycle on PT0 output void PWM_Duty0(unsigned char duty){     PWMDTY0 = duty; // 0 to 250 }</pre>
--	---

### Program 6.15

Implementation of an 8-bit PWM output.

**Checkpoint 6.18:** State another way to create a prescale of 160 on channel 0.

**Checkpoint 6.19:** How would you modify Program 6.15 to have a 100ms period?

In this second design example, we will create a 1s period 16-bit PWM output using concatenated Channel 23 generated on the PT3 output. In order to maximize precision, it is best to create the 1s period using as large a value in PWMPER23 as possible. Since 1s/65536 equals 15.2587890625  $\mu$ s, we need a clock just larger than 15  $\mu$ s. The fastest clock that can be used is 16  $\mu$ s, resulting in PWMPER23 equal to 62500. Assuming the E clock period is 250 ns, the prescale needs to be 16/0.25 or 64. There are a number of ways to make this happen, but one way is to make Clock B to be E/64, then select the B clock for Channel 23, as shown in Program 6.16.

<pre>;MC9S12C32 assembly PWM_Init3          ;1s PWM on PT3     bset MODRR,#\$08 ;PT0 with PWM     bset PWME,#\$08  ;enable chan 3     bset PWMPOL,#\$08 ;high then low     bclr PWMCLK,#\$08 ;Clock B     bset PWMCTL,#\$20 ;concat 2+3     ldaa PWMPRCLK     anda #\$8F     oraa #\$60     staa PWMPRCLK      ;B=E/64     movw #62500,PWMPER23 ;1s period     movw #0,PWMDDTY23   ;off     rts     PWM_Duty3          ;RegD is duty cycle     std  PWMDTY0        ;0 to 62500     rts</pre>	<pre>// MC9S12C32 C // 1s PWM on PT3 void PWM_Init(void){     MODRR  = 0x08; // PT3 with PWM     PWME  = 0x08; // enable channel 3     PWMPOL  = 0x08; // PT3 high then low     PWMCLK &amp;= ~0x08; // Clock B     PWMCTL  = 0x20; // Concatenate 2+3     PWMPRCLK = (PWMPRCLK&amp;0x8F) 0x60; // B=E/64     PWMPER23 = 62500; // 1s period     PWMDTY23 = 0;     // initially off } // Set the duty cycle on PT3 output void PWM_Duty(unsigned short duty){     PWMDTY23 = duty; // 0 to 62500 }</pre>
--	--

### Program 6.16

Implementation of a 16-bit PWM output.

**Checkpoint 6.20:** What would be the effect of creating the 1s output using a 1ms SB clock and a PWMPER23 value of 1000?

**Checkpoint 6.21:** Are Programs 6.15 and 6.16 friendly enough to be used together?

## 6.7 Exercises

**D6.1** The objective of this problem is to measure the frequency of a square wave connected to IC1. You may use only IC1 and OC5 (i.e., no other 9S12 I/O feature or external device is allowed). The frequency range is 0 to 200 Hz, and the *resolution* is 0.01 Hz. For example, if the frequency is 56.783 Hz, then your software will set the global Freq to 5678. The C program in Example 6.10 measures frequency with units of 0.1 Hz. Make modifications to this program so that the resolution is improved to 0.01 Hz. Don't worry about frequencies above 200 Hz.

**D6.2** When a debugger wishes to single-step a program that exists in RAM, it can replace opcodes one at a time with SWI (it will have to replace two opcodes when single-stepping a conditional branch). This approach is not feasible when testing software stored in ROM or PROM. In this exercise you will use output compare interrupts to implement single-step debugging. Your approach will work for programs in RAM or ROM.

- a) Write a debugging function that initializes the OC5 interrupt then calls the UserRoutine. The first OC5 interrupt should occur after exactly one instruction of the UserRoutine has been executed. You may assume the UserRoutine has no I/O parameters. You may also

assume that `UserRoutine` has no interrupts of its own and it does not disable interrupts. When the `UserRoutine` returns back to your function, you should shut off OC5. You can start with the following syntax and add the OC5 code. You may write your answer in assembly or C.

```
// Single step in C                                * Single Step in assembly
void debug(void (*UserRoutine)(void)){           debug    * X points to UserRoutine
// add stuff here to initialize OC5              * stuff here to initialize OC5
    (*UserRoutine)();                           jsr 0,X  call UserRoutine
// add stuff here to stop OC5                   * add stuff here stop OC5
}                                              rts
```

You are given two functions that you will call but do not need to write.

```
char GetChar(void); // waits for keyboard input and returns the ASCII code
void Display(void); // displays debugging information like registers
```

- b)** Write the OC5 interrupt handler that calls `Display` then `GetChar`. In a real debugger we would process the keyboard input and interact with the user, but in this simple solution the keyboard input is used only to pause. Before returning from the interrupt, you should set up OC5 so that the 9S12 will execute exactly one more instruction of `UserRoutine` before another OC5 interrupt is generated.

- D6.3** A microcomputer-based PID controller requires a frequency measurement to be performed every 20 ms. This problem will investigate three techniques:

- A.** Direct frequency measurement
- B.** Period measurement
- C.** A combined frequency/period measurement

The frequency range is 100 to 1000 Hz, and we wish to obtain the best frequency resolution possible under the constraint that a new frequency measurement must be available every 20 ms for the PID controller. Each method has two interrupt handlers:

1. A 20-ms real time clock using OC5
2. Falling edge of the square wave using 9S12 PJ0 key wake up.

There are five 16-bit unsigned global variables used by the three techniques:

<code>FREQ</code> is the frequency in hertz calculated during the OC5 handler	used by A, B, C
<code>CNT</code> is the number of falling edges in the current 20-ms interval	used by A, C
<code>FIRST</code> is the TCNT value at the time of the first falling edge	used by C
<code>PREVIOUS</code> is the TCNT value at the time of the previous falling edge	used by B
<code>LAST</code> is the TCNT value at the time of the last falling edge	used by B, C

- A. Direct frequency measurement** The ritual performs:

```
CNT=0;
```

The key wake-up interrupt handler performs:

```
PIFJ=0x01; // ack
CNT=CNT+1;
```

The OC5 interrupt handler performs:

```
Write OC5F=1;                                /* Acknowledge OC5 */
TC5=TC5+40000;
Calculates FREQ=50·CNT;
CNT=0;
```

The frequency resolution is 50 Hz regardless of the square-wave frequency.

**B. Period measurement** The ritual performs:

```
PREVIOUS =LAST =0;
```

The key wake-up interrupt handler performs:

```
// 9S12 code
PIFJ=0x01; // ack
PREVIOUS=LAST;
LAST=TCNT;
```

The OC5 interrupt handler performs:

```
Write OC5F=1; /* Acknowledge OC5 */
TC5=TC5+40000;
Calculates FREQ=????(answered in part a)
PREVIOUS =LAST =0;
```

**C. Combined frequency/period measurement** The ritual performs:

```
FIRST=LAST=0;
CNT=-1;
```

The key wake-up interrupt handler performs:

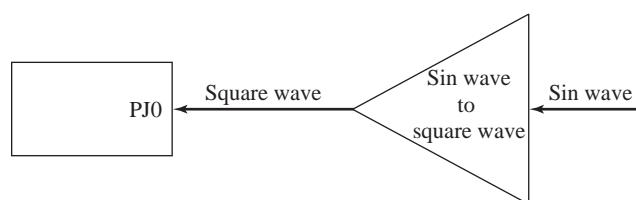
```
// 9S12 code
PIFJ=0x01; // ack
If (CNT== -1) FIRST=TCNT;
else LAST=TCNT;
CNT=CNT+1;
```

The OC5 interrupt handler performs:

```
Write OC5F=1; /* Acknowledge OC5 */
TC5=TC5+40000;
Calculates FREQ=????(answered in part d)
FIRST=LAST=0;
CNT=-1;
```

The hardware for each method is shown in Figure 6.24.

**Figure 6.24**  
Interface for  
Exercise 6.3.



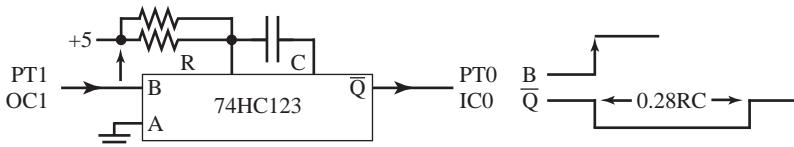
- Specify the equation used to calculate `FREQ` from `PREVIOUS` and `LAST` in the **Period measurement** OC5 handler (no software is required).
- What is the frequency resolution with units when the square-wave frequency is 100 Hz?
- What is the frequency resolution with units when the square-wave frequency is 1000 Hz?
- Specify the equation used to calculate `FREQ` from `FIRST`, `LAST`, and `CNT` in the **Combined frequency/period measurement** OC5 handler (no software is required).
- What is the frequency resolution with units when the square-wave frequency is 100 Hz?
- What is the frequency resolution with units when the square-wave frequency is 1000 Hz?

**D6.4** The objective of this problem is to measure body temperature using input capture (pulse-width measurement) (Figure 6.25). A shunt resistor is placed in parallel with a thermistor. The thermistor-shunt combination R has the following linear relationship for temperatures from 90 to 110°F.

$$R = 100 \text{ k}\Omega - (T - 90^\circ\text{F}) \cdot 1 \text{ k}\Omega/\text{ }^\circ\text{F} \quad \text{where } R \text{ is the resistance of the thermistor-shunt}$$

**Figure 6.25**

Interface for Exercise D6.4.



In other words, the resistance varies from  $100 \text{ k}\Omega$  to  $80 \text{ k}\Omega$  as the temperature varies from 90 to 110°F. The range of your system is 90 to 110°F and the resolution should be better than  $0.01^\circ\text{F}$ . You will use a 74HC123. On the rise of the B trigger input, the 123 creates a negative logic pulse on its output. The width of the pulse is about  $0.28 \cdot R \cdot C$ . Temperature will be measured 100 times a second.

- a) The pulse-width measurement resolution will be 125 ns for the 9S12. Choose the capacitor value so that this pulse width measurement resolution matches the desired temperature resolution of  $0.01^\circ\text{F}$ .
- b) Given this value of C, what is the pulse width at  $90^\circ\text{F}$ ? Give the answer in both microseconds and E clock cycles.
- c) Given this value of C, what is the pulse width at  $110^\circ\text{F}$ ? Give the answer in both microseconds and E clock cycles.
- d) Write the ritual that configures an output compare on PT1 as a periodic interrupt. A rising edge should occur every 10 ms. Configure an input capture on PT0 to measure the pulse width.
- e) Show the interrupt handler(s) that perform the temperature measurement tasks in the background and sets a global, `Temperature`, 100 times a second. The units of this unsigned decimal fixed point number are  $0.01^\circ\text{F}$ . For example, `Temperature` will vary from 9000 to 11000 as temperature varies from 90 to 110°F. The output compare interrupt will start the pulse (You may assume the interrupt vectors are properly established, but please use simple interrupt handler names like `TOC1handler()` and `TIC0handler()`).

**D6.5** Design a wind direction measurement instrument using the input capture technique. Again, you are given a transducer that has a resistance that is linearly related to the wind direction. As the wind direction varies from 0 to 360 degrees, the transducer resistance varies from 0 to  $1000 \Omega$ . The frequencies of interest are 0 to 0.5 Hz, and the sampling rate will be 1 Hz. One way to interface the transducer to the computer is to use an astable multivibrator like the 555. The period of a 555 timer is  $0.693 \cdot C_T \cdot (R_A + 2R_B)$ . The 555 output could be connected to the Input Capture Port Channel 7. (See Exercise D12.16.)

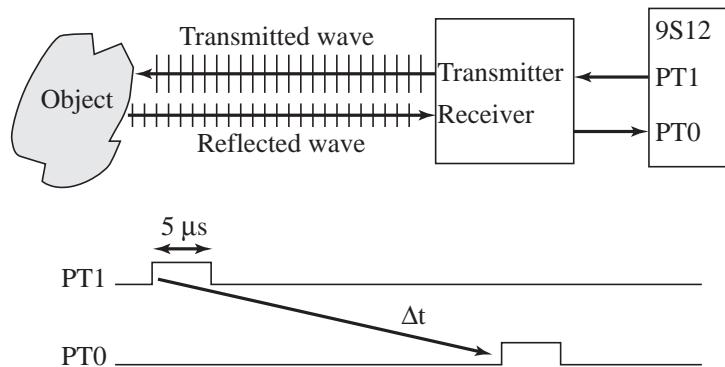
- a) Show the hardware interface.
- b) Write the ritual and gadfly function/subroutine that measures the wind direction and returns a 16-bit unsigned result with units of degrees (i.e., the value varies from 0 to 359). (You do not have to write software that samples at 1 Hz, simply a function that measures wind direction once.)

**D6.6** The objective of this problem is to design an underwater ultrasonic ranging system. The distance to the object,  $d$ , can vary from 1 to 100 m. The ultrasonic transducer will send a short  $5-\mu\text{s}$  sound pulse into the water in the direction of interest. The sound wave will travel at 1500 m/s and reflect off the first object it runs into. The reflected wave will also travel at 1500 m/s back to the transducer. The reflected pulse is sensed by the same transducer. Your system will trigger the electronics (give a  $5-\mu\text{s}$  digital pulse), measure the time of flight, then calculate the distance to the object. Using periodic interrupts, the software will issue a  $5-\mu\text{s}$  pulse out PT1 once a second. Using interrupting input capture, the software will measure the time of flight,  $\Delta t$ . The input capture interrupt handler will calculate distance  $d$  as a decimal fixed-point value with

units of 0.01 m and enter it into a FIFO queue. The main program will call the ritual, then get data out of the FIFO queue. The main program will call `Alarm()` if the distance is less than 15 m. You do not have to give the implementation of `Alarm()`. You may use any of the FIFOs in Chapter 4 without showing its implementation. To solve this problem on other microcomputers, you may modify the hardware in Figure 6.26 to conform to the available ports. As shown in the figure the time of flight,  $\Delta t$ , is measured from the rise of PT1 (9S12 output) to the rise of PT0 (9S12 input).

**Figure 6.26**

Interface for Exercise D6.6.



- Derive an equation that relates the distance  $d$  to the time of flight  $\Delta t$ .
- Use this equation to calculate the minimum and maximum possible time of flight  $\Delta t$ .
- Choose the TCNT rate that will satisfy the range and resolution requirements of this problem.
- Give the ritual that initializes the interface, including PORTT timer channels 0 and 1. Don't worry about the other six timer channels.
- Give the main program that first calls the ritual. The main program will empty the FIFO and call `Alarm()` if the distance drops below 15 m. Decide whether it is better to convert time (TCNT counts) to distance (decimal fixed point, 0.01m) here in the main program or in the interrupt handler.
- Give the `TC1handler()` periodic interrupt handler that issues pulses of about 5-μs duration on PT1 every 1 sec. Good interrupt software has no backward jumps.
- Give the `TC0handler()` interrupt handler that measures the time of flight and puts the result (either the count or the calculated distance) into the FIFO. Good interrupt software has no backward jumps. There may be additional sonic echoes, so only calculate the range of the first one and ignore the others.

- D6.7** The objective of this exercise is to interface a silicon-controlled rectifier (SCR) using input capture and/or output compare. A 120-Hz digital logic waveform (**Sync**) is available that is synchronized to the zero crossings of the 60-Hz alternating current wave. A 100-μs pulse on the digital logic **Control** signal will turn on the SCR. The SCR will automatically shut off on the next zero crossing of the 60-Hz wave. **Sync** will be an input to the microcomputer, and **Control** will be an output (Figure 6.27).

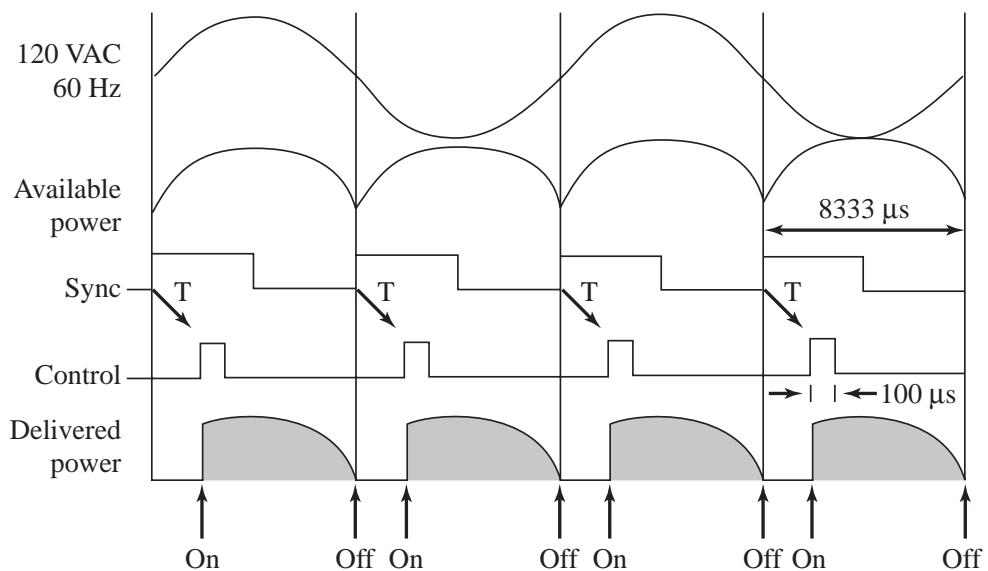
The software controls the amount of delivered power by adjusting the time  $T$ , which is the delay from the rising edge of the **Sync** input to the rising edge of **Control** output. To solve this problem on other microcomputers, you may modify the hardware in Figure 6.28 to conform to the available ports.

The delay  $T$  will vary from 300 to 8000 μs. When  $T$  is 300 μs, full power is being delivered. When  $T$  is 8000 μs, almost no power is delivered to the load. A 16-bit unsigned global variable

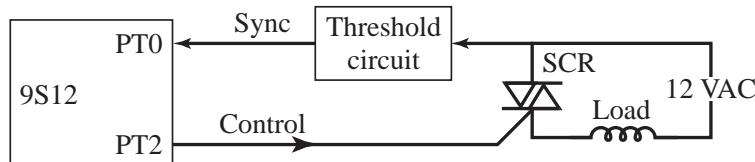
```
unsigned short T; // delay 2400 to 64000 in 125 ns clock cycles
```

will be set by the main program (which you will not write) and read by the interrupt software (which you will write).

**Figure 6.27**  
Timing for Exercise D6.7.



**Figure 6.28**  
Interface for  
Exercise D6.7.



- a) Give the ritual which initializes the interface. You may use any available feature of your microcomputer.
- b) Give the `TC2handler()` and `TC0handler()` interrupt handlers that implement this interface. Good interrupt software has no backward jumps.
- c) This is indeed an example of a real-time system. Give the upper bound on the latency of the `TC0handler()`. Don't calculate what it actually is but rather determine theoretically how fast it must be to work properly.

## 6.8 Lab Assignments

**Lab 6.1** The overall objective is to interface a joystick to the microcontroller. The joystick is made with two potentiometers. You can use two astable multivibrators (LM555) to convert the two resistances into two periods. Use two period measurement channels to estimate the X-Y position of the joystick. Organize the software interface into a device driver, and write a main program to test the interface.

**Lab 6.2** The overall objective is to measure temperature. A thermistor is a transducer with a resistance that is a function of its temperature. You can use an astable multivibrator (LM555) to convert the resistance into a period. Use a period measurement channel to estimate the resistance of the thermistor. Use a table lookup with linear interpolation to convert resistance to temperature. Organize the software interface into a device driver, and write a main program that outputs temperature to the SCI channel.

**Lab 6.3** The overall objective is to measure linear position. A slide-pot is a transducer with a resistance that is a function of the linear position of the slide. You can use an astable multivibrator (LM555)

to convert the resistance into a period. Use a period measurement channel to estimate the resistance of the potentiometer. Use a table lookup with linear interpolation to convert resistance to position. Organize the software interface into a device driver, and write a main program that outputs position to the SCI channel.

**Lab 6.4** The objective of this lab is to control a servo motor (see Figure 6.29). Interface a servo motor to the PWM output pin of the microcontroller. The desired angle is input from the SCI channel (connected to a PC running HyperTerminal). Organize the software interface into a device driver, and write a main program that inputs from the SCI channel and maintains the PWM output to the servo. Servos are a popular mechanism to implement steering in robotics. Ranging from micro servos with 15 oz-in torque to powerful heavy-duty sailboat servos, they all share several common characteristics. Servos allow you to control the position of the shaft, or **horn**. The servo senses where it is (the actual position). You specify where it should be (the desired position). When the servo receives a position, it attempts to move the servo horn to the desired position. The task of the servo, then, is to make the actual position the desired position. The first step to understanding how servos work is to understand how to control them. Power is usually between 4.8 and 6 V and should be separate from system power (as servos are electrically noisy). Even small servos can draw over an amp under heavy load, so the power supply should be appropriately rated. Servos are commanded through “pulse-width modulation” (PWM) signals sent through the command wire. Essentially, the width of a pulse defines the position. For example, sending a 1.5 ms pulse to the servo tells the servo that the desired position is at the midpoint, as shown in Figure 6.30. In order for the servo to hold this position, the command must be sent at about 50 Hz, or every 20 ms. If you were to send a pulse longer than 2.1 ms or shorter than 0.9 ms, the servo would attempt to overdrive (and possibly damage) itself. Once the servo has received the desired position (via the PWM signal) the servo must attempt to match the desired and actual positions. It does this by turning a small geared motor left or right. If, for example, the desired position is less than the actual position, the servo will turn to the left. On the other hand, if the desired position is greater than the actual position, the servo will turn to the right. In this manner, the servo “zeros-in” on the correct position. Should a load force the servo horn to the right or left, the servo will attempt to compensate. Note that there is no control mechanism for the speed of movement and, for most servos, the speed is specified in degrees per second. For more information refer to the data sheet of your servo.

**Figure 6.29**

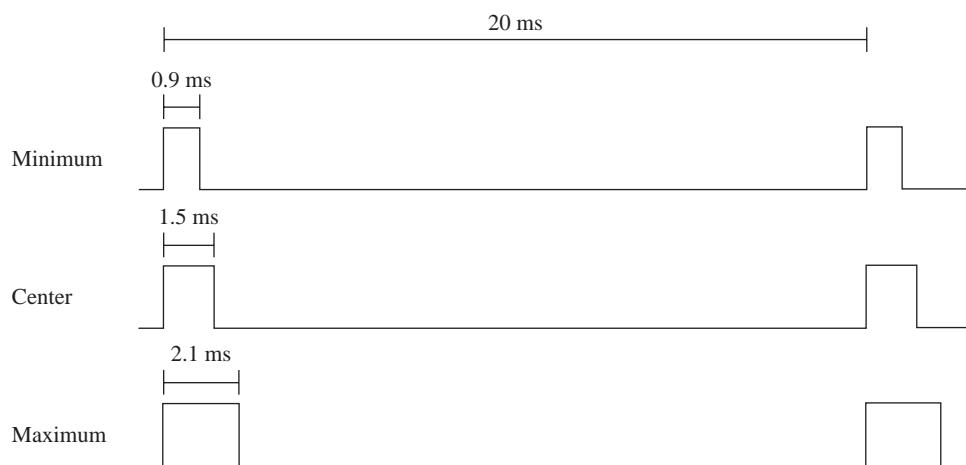
HS-311 servo

<http://www.hitecrcd.com>.



Courtesy of Jonathan Valvano.

**Figure 6.30**  
Typical servo command signals.



**Lab 6.5** Do Exercise 6.6 with the Ping))) sensor.

# 7 Serial I/O Devices

## Chapter 7 objectives are to:

- ❖ Discuss fundamental concepts associated with serial communication systems: asynchronous, synchronous, bandwidth, full-duplex, half-duplex, and simplex
- ❖ Present the RS232 and RS422 communication protocols, signals, and interface chips
- ❖ Write low-level device drivers that perform basic I/O with serial ports
- ❖ Discuss interfacing issues associated with performing the I/O as a background interrupt thread
- ❖ Define I<sup>2</sup>C as a mechanism to interface peripherals
- ❖ Introduce the fundamentals of the Universal Serial Bus (USB)

In many applications, a single dedicated microcomputer is insufficient to perform all the tasks of the embedded system. One solution would be to use a larger and more powerful microcomputer, and another approach would be to distribute the tasks among multiple microcomputers. This second approach has the advantages of modularity and expandability. To implement a distributed system, we need a mechanism to send and receive information between the microcomputers. A second scenario that requires communication is a central general-purpose computer linked to multiple remote embedded systems for the purpose of distributed data collection or distributed control. For situations where the required bandwidth is less than about 1000 bytes/s, the built-in serial ports of the microcomputer can be used.

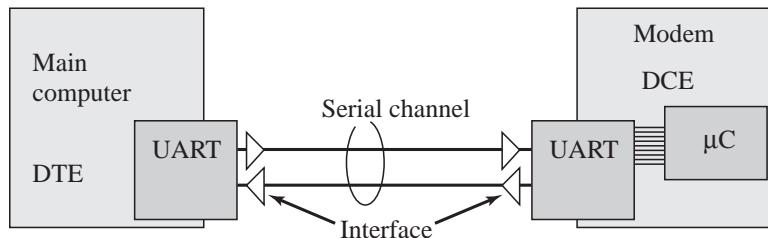
Chapter 7 deals with external devices that we connect to the serial I/O ports of our computer. In particular, we will interface terminals, keyboards, displays, printers, and other computers. In this chapter we will focus on serial channels that employ a direct physical connection between the microcomputers; later (in Chapter 14) we expand the communication system to include networks and modems.

## 7.1 Introduction and Definitions

*Serial communication* involves the transmission of one bit of information at a time. One bit is sent, a time delay occurs, then the next bit is sent. This section will introduce the use of serial communication as an interfacing technique for various microcomputer peripherals. Since many peripheral devices such as printers, keyboards, scanners, and mice have their own computers, the communication problem can be generalized to one of transmitting information between two computers. The *universal asynchronous receiver/transmitter (UART)* is the hardware port that implements the serial data transmission. Freescale calls it an *asynchronous communications interface adapter (ACIA)* or *SCI*. The *serial channel* is

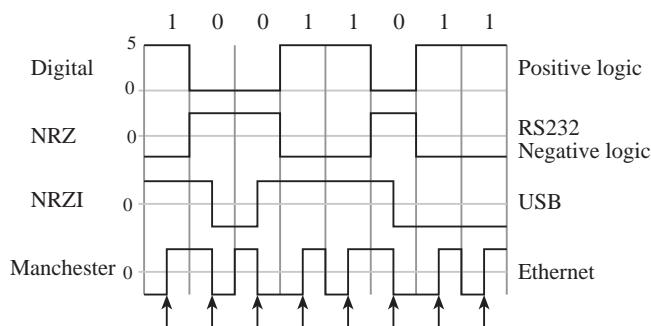
the collection of signals (or wires) that implement the communication. To improve bandwidth, remove noise, and increase range, we place interface logic between the digital logic UART port and the serial channel. We define the *data terminal equipment (DTE)* as the computer or a terminal and the *data communication equipment (DCE)* as the modem or printer (Figure 7.1).

**Figure 7.1**  
A serial channel connects a DTE to a DCE.



When transmitting in a serial fashion, there are many ways to encode binary information on the line (Figure 7.2). The goal is to maximize bandwidth and minimize errors. *Non-return-to-zero (NRZ)* encoding is a binary code in which the signal is never at zero voltage. There is more energy in the wire compared to typical 0/5 V digital logic, because there is always a voltage causing current to flow continuously. Energy is the fundamental property needed to communicate information over a distance. On a single wire, voltages are measured relative to ground (e.g., RS232). On a twisted pair, voltages are differential (e.g., RS422, USB). The binary values of 1 and 0 are encoded as positive or negative voltages. In positive logic, 1 is a positive voltage, and 0 is a negative voltage. In negative logic, the voltage representing 0 is higher than the voltage representing 1. *Non-return-to-zero-inverted (NRZI)* is a method of encoding binary signal as transitions or changes in the signal. Similar to NRZ, the signal in a NRZI protocol is never zero. The binary information is encoded as the presence or absence of a transition at a clock boundary, illustrated by the arrows in Figure 7.2. Both the transmitter and receiver will synchronize their clocks so the receiver knows when to look for the transition. A transition is a change from positive to negative or from negative to positive. For example, we could send a 1 by placing a transition on the signal, or send a 0 by causing no transition. Also, NRZI might take the opposite convention, as in Universal Serial Bus (USB), where a transition means 0 and a steady level means 1. *Manchester encoding* code encodes binary bits as either a low-to-high transition, or a high-to-low transition. Manchester encoding is a type of phase encoding, which means the information transmitted as phase shifts. It is used with the IEEE 802.3 Ethernet protocol. There is a fixed time period inside which one bit is transmitted. The original Manchester encoding scheme defined 0 as a low-to-high, and 1 as a high-to-low. However, the IEEE 802.3 convention defines the bits in the opposite manner. The transitions that signify 0 or 1 occur at the midpoint of a period, shown as arrows in Figure 7.2. There may be an additional transitions at the start of a period; these are extra

**Figure 7.2**  
Four encodings of the binary 10011011.



and do not signify data. Since every bit has at least one transition, it is easier for the receiver to align correctly or to synchronize its clock with the transmitter clock. However, the cost of this ease of synchronization is a doubling of the bandwidth requirement of the physical channel as compared to NRZ or NZRI encoding. Consider the case where a system is communicating a long sequence of zeros at 1 Mb/s. Using the NRZI USB encoding the line will be a square wave at 1 MHz. However, using IEEE 802.3 encoding, the line will be a square wave at 2 MHz.

The interface logic (e.g., DS275, MAX232, or MC145407) converts between TTL/MOS/CMOS logic levels and RS232 logic levels. In this protocol, a typical bidirectional channel requires three wires (Rx<sub>D</sub>, Tx<sub>D</sub>, ground). We use RS422 voltage levels when we want long cable lengths and high bandwidths. The binary signal is encoded on the RS422 line as a voltage difference (MC3486/3487, MAX485, 3691, 3695, 8921/8922/8923, 75176, or 78120). In this protocol, a typical bidirectional channel requires five wires (Rx<sub>D</sub><sup>+</sup>, Rx<sub>D</sub><sup>-</sup>, Tx<sub>D</sub><sup>+</sup>, Tx<sub>D</sub><sup>-</sup>, ground). Typical voltage levels are shown in Table 7.1.

**Table 7.1**

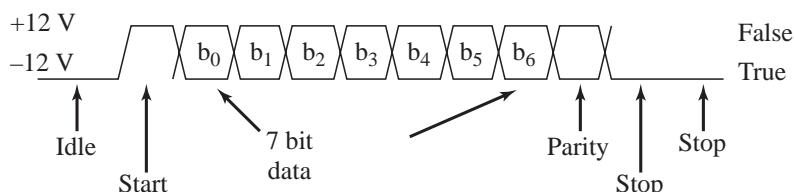
Voltage levels for the CMOS RS232 and RS422 protocols.

		Typical CMOS Level	Typical RS232 Level	Typical RS422 Level
True	Mark	+5 V	TxD = -12 V	(TxD <sup>+</sup> - TxD <sup>-</sup> ) = -3 V
False	Space	+0.1 V	TxD = +12 V	(TxD <sup>+</sup> - TxD <sup>-</sup> ) = +3 V

A *frame* is a complete and nondivisible packet of bits. A frame includes both information (e.g., data, characters) and overhead (start bit, error checking, and stop bits). A frame is the smallest packet that can be transmitted. The RS232 and RS422 protocols have one *start bit*, seven/eight data bits, no/even/odd *parity*, and one/1.5/two *stop bits* (Figure 7.3). The RS232 idle level is true (-12 V). The start bit is false (+12 V). A true data bit is -12 V, and a false data bit is +12 V. Stop bits are true (-12V).

**Figure 7.3**

A RS232 frame showing one start, seven data, one parity, and two stop bits.



**Observation:** The RS232 protocol always has one start bit and at least one stop bit.

**Observation:** RS232 and RS422 data channels are in negative logic because the true voltage is less than the false voltage.

**Checkpoint 7.1:** If the RS232 protocol has eight data bits, no parity, and one stop bit, how many total bits are in a frame?

Parity is generated by the transmitter and checked by the receiver. Parity is used to detect errors. For *even parity*, the number of 1s in the data plus parity is an even number. For *odd parity*, the number of 1s in the data plus parity is an odd number. If errors are unlikely, then operating without parity is faster and simpler. The *bit time* is the basic unit of time used in serial communication. It is the time between each bit. The transmitter outputs a bit, waits one bit time, then outputs the next bit. The start bit is used to synchronize the receiver with the transmitter. The receiver waits on the idle line until a start bit is first detected. After the true-to-false transition, the receiver waits a half a bit time. The half a bit time wait places the input sampling time in the middle of each data bit, giving the best tolerance to variations between the transmitter and receiver clock rates. We will discuss the detailed timing later in

the chapter. Next, the receiver reads one bit every bit time. The *baud rate* is the total number of bits (information, overhead, and idle) per time that is transmitted in the serial communication. Later in Chapter 14, we will define the baud rate of a modem as the number of sounds per second. But for now, we define,

$$\text{Baud rate} = \frac{1}{\text{bit time}}$$

We will define *information* as the data that the “user” intends to be transmitted by the communication system. Examples of information include:

- Characters to be printed on your printer
- A picture file to be transmitted to another computer
- A digitally encoded voice message communicated to your friend
- The object code file to be downloaded from the PC to the 9S12

We will define *overhead* as signals added by the “operating system” to the communication to affect reliable transmission. Examples of overhead include:

- Start bit(s), start byte(s), or start code(s)
- Stop bit(s), stop byte(s), or stop code(s)
- Error checking bits such as parity, cyclic redundancy check (CRC), and checksum
- Synchronization messages like ACK, NAK, XON, XOFF

Although, in a general sense overhead signals contain “information,” overhead signals are not included when calculating bandwidth or considering full-duplex, half-duplex, or simplex.

An important parameter in all communication systems is *bandwidth*. We will use the three terms *bandwidth*, *bit rate*, and *throughput* interchangeably to specify the number of information bits per time that is transmitted. These terms apply to all forms of communication:

- Parallel
- Serial
- Mixed parallel/serial

For serial communication systems, we can calculate:

$$\text{Bandwidth} = \frac{\text{number of information bits/frame}}{\text{total number of bits/frame}} \quad \text{baud rate}$$

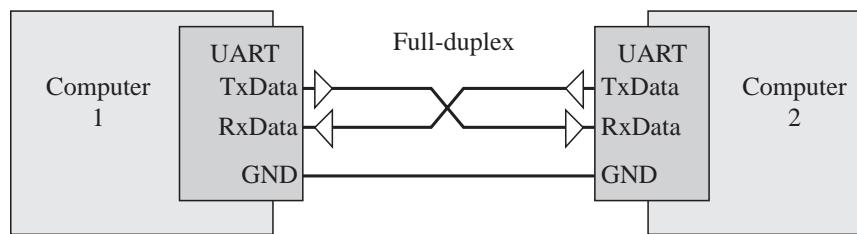
Bandwidth, latency, and reliability are the fundamental performance measures for a communication system. *Latency* is the time delay between when a message is sent and when it is received. For the simple systems in this chapter, at the physical layer, latency can be calculated as the frame size in bits divided by the baud rate in bits/sec. For example a RS232 protocol with 10-bit frames running at 9600-bps baud rate will take 1.04 ms to go from transmitter to receiver. *Reliability* is defined as the probability of corrupted data or the *mean time between failures* (MTBF). One of the confusing aspects of bandwidth is that it could mean two things. The *peak bandwidth* is the maximum achievable data-transfer rate over short periods during times when nothing else is competing for resources. When we say the bandwidth of a serial channel with 10-bit frames and a baud rate of 9600 bps is 960 bytes/s, we are defining peak bandwidth. At the component level, it is appropriate to specify peak bandwidth. However, on a complex system, there will be delays caused by the time it takes software to run, and there will be times when the transmission will be stalled due to conditions like full or empty FIFOs. The *sustained bandwidth* is the achievable data-transfer rate over long periods of time and under typical usage and conditions. At the system level, it is appropriate to specify sustained bandwidth.

The design parameters that affect bandwidth are capacitance and power. It takes energy to encode each bit; therefore, the bandwidth in bits per second is related to the power, which is energy per second. Capacitance exists because of the physical proximity of the wires in the cable. The time constant  $\tau$  of a simple RC circuit is  $R \cdot C$ . An increase in capacitance will decrease the slew rate of the signal (see Figure 1.13), limiting the rate at which signals can change, thereby reducing the bandwidth of the digital transmission. However, we can increase the slew rate by using more power. We can increase the energy over the same time period by increasing voltage, increasing current, or decreasing resistance.

A *full-duplex communication system* allows information (data, characters) to transfer simultaneously in both directions. A *full-duplex channel* allows bits (information, error checking, synchronization, or overhead) to transfer simultaneously in both directions (Figure 7.4).

**Figure 7.4**

A full-duplex serial channel connects two DTEs (computers).



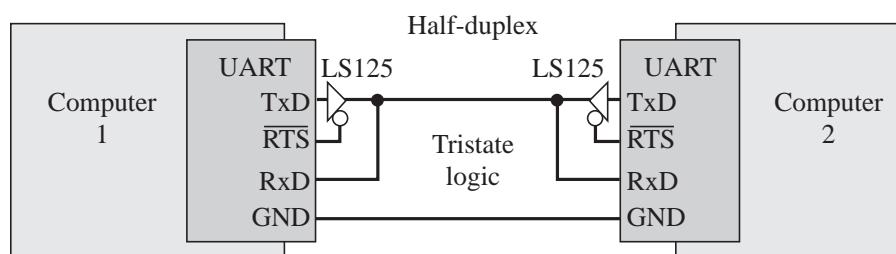
A *half-duplex communication system* allows information to transfer in both directions, but in only one direction at a time. Half-duplex is a term usually defined for modem communications, but in this book we will expand its meaning to include any serial protocol that allows communication in both directions, but in only one direction at a time. A fundamental problem with half-duplex is the detection and recovery from a collision. A collision occurs when both computers simultaneously transmit data. Fortunately, every transmission frame is echoed back into its own receiver. The transmitter program can output a frame, wait for the frame to be transmitted (which will be echoed into its own receiver), then check the incoming parity and compare the data to detect a collision. If a collision occurs, then it probably will be detected by both computers. After a collision, the transmitter can wait awhile and retransmit the frame. The two computers need to decide which one will transmit first after a collision so that a second collision can be avoided. The first hardware mechanism to implement half-duplex utilizes tristate logic. In this system, the transmitter will enable the driver ( $\overline{\text{RTS}} = 0$ ) before transmission, then disable the driver after complete transmission ( $\overline{\text{RTS}} = 1$ ).

**Observation:** People communicate in half-duplex.

When using the SCI (built into the 9S12), complete transmission occurs when the SCI **Receive Data Register Full** flag is set ( $\text{RDRF} = 1$ ) and not when the SCI **Transmit Data Register Empty** flag is set ( $\text{TDRE} = 1$ ). The SCI Transmit Complete Flag (TC) should be set approximately at the same time as the RDRF, because  $\text{TC} = 1$  means a frame has been shifted out, and  $\text{RDRF} = 1$  means a frame has been shifted in (Figure 7.5).

**Figure 7.5**

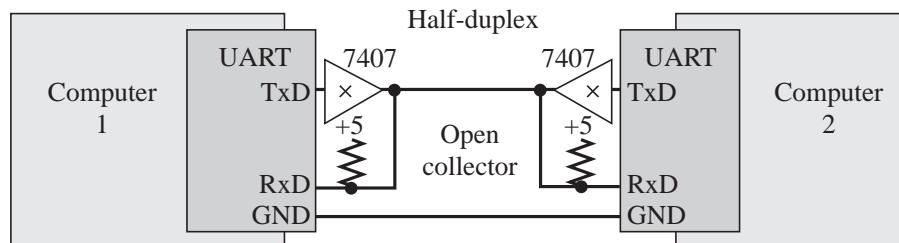
A half-duplex serial channel can be implemented with tristate logic.



Another hardware mechanism for half-duplex utilizes open-collector logic (Figure 7.6). The 7407 driver has two output states: zero and hiZ. The logic high is created with the passive pull-up. With open collector, the half-duplex channel is the logical AND of the two TxD outputs. In this system, the transmitter simply transmits its frame without needing to enable or disable the driver.

**Figure 7.6**

A half-duplex serial channel can be implemented with open-collector logic.

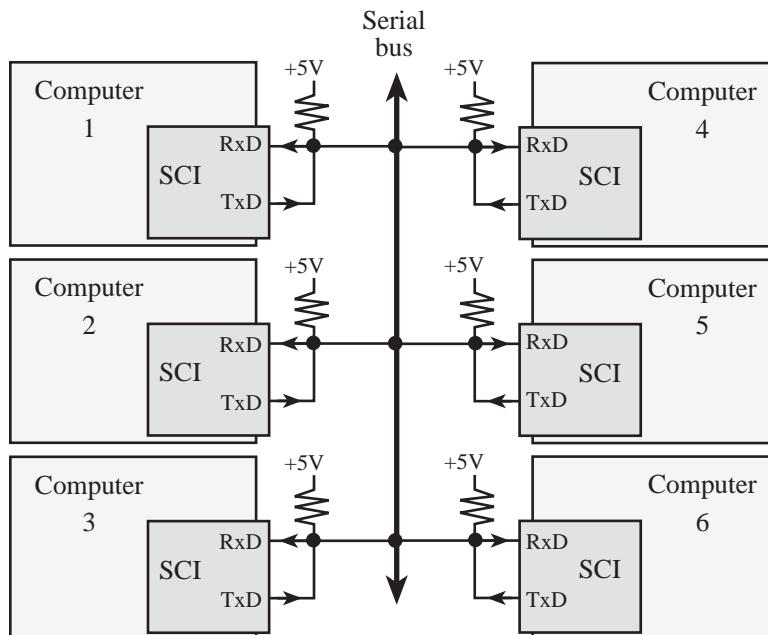


One application of the half-duplex protocol is the desktop serial bus, or multidrop network (Figure 7.7). In the following example, each microcomputer can send a message to another microcomputer. 9S12 serial port output can be made open-collector by setting WOMS=1. All the grounds are tied together. Assume there are less than 128 microcomputers in this network so that each microcomputer can have a unique 7-bit address. A frame with bit 7 equal to 1 is defined as an address; otherwise, the frame is data. The transmitting microcomputer first sends the address of the destination microcomputer, followed by multiple data frames. All the receiver microcomputers listen for their particular address. Once a microcomputer receiver recognizes its address, it will accept the data frames that follow. There are various options for determining the end of the message:

1. Define each message to be a fixed length
2. Send the message length as the second character
3. Define a special character that specifies end of message

**Figure 7.7**

A desktop network is created using a half-duplex serial channel implemented with open-collector logic.

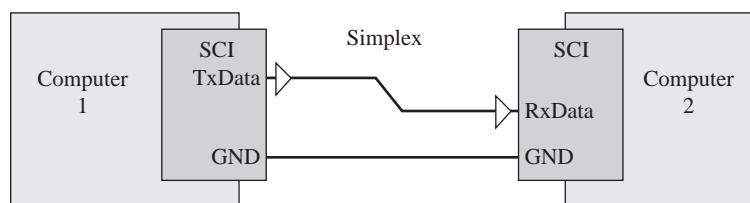


**Checkpoint 7.2:** What is the difference between full-duplex and half-duplex?

A *simplex communication* system allows information to transfer in only one direction. The XON/XOFF protocol that we will cover later is an example of a communication system that has a full-duplex channel but implements simplex communication. This is because with XON/XOFF information (characters) are transmitted from the computer to the printer, but only XON/XOFF (error checking) flags are sent from the printer back to the computer. In this case, no information (data, characters) is sent from the printer to the computer. Figure 7.8 shows a simplex serial channel.

**Figure 7.8**

A simplex serial channel between two computers.

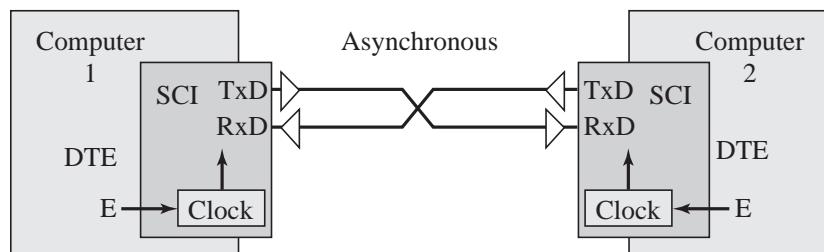


One application of simplex channels is the ring network. Rather than using a common bus like Figure 7.7, the six microcomputers could be connected in a ring topology using six simplex channels like Figure 7.8. The system could use the same address/data format described for the multidrop half-duplex connection. If a microcomputer receives a message addressed to a different node, it simply retransmits it along the ring. This system is slower than the multidrop system, but it does not have to contend with collision errors.

To transfer information correctly, both sides of the channel must operate at the same baud rate. In an *asynchronous* communication system, the two devices have separate and distinct clocks (Figure 7.9). Because these two clocks are generated separately (one on each side), they will not have exactly the same frequency or be in phase. If the two baud rate clocks have different frequencies, the phase between the clocks will also drift over time. Transmission will occur as long as the periods of the two baud rate clocks are close enough. The  $-12\text{ V}$  to  $+12\text{ V}$  edge at the beginning of the start bit is used to synchronize the receiver with the transmitter. If the two baud rate clock periods in a RS232 system differ by less than 5%, then after 10 bits the receiver will be off by less than half a bit time (and no error will occur). Any larger difference between the two periods will cause an error.

**Figure 7.9**

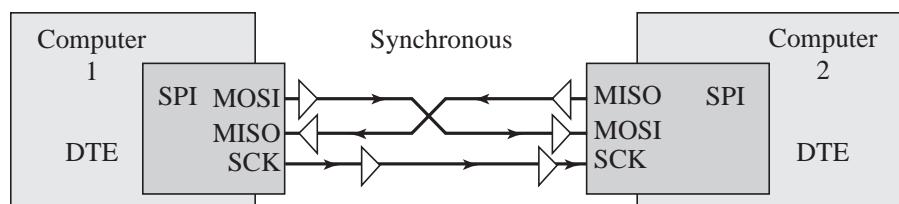
Nodes on an asynchronous channel run at the same frequency but have separate clocks.



In a *synchronous* communication system, the two devices share the same clock (Figure 7.10). Typically a separate wire in the serial cable carries the clock. In this way, very high baud rates can be obtained. Another advantage of synchronous communication is that

**Figure 7.10**

Nodes on a synchronous channel operate off a common clock.



very long frames can be transmitted. Larger frames reduce the OS overhead for long transmissions because fewer frames need be processed per message. Even though in this chapter we will design various low-bandwidth synchronous systems using the SPI, synchronous communication is best applied to systems that require bandwidths above 1 Mb/s. The cost of this increased performance is the additional wire in the cable. The clock must be interfaced with channel drivers (e.g., RS232, RS422, optocouplers) similar to the transmit and receiver data signals.

**Checkpoint 7.3:** What is the difference between synchronous and asynchronous serial?

## 7.2 RS232 Specifications

The baud rate in a RS232 system can be as high as 20,000 bits/s. Because a typical cable has 50 pF/ft, the maximum distance for RS232 transmission is limited to 50 ft. There are 21 signals defined for full modem (*M*Odulate/*D*EModulate) communication (Figure 7.11).

**Figure 7.11**

DB25 (RS232), DB9 (EIA-574), and RJ45 (EIA-561) connectors used in many serial applications.

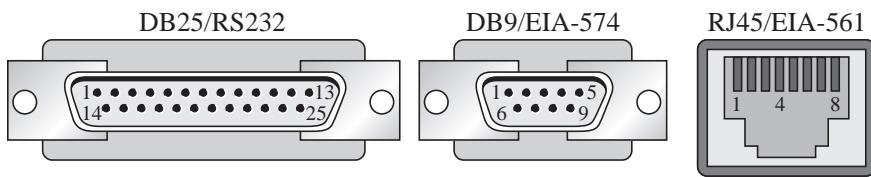


Table 7.2 shows the entire set of RS232 signals. The RS232 standard uses a DB25 connector that has 25 pins. The EIA-574 standard uses RS232 voltage levels and a DB9 connector that has only nine pins. The EIA-561 standard also uses RS232 voltage levels but

DB25 Pin	RS232 Name	DB9 Pin	EIA-574 Name	RJ45 Pin	EIA-561 Name	Signal	Description	True (V)	DTE	DCE
1						FG	Frame Ground/Shield			
2	BA	3	103	6	103	TxD	Transmit Data	-12	Out	In
3	BB	2	104	5	104	RxD	Receive Data	-12	In	Out
4	CA	7	105/133	8	105/133	RTS	Request to Send	+12	Out	In
5	CB	8	106	7	106	CTS	Clear to Send	+12	In	Out
6	CC	6	107			DSR	Data Set Ready	+12	In	Out
7	AB	5	102	4	102	SG	Signal Ground			
8	CF	1	109	2	109	DCD	Data Carrier Detect	+12	In	Out
9							Positive Test Voltage			
10							Negative Test Voltage			
11							Not Assigned			
12						sDCD	Secondary DCD	+12	In	Out
13						sCTS	Secondary CTS	+12	In	Out
14						sTxD	Secondary TxD	-12	Out	In
15	DB					TxC	Transmit Clk (DCE)		In	Out
16						sRxD	Secondary RxD	-12	In	Out
17	DD					RxC	Receive Clock		In	Out
18	LL						Local Loopback			
19						sRTS	Secondary RTS	+12	Out	In
20	CD	4	108	3	108	DTR	Data Terminal Rdy	+12	Out	In
21	RL					SQ	Signal Quality	+12	In	Out
22	CE	9	125	1	125	RI	Ring Indicator	+12	In	Out
23						SEL	Speed Selector DTE		In	Out
24	DA					TCK	Speed Selector DCE		Out	In
25	TM					TM	Test mode	+12	In	Out

**Table 7.2**

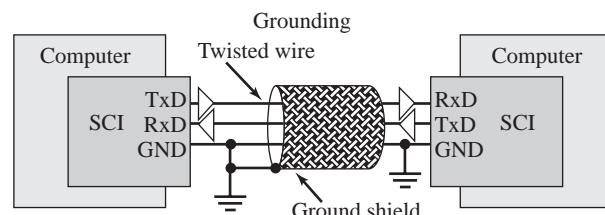
Pin assignments for the RS232 EIA-574 and EIA-561 protocols.

with a RJ45 connector that has only eight pins. The most commonly used signals of the full RS232 standard are available with the EIA-561/EIA-574 protocols.

The frame ground is connected on one side to the *ground shield* of the cable. The shield will provide protection from electric field interference. The *twisted cable* has a small area between the wires. The smaller the area, the less the magnetic field pickup. There is one disadvantage to reducing the area between the connectors. The capacitance to ground is inversely related to the separation distance between the wires. Thus as the area decreases, the capacitance will increase. This increased capacitive load will limit both the distance and the baud rate (Figure 7.12).

**Figure 7.12**

The simplest RS232 cable uses just TxD, RxD, and ground (with an optional ground shield).



The signal ground is connected on both sides to the supply return. A separate wire should be used for signal ground (do not use the ground shield to connect the two signal grounds). The noise immunity will be degraded if the ground shield is connected on both sides. There are many available RS232 driver chips, but the Maxim MAX232 (Figure 7.33) and DS275 (Figure 7.13) are popular devices because of their low cost and simple implementation. The MAX232 employs a charge pump (using 100-nF capacitors) to create the standard +12 and -12 output voltages with only a +5-V supply. The DS275 operates on +5-V and requires no external capacitors. Because the transmitter captures power from the receiver line, the DS275 is best employed in applications where data flows in only one direction at a time.

**Figure 7.13**

RS232 interface to Freescale microcomputers.

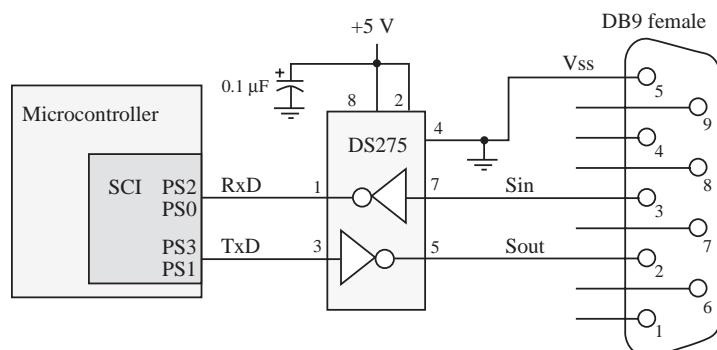


Figure 7.14 overviews the RS232 specifications for an output signal. Similarly, Figure 7.15 overviews the RS232 specifications for an input signal.

**Figure 7.14**

RS232 output specifications.

Must withstand

Short to ground

Short to any other wire

Operating range

True  $-15 \leq V_{out} \leq -5$  V

False  $+5 \leq V_{out} \leq +15$  V

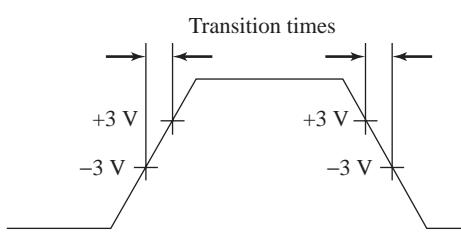
Maximum output voltage

$-25 \leq V_{out} \leq +25$  V

Short circuit current

$I_{out} \leq 0.5$  A

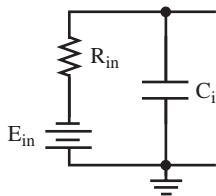
Transition (-3 to 3) range time  $\leq 4\%$



**Figure 7.15**

RS232 input specifications.

Maximum Slew Rate
$dV_{in}/dt \leq 30 \text{ V}/\mu\text{s}$
Operating Range
True $-15 \leq V_{in} \leq -3 \text{ V}$
Translation $-3 < V_{in} < +3 \text{ V}$
False $+3 \leq V_{in} \leq +15 \text{ V}$
Input Resistance
$3000 \Omega \leq R_{in} \leq 7000 \Omega$
Input Capacitance including cable
$C_{in} \leq 2500 \text{ pF}$
Input open circuit voltage
$E_{in} \leq 2 \text{ V}$



On the data lines, a true or mark voltage is negative. Conversely, for the control lines, a true signal has a positive voltage. *Request to Send (RTS)* is a signal from the computer to the modem requesting transmission be allowed. *Clear to Send (CTS)* is the acknowledge signal back from the modem signifying transmission can proceed. *Data Set Ready (DSR)* is a modem signal specifying that I/O can occur. *Data Carrier Detect (DCD)* is a modem signal specifying that the carrier frequencies have been established on its telephone line. *Ring Indicator (RI)* is a modem signal that is true when the phone rings. The *Receive Clock* and *Transmit Clock* are used to establish synchronous serial communication. *Data Terminal Ready (DTR)* is a printer signal specifying the status of the printer. This signal is sometimes called *Busy*. When DTR is +12 V, the printer is ready and can accept more characters. Conversely when DTR is -12 V, the printer is busy and cannot accept more characters. None of the built-in serial ports of the Freescale microcomputer discussed in this book supports these control lines explicitly. On the other hand, it is straightforward to implement these hardware handshaking signals using simple I/O lines. If we wish to generate interrupts on edges of these lines, then we would use input capture or key wake-up features.

A typical sequence for initiating communication between a computer (DTE) and a modem (DCE) begins with the turning on of the power in both devices. The computer activates the DTR line, and the modem responds by activating the DSR signal. Next, the modem establishes a link across the telephone line with the other modem. A functional link requires a separate carrier frequency from both modems. See Section 14.5 for additional details about the modem/modem link. The modem signals to the computer that a proper link has been established by activating DCD. When the computer wishes to transmit it, it activates RTS. If okay, the modem responds by activating CTS. After receiving the CTS, the computer can transmit data. The computer can continuously activate RTS if multiple frames are to be sent, but it must postpone transmission if CTS becomes false.

Faced with the design of a RS232 interface, we should use one of the many RS232 chips available. The MC1488/MC1489 require  $\pm 12\text{-V}$  supplies. Some devices (e.g., MAX232, MC145407) use a charge-pump mechanism so that they can operate on a single +5-V power supply; other devices (e.g., MC145406) have multiple bidirectional interfaces.

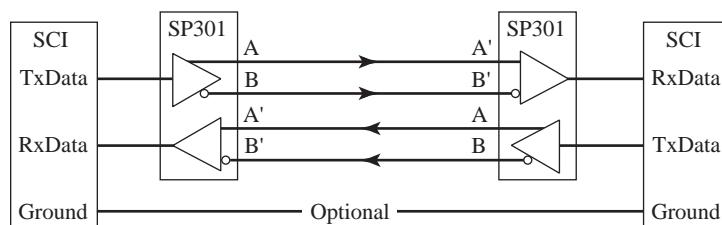
## 7.3 RS422/USB/RS423/RS485 Balanced Differential Lines

To increase the baud rate and maximum distance, the balanced differential line protocols were introduced. The RS422 signal is encoded in a differential signal, A-B. There are many RS422 interface chips, such as SP301, SP304, MAX486, MAX488, MC3486/3487, MC3691, MC3695, 8921/8922/8923, 75176, or 78120. A full-duplex RS422 channel is implemented in Figure 7.16 with Sipex SP301 drivers.

Because each signal requires two wires, five wires (ground included) are needed to implement a full-duplex channel. With RS232 one typically connects one receiver to one transmitter. But with RS422, up to ten receivers can be connected to one transmitter. Table 7.3 summarizes four common EIA standards.

**Figure 7.16**

RS422 serial channel.

**Table 7.3**

Specifications for the RS232, RS423A, RS422, and RS485 protocols.

Specification	RS232D	RS423A	RS422	RS485
Mode of operation	Single-ended	Single-ended	Differential	Differential
Drivers on one line	1	1	1	32
Receivers on one line	1	10	10	32
Maximum distance (ft)	50	4,000	4,000	4,000
Maximum data rate	20 kbits/sec	100 kbits/sec	10 Mbits/sec	10 Mbits/sec
Maximum driver output	$\pm 25$ V	$\pm 6$ V	$-0.25$ to $+6$ V	$-7$ to $+12$ V
Driver output (loaded)	$\pm 5$ V	$\pm 3.6$ V	$\pm 2$ V	$\pm 1.5$ V
Driver output (unloaded)	$\pm 15$ V	$\pm 6$ V	$\pm 5$ V	$\pm 5$ V
Driver load impedance	$3k\Omega$ to $7k\Omega$	$450\Omega$ min	$100\Omega$	$54\Omega$
Receiver input voltage	$\pm 15$ V	$\pm 12$ V	$\pm 7$ V	$-7$ to $+12$ V
Receiver input sensitivity	$\pm 3$ V	$\pm 200$ mV	$\pm 200$ mV	$\pm 200$ mV
Receiver input resistance	$3k\Omega$ to $7k\Omega$	$4k\Omega$ min	$4k\Omega$ min	$12k\Omega$ min

The maximum baud rate at 40 ft is 10 Mbits/sec. At 4000 ft, the baud rate can only be as high as 100 kbits/sec. Table 7.4 shows two implementations of the RS422 protocol.

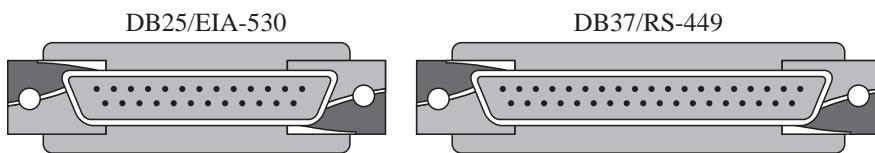
**Table 7.4**

Pin assignments for the EIA-530 and RS449 protocols.

DB25 Pin	EIA-530 Name	DB37 Pin	RS449 Name	Signal	Description	DTE	DCE
1		1		FG	Frame Ground/Shield		
2	BA (A)	4	SD (A)	TxD	Transmit Data	Out	In
14	BA (B)	22	SD (B)	TxD	Transmit Data	Out	In
3	BB (A)	6	RD (A)	RxD	Receive Data	In	Out
16	BB (B)	24	RD (B)	RxD	Receive Data	In	Out
4	CA (A)	7	RS (A)	RTS	Request to Send	Out	In
19	CA (B)	25	RS (B)	RTS	Request to Send	Out	In
5	CB (A)	9	CS (A)	CTS	Clear to Send	In	Out
13	CB (B)	27	CS (B)	CTS	Clear to Send	In	Out
6	CC (A)	11	DM (A)	DSR	Data Set Ready	In	Out
22	CC (B)	29	DM (B)	DSR	Data Set Ready	In	Out
20	CD (A)	12	TR (A)	DTR	Data Terminal Rdy	Out	In
23	CD (B)	30	TR (B)	DTR	Data Terminal Rdy	Out	In
7	AB	19	SG	SG	Signal Ground		
8	CF (A)	13	RR (A)	DCD	Data Carrier Detect	In	Out
10	CF (B)	31	RR (B)	DCD	Data Carrier Detect	In	Out
15	DB (A)	5	ST (A)	TxC	Transmit Clk (DCE)	In	Out
12	DB (B)	23	ST (B)	TxC	Transmit Clk (DCE)	In	Out
17	DD (A)	8	RT (A)	RxC	Receive Clock	In	Out
9	DD (B)	26	RT (B)	RxC	Receive Clock	In	Out
18	LL	10	LL		Local Loopback	Out	In
21	RL	14	RL		Remote Loopback	Out	In
24	DA (A)	17	TT (A)	TCK	Speed Selector DCE	Out	In
11	DA (B)	35	TT (B)	TCK	Speed Selector DCE	Out	In
25	TM	18	TM	TM	Test mode	In	Out

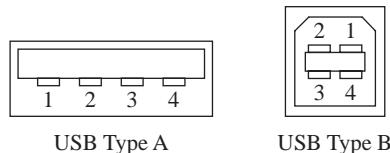
The EIA-530 standard uses a DB25 connector that has 25 pins. The RS449 standard uses a DB37 connector that has 37 pins (Figure 7.17).

**Figure 7.17**  
DB25 (EIA-530), and  
DB37 (RS-449)  
connectors used in  
RS422 applications.



In this section we will introduce the electrical specifications of the **Universal Serial Bus** (USB). Figure 7.18 shows the two types of USB connectors. A single host computer controls the USB, and there can only be one host per bus. The host controls the scheduling of all transactions using a token-based protocol. The USB architecture is a tiered star topology, similar to 10BaseT Ethernet. The host is at the center of the star and devices are attached to the host. The number of nodes on the bus can be extended using USB hubs. Up to 127 devices can be connected to any one USB bus at any one given time. USB plug ‘n’ plug is implemented with dynamically loadable and unloadable drivers. When the user plugs the device into the USB bus, the host will detect the connection, interact with the newly inserted device, and load the appropriate driver. The USB device can be used without explicitly installing drivers or rebooting. When the device is unplugged, the host will automatically unload its driver.

**Figure 7.18**  
USB connectors.



USB uses four shielded wires, +5 V power, GND, and twisted-pair differential data signals, as listed in Table 7.5. It uses a NRZI (non-return-to zero-invert) encoding scheme to send data with a sync field to synchronize the host and receiver clocks. The D+ signal has a 15 k $\Omega$  pull-down resistor to ground, and the D– signal has a 1.5 k $\Omega$  pull-up resistor to +3.6 V. Like the other protocols in this section, the data is encoded as a differential signal

**Table 7.5**  
USB signals.

Pin Number	Color	Function
1	Red	VBUS (5 V)
2	White	D–
3	Green	D+
4	Black	Ground

between D+ and D–. In general, a differential 1 exists when D+ is greater than D–. More specifically, a differential 1 is transmitted by pulling D+ over 2.8 V and D– under 0.3 V. The transmitter creates a differential 0 by making D– greater than 2.8 V and D+ less than 0.3 V. The receiver recognizes the differential 1 when D+ is 0.2 V greater than D–. The receiver will consider the input as a differential 0 when D+ 0.2 V less than D–. The polarity of the signal is inverted depending on the speed of the bus. Therefore the terms ‘J’ and ‘K’ states are used in signifying the logic levels. At low speed, a ‘J’ state is a differential 0. At high speed, a ‘J’ state is a differential 1. USB interfaces employ both differential and single-ended outputs. Certain bus states are indicated by single-ended signals on D+, D– or

both. For example, a single-ended zero (SE0) signifies device reset when held for more than 10mS. More specifically, SE0 is generated by holding both D<sub>-</sub> and D<sub>+</sub> low (< 0.3 V). USB can operate at three speeds. The low speed/full speed bus has a characteristic impedance of 90 Ω. High-speed mode uses a constant current protocol to reduce noise.

High-speed data is clocked at 480 Mb/s

Full-speed data is clocked at 12 Mb/s

Low-speed data is clocked at 1.5 Mb/s

### 7.3.1

## RS422 Output Specifications

The output voltage levels are shown in Table 7.6. A key RS422 specification is that the output impedances should be balanced. If the I/O impedances are balanced, then added noise in the cable creates a common-mode voltage, and the common-mode rejection of the input will eliminate it. More details about common mode are presented later in Chapters 11 and 12.

$$R_{A\text{out}} = R_{B\text{out}} \approx 100 \Omega$$

**Table 7.6**

Output voltage levels for the RS422 differential line protocol.

Output Voltage	
True or Mark	$-6 \leq A - B \leq -2 \text{ V}$
Transition	$-2 \leq A - B \leq +2 \text{ V}$
False or Space	$+2 \leq A - B \leq +6 \text{ V}$

The time in the transition region must be less than

10% for baud rates above 5 Mb/s

20 ns for baud rates below 5 Mb/s

### 7.3.2

## RS422 Input Specifications

The input voltage levels are as shown in Table 7.7.

**Table 7.7**

Input voltage thresholds for the RS422 differential line protocol.

Input Voltage	
True or Mark	$A - B \leq -0.2 \text{ V}$
Transition	$-0.2 \text{ V} \leq A - B \leq +0.2 \text{ V}$
False or Space	$+0.2 \text{ V} \leq A - B$

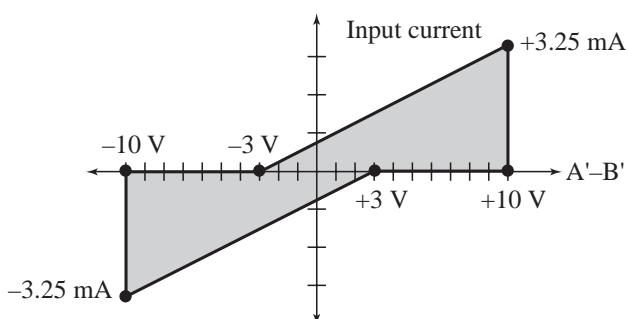
As mentioned earlier, to provide noise immunity the common-mode input impedances must also be balanced

$$4 \text{ k}\Omega \leq R_{A\text{in}} = R_{B\text{in}}$$

The balanced nature of the interface produces good noise immunity. The differential input impedance is specified by the plot in Figure 7.19. Any point within the shaded region is allowed. Even though the ground connection in the RS422 cable is optional, it is assumed the grounds are connected somewhere. In particular, the interface will operate with a common-mode voltage up to 7 V.

$$\frac{|A + B|}{2} \leq +7 \text{ V}$$

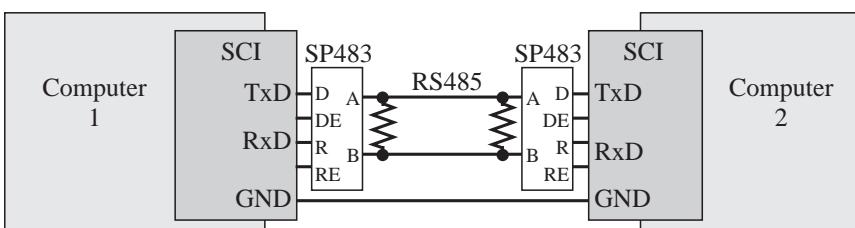
**Figure 7.19**  
RS422 input current versus input voltage relationship.



### 7.3.3 RS485 Half-Duplex Channel

RS485 can be either half-duplex or full-duplex. The RS485 protocol, illustrated in Figure 7.20, implements a half-duplex channel using differential voltage signals. The Sipex SP483 or Maxim MAX483 implements the half-duplex RS485 channel. One of the advantages of RS485 is that up to 32 devices can be connected onto a single serial bus. When more than one transmitter can drive the serial bus, the protocol is also called *multidrop*. To transmit the computer enables the driver by making DE active, then sends the serial frame from the TxD output of the SCI port. If RE is also active during transmission, the transmitted frame is echoed into the serial receiver of the SCI RxD line. To receive a frame the computer simply enables its receiver (by making RE active) and accepts a serial frame on the RxD line in the usual manner. Be careful when selecting the resistances on a half-duplex network so that the total driver impedance is about  $54\ \Omega$ .

**Figure 7.20**  
A half-duplex serial channel is implemented with RS485 logic.



RS422, RS485, Ethernet, and CAN (described in Chapter 14) are high-speed communication channels. This means the bandwidth and slew rate on the signals are higher than RS232. There is a correspondence between rise time ( $\tau$ ) of a digital signal and equivalent sinusoidal frequency ( $f$ ). One estimation of this relationship is

$$f = 1/\tau$$

For example, if the rise time is 5 ns, the equivalent frequency is 200 MHz. Notice that this equivalent frequency is independent of baud rate. So even at 1000 bits/sec, if the rise time is 5 ns, then the signal has a strong 200 MHz frequency component! To deal with this issue, the RS232 protocol limits the slew rate to a maximum of  $30V/\mu s$ . This means it will take about 1  $\mu s$  for a signal to rise from -12 to +12 V. Consequently, RS232 signals have frequency components less than 1 MHz. However, to transmit faster than RS232, the protocol must have faster rise times. Electrical signals travel at about 0.6 to 0.9 times the speed of light. For example, velocity factor for RG-6/U coax cable is 0.75, whereas the factor is only 0.66 for RG-58/U coax cable. Using the slower 0.66 estimate, the speed is  $v = 1.8 \cdot 10^8$  m/s. According to wave theory, the wavelength is  $\lambda = v/f$ . Estimating the frequency from rise time, we get

$$\lambda = v * \tau$$

In our example, a rise time of 5 ns is equivalent to a wavelength of about 1 m. As a rule of thumb, we will consider the channel as a *transmission line* if the length of the wire is greater

than  $\lambda/4$ . Another requirement is for the diameter of the wire to be much smaller than the wavelength. In a transmission line, the signals travel down the wires as waves according to the wave equation. Analysis of the wave equation is outside the scope of this book. However, you need to know that when a wave meets a change in impedance, some of the energy will transmit (a good thing) and some of the energy will reflect (a bad thing). Reflections are essentially noise on the signal, and if large enough they will cause bit errors in transmission. We can reduce the change in impedance by placing terminating resistors on both ends of a long high-speed cable, as shown in Figure 7.20. These resistors reduce reflections; hence, they improve signal to noise ratio.

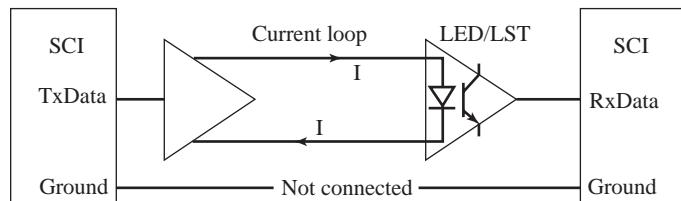
## 7.4 Other Communication Protocols

There are a variety of communication protocols for implementing communications channels. They vary in cost, distance, bandwidth, and noise immunity. Some protocols require the two computers to have a common ground, while others are isolated from each other. Isolation is important to prevent noise on one system from creating errors on another.

### 7.4.1 Current Loop Channel

*Current loop* is an old standard where true is encoded as a 20-mA current and false is signified by no current. The advantage of current loop is its inherent electrical isolation between the two computers. We could use current loop in applications where we wished to prevent noise in one computer from coupling into the electronics of the other (Figure 7.21).

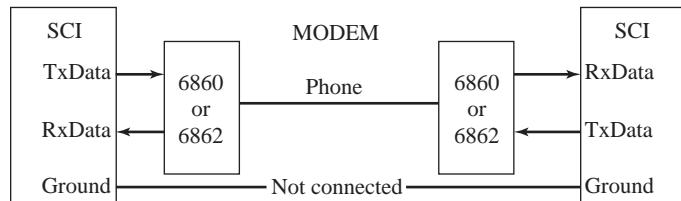
**Figure 7.21**  
Current loop serial interface.



### 7.4.2 Introduction to Modems

Detailed information about modems is presented later in Chapter 14. *Frequency-shift keying* (FSK) modems encode the binary data as frequencies that are transmitted as sounds on standard phone lines (Figure 7.22). The modulator converts the TxD data from the transmission computer into sounds. The phone line transmits the sounds to the receiver modem. The demodulator on the receiver converts the sound frequencies back into a regular true/false digital line. The SCI on the receiver accepts the serial frame in the usual way (start bit, data bits, stop bit).

**Figure 7.22**  
Modem serial interface.



Two pairs of frequencies implement full-duplex at 300 bits/s (Table 7.8).

**Table 7.8**

Frequency parameters for a 300 bits FSK modem protocol.

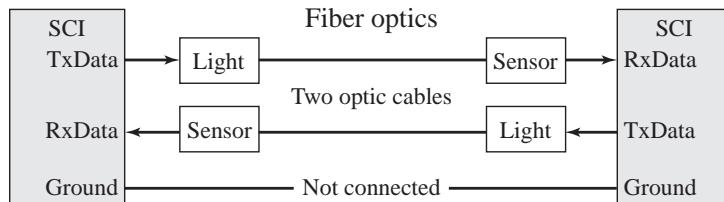
Logic	Originate	Answer
True	1270 Hz	2225 Hz
False	1070 Hz	2025 Hz

The *phase-encoded modem* also uses standard phone lines. Phase-encoded modems provide for increased bandwidth by encoding multiple bits into periodic phase shifts of the sound signal. More details can be found in Section 14.9. Even higher bandwidth is implemented by combining phase and amplitude information. A significant amount of digital signal processing is required to produce reliable transmission at high speeds.

### 7.4.3 Optical Channel

A *fiber-optic light* channel uses a LED transmitter, a fiber-optic cable, and a light sensor. Binary information is encoded as the presence or absence of light in the cable (Figure 7.23). Fiber-optic cable is used in applications requiring long distances and/or high bandwidth. Similar to the current loop channel, fiber-optic cables provide electrical isolation between the two computers. Techniques for interfacing LEDs can be found in Section 8.2. Another advantage of the optical channel is its noise immunity. All the other protocols are to some degree susceptible to electric field noise. Communication using fiber optics is not affected by electric fields along the cable.

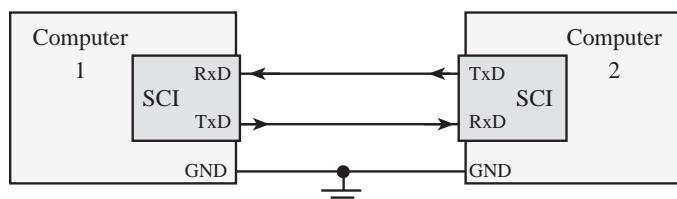
**Figure 7.23**  
Fiber-optic serial interface.



### 7.4.4 Digital Logic Channel

The simplest channel to implement uses standard *digital logic*. When the two computers are located in the same box, it is appropriate to send information directly from one to the other without special hardware circuits (Figure 7.24). Although standard digital logic will not be appropriate for long distances and/or in the presence of strong electric fields, it certainly is cheap and simple and should at least be considered for communication across short distances.

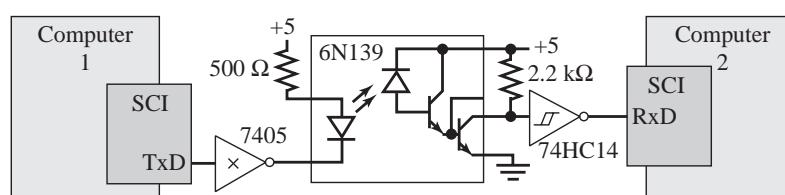
**Figure 7.24**  
Simple digital logic serial interface.



If isolation is required, then optoisolators like the 6N139 can be used (Figure 7.25).

**Observation:** When using an optocoupler such as the 6N139 in Figure 7.25, the power and ground signals are NOT connected between the two computers. This greatly reduces noise coupling.

**Figure 7.25**  
Isolated digital logic serial interface.



## 7.5 Serial Communications Interface

Most of the Freescale embedded microcomputers support at least one SCI. Before discussing the detailed operation of particular devices, we will begin with general features common to all devices. Common features that affect the entire SCI module include:

- A baud rate control register used to select the transmission rate
- A mode bit M used to select 8-bit (M=0) or 9-bit (M=1) data frames

Each device is capable of creating its own serial port clock with a period that is an integer multiple of the E clock period. The programmer will select the baud rate by specifying the integer divide-by used to convert the E clock into the serial port clock.

**Common error:** If you change the E clock frequency without changing the baud rate register, the SCI will operate at an incorrect baud rate.

Table 7.9 lists the I/O port pins for some 9S12 microcontrollers used for asynchronous serial transmission. 9S12 microcontrollers have one or two SCI devices.

**Table 7.9**

Serial port pins available on various Freescale 9S12 microcontrollers.

Microcomputer	SCI0 Pin for TxD	SCI0 Pin for RxD	SCI1 Pin for TxD	SCI1 Pin for RxD
MC9S12C	PS1	PS0	—	—
MC9S12DG	PS1	PS0	PS3	PS2
MC9S12DP	PS1	PS0	PS3	PS2
MC9S12XD	PS1	PS0	PS3	PS2
MC9S12E	PS1	PS0	PS3	PS2

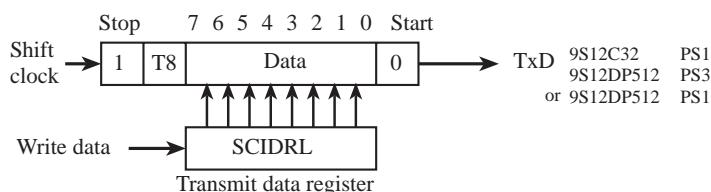
### 7.5.1 Transmitting in Asynchronous Mode

We will begin with transmission, because it is straightforward. Common features that affect the transmitter portion of the SCI include (Figure 7.26):

- TxD data output pin, with TTL voltage levels
- 10- or 11-bit shift register, which cannot be directly accessed by the programmer; this shift register is separate from the receive shift register
- Serial Communications Data Register (SCIDRL), which is write only, even though at the same address this data register is separate from the receive data register
- T8 data bit that you set before writing to SCIDRL when 9-bit data mode (M=1) is used

**Figure 7.26**

Data register and shift register used to implement the transmit serial interface.



The control bits that affect the transmitter are:

- Transmit Enable control bit (TE), which you initialize to 1 to enable the transmitter
- Send Break control bit (SBK), which you set to 1 to send blocks of 10 or 11 zeros
- Transmit Interrupt Enable control bit (TIE), which you set to 1 to arm the TDRE flag
- Transmit Complete Enable control bit (TCIE), which you set to 1 to arm the TC flag

The status bits generated by transmitter activity are:

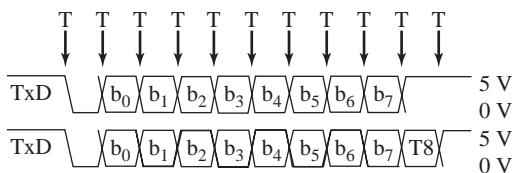
- Transmit Data Register Empty flag (TDRE), which is set when the transmit SCIDRL is empty; TDRE is cleared by reading the TDRE flag (with it set), then writing to the SCIDRL

Transmit Complete flag (TC), which is set when the transmit shift register is done shifting; TC is cleared by reading the TC flag (with it set), then writing to the SCIDRL

When new data (8 bits) are loaded in the SCIDRL, they are copied into the 10- or 11-bit transmit shift register. Next, the start bit, T8 (if M=1), and stop bits are added. Then, the frame is shifted out one bit at a time at a rate specified by the baud rate register. If there are already data in the shift register when the SCIDRL is written, it will wait until the previous frame is transmitted before it, too, is transferred. In the timing diagrams of Figures 7.27 and 7.28, the “T” arrows refer to times when the transmit shift register is shifted, causing a change in the TxD output pin.

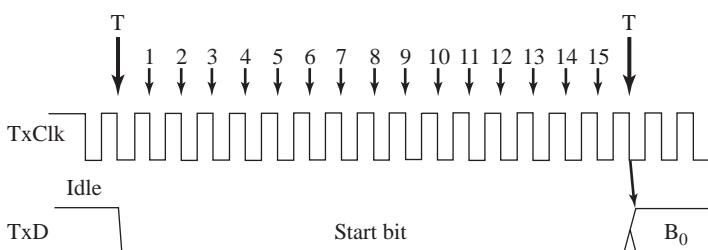
**Figure 7.27**

Transmit data frames for M=0 and M=1.



**Figure 7.28**

Start bit timing during a transmit data frame.



The serial port hardware is actually controlled by a clock that is 16 times faster than the baud rate. The digital hardware in the SCI counts 16 times in between changes to the TxD output line. Pseudocode for the transmission process is shown below (these operations occur automatically in hardware).

```

TRANSMIT Set TxD=0           Output start bit
          Wait 16 clock times   Wait 1 bit time
          Set n=0               Bit counter
TLOOP    Set TxD=bn         Output data bit
          Wait 16 clock times   Wait 1 bit time
          Set n=n+1
          Goto TLOOP if n≤7
          Set TxD=T8            Output T8 bit (optional if M=1)
          Wait 16 clock times   Wait 1 bit time (optional if M=1)
          Set TxD=1              Output a stop bit
          Wait 16 clock times   Wait 1 bit time

```

In essence, the SCIDRL and transmit shift register behave together like a two-element FIFO, with writing into the SCIDRL analogous to the PutFifo operation and the shifting data out analogous to the GetFifo operation. In fact, the serial port interface chip used in most PCs has a 16-byte hardware FIFO between the data register and the shift register. A PC that has a 16C550-compatible UART supports this hardware FIFO function. This FIFO reduces the latency requirements of the OS to service the serial port hardware.

## 7.5.2 Receiving in Asynchronous Mode

Receiving data frames is a little trickier than transmission because we have to synchronize the receive shift register with the incoming data. Common features that affect the receiver portion of the SCI include:

RxD data input pin, with TTL voltage levels

10- or 11-bit shift register, which cannot be directly accessed by the programmer; this shift register is separate from the transmit shift register

Serial Communications Data Register (SCIDRL), which is read only, even though at the same address this data register is separate from the transmit data register R8 bit that you can read after receiving a frame in 9-bit data mode ( $M=1$ )

The control bits that affect the receiver are:

Receiver Enable control bit (RE), which you initialized to 1 to enable the receiver  
Receiver Wakeup control bit (RWU), which you set to 1 to allow a receiver input to wake up the computer

Receiver Interrupt Enable control bit (RIE), which you set to 1 to arm the RDRF flag  
Idle Line Interrupt Enable control bit (ILIE), which you set to 1 to arm the IDLE flag

The status bits generated by receiver activity are (Figure 7.29):

Receive Data Register Full flag (RDRF), which is set when new input data is available; RDRF is cleared by reading the RDRF flag (with it set), then reading the SCIDRL

Receiver Idle flag (IDLE), which is set when the receiver line becomes idle; IDLE is cleared by reading the IDLE flag (with it set), then reading the SCIDRL

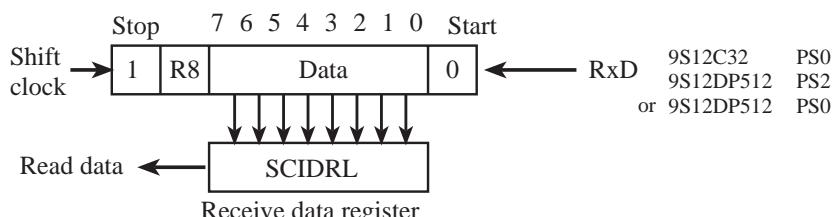
Overrun flag (OR), which is set when input data is lost because previous data frames had not been read; OR is cleared by reading the OR flag (with it set), then reading the SCIDRL

Noise flag (NF), which is set when the input is noisy; NF is cleared by reading the NF flag (with it set), then reading the SCIDRL

Framing Error (FE), which is set when the stop bit is incorrect; FE is cleared by reading the FE flag (with it set), then reading the SCIDRL

**Figure 7.29**

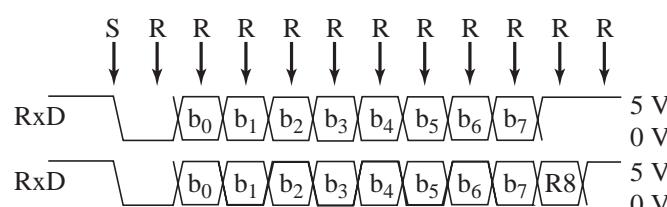
Data register and shift register used to implement the receive serial interface.



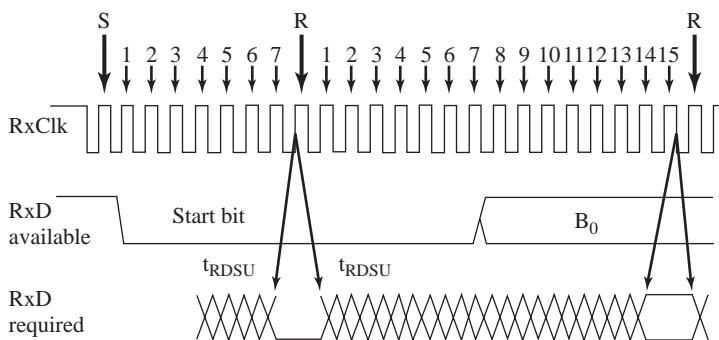
The receiver waits for the 1 to 0 edge signifying a start bit, then shifts in 10 or 11 bits of data one at a time from the RxD line. The start and stop bits are removed (checked for noise and framing errors), the 8 bits of data are loaded into the SCIDRL, the ninth data bit is put in R8 (if  $M=1$ ), and the RDRF flag is set. If there are already data in the SCIDRL when the shift register is finished, it will wait until the previous frame is read by the software before it is transferred. An overrun occurs when there is one receive frame in the SCIDRL, one receive frame in the receive shift register, and a third frame comes into RxD. In the timing diagrams of Figures 7.30 and 7.31, the "S" arrow refers to the time when the receiver detects the 1 to 0 edge of the start bit, and the "R" arrows refer to the times the receiver shift register is shifted, causing the current value of the RxD input pin to be recorded.

**Figure 7.30**

Receive data frames for  $M=0$  and  $M=1$ .



**Figure 7.31**  
Start bit timing during a receive data frame.



If “R” refers to the time the RxD pin is recorded, the setup time is the time before “R” the input must be valid. The hold time is the time after the “R” the data must continue to be valid. Pseudocode for the receive process is shown below (these operations occur automatically in hardware). Because the receiver must wait a half a bit time after the start 1 to 0 edge, it requires a clock faster than the baud rate. In particular, with a serial clock 16 times faster than the baud rate, it waits a half a bit time by waiting eight serial clock periods. All of the operations (e.g., waiting for RxD = 0) are synchronized to the serial clock.

```

RECEIVE Goto RECEIVE if RxD=1    Wait for start bit
        Wait 8 clock times      Wait half a bit time
        Goto RECEIVE if RxD=1  False start? (NF)
        Set n=0                 Bit counter
RLOOP   Wait 16 clock times     Wait 1 bit time
        Set bn=RXD           Input data bit
        Set n=n+1
        Goto RLOOP if n ≤ 7
        Wait 16 clock times   Wait 1 bit time (optional if M=1)
        Set R8= RxD            Read R8 bit (optional if M=1)
        Wait 16 clock times   Wait 1 bit time
        Set FE=1 if RxD = 0    Framing error if no stop bit
  
```

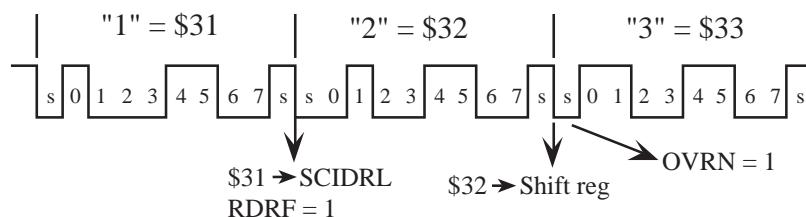
An overrun occurs when there is one receive frame in the SCIDRL, one receive frame in the receive shift register, and a third frame comes into RxD. To avoid overrun, we can design a real-time system (i.e., one with a maximum latency). The latency of a SCI receiver is the delay between the time when new data arrives in the receiver SCIDRL and the time the software reads the SCIDRL. If the latency is always less than 10 (11 if M=1) bit times, then overrun will never occur.

**Observation:** With a serial port that has a shift register and one data register (no additional FIFO buffering), the latency requirement of the input interface is the time it takes to transmit one data frame.

In the following example, assume the SCI receive shift register and receive data register are initially empty. Three incoming serial frames occur one right after another, but the software does not respond. At the end of the first frame, the \$31 goes into the receive SCIDRL and the RDRF flag is set. In this scenario, the software is busy doing other things and does not respond to the setting of RDRF. Next, the second frame is entered into the receive shift register. At the end of the second frame, there is the \$31 in the SCIDRL and the \$32 in the shift register. If the software were to respond at this point, then both characters would be properly received. If the third frame begins before the first is read by the software, then an overrun error occurs and a frame is lost (Figure 7.32). We can see from this worst-case scenario that the software must read the data from SCIDRL within 10 bit times of the setting of RDRF.

**Figure 7.32**

Three receive data frames result in an overrun (OR) error.



### 7.5.3 9S12 SCI Details

The MC9S12C family has one serial port called SCI using Port S bits 1,0. The MC9S12D family has two serial ports: SCI1 uses Port S bits 3,2 and SCI0 uses Port S bits 1,0. The SCI transmitter and receiver are independent, but use the same data format and bit rate. On the MC9S12DP512, the SCI port names include a 0 or 1 to specify which SCI module; otherwise, the SCI modules on the various 9S12 microcontrollers operate similarly. For example, the one baud rate register on the MC9S12C32 is called **SCIBD**, but on the MC9S12DP512, there are two SCI modules, so there are two baud rate registers called **SCI0BD** and **SCI1BD**. Table 7.10 shows the I/O ports that implement the SCI functions. It has a RS232 Non-Return-to-Zero (NRZ) format with one start bit, eight or nine data bits, and one stop bit, as shown in Figure 7.27. The SCI transmitter and receiver are independent, but use the same data format and bit rate.

**Table 7.10**  
9S12 SCI ports.

Address	msb	lsb	Name
\$00C8	- - - 1 2 11 10 98 7 6 5 4 3 2 1 0		SCIBD

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$00CA	LOOPS	SWAI	RSRC	M	WAKE	ILT	PE	PT	SCICR1
\$00CB	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCICR2
\$00CC	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCISR1
\$00CD	0	0	0	0	0	BRK13	TXDIR	RAF	SCISR2
\$00CE	R8	T8	0	0	0	0	0	0	SCIDRH
\$00CF	R7T7	R6T6	R5T5	R4T4	R3T3	R2T2	R1T1	ROTO	SCIDRL

The least significant 13 bits of **SCIBD** determine the baud rate for the SCI port. If **BR** is the value written to bits 12:0, and **Mclk** is the module clock (typically this is the same as the E clock), then the baud rate is:

$$\text{SCI Baud Rate} = \frac{\text{Mclk}}{16 * \text{BR}}$$

The **SCICR2** control register contains the bits that turn on the SCI, and contains the interrupt arm bits. **TE** is the Transmitter Enable bit, and **RE** is the Receiver Enable bit. We set both TE and RE equal to 1 in order to activate the SCI device. **TIE** is the Transmit Interrupt Enable bit. We set TIE=1 to arm the transmitter so that an interrupt is requested when TDRE is set. We clear TIE=0 to disarm the TDRE-triggered interrupts. **TCIE** is the Transmit Complete Interrupt Enable bit. We set TCIE=1 to arm the transmitter so that an interrupt is requested when TC is set. We clear TCIE=0 to disarm the TC-triggered interrupts. **RIE** is the Receiver Interrupt Enable bit. We set RIE=1 to arm the receiver so that an interrupt is requested when RDRF is set. We clear RIE=0 to disarm the RDRF-triggered interrupts. **ILIE** is the Idle Line Interrupt Enable bit. We set ILIE=1 to arm the receiver so that an interrupt is requested when IDLE is set. We clear ILIE=0 to disarm the IDLE-triggered interrupts. **RWU** is the Receiver Wake Up Control bit. We set RWU=1 to enable wake up and inhibit receiver interrupts. **SBK** is the Send Break bit. We set SBK=1 to send a break (continuous TxD low) as long as the SBK is 1.

The **SCICR1** control register contains the bits that handle special modes of the SCI. **LOOPS** is the SCI LOOP Mode/Single Wire Mode Enable bit. We set it to 1 to enable loop mode. When loop mode is active, the SCI receive section is disconnected from the RxD pin and the RxD pin is available as general purpose I/O. The receiver input is determined by the RSRC bit. The transmitter output is controlled by the associated DDRS bit. Both the transmitter and the receiver must be enabled to use the LOOP or the single wire mode. **RSRC** is the Receiver Source when LOOPS=1, the RSRC bit determines the internal feedback path for the receiver. If RSRC equals 0, the receiver input is connected to the transmitter internally (not TxD pin). If RSRC is 1, then the receiver input is connected to the TxD pin. **M** is the Mode bit. We set M=0 to create a 10 bit frame with 1 start bit, 8 data bits, 1 stop bit. We set M=1 to create an 11-bit frame with 1 start bit, 9 data bits, 1 stop bit (the ninth data bit is in T8/R8). If **Wake**=0, then the SCI will wake up by an IDLE line recognition. If Wake=1, then the SCI will wake up by address mark (most significant data bit set). **ILT** is the Idle Line Type specifying which of two types of idle line detection will be used by the SCI receiver. ILT determines when the receiver starts counting logic 1s as idle character bits. The counting begins either after the start bit or after the stop bit. If the count begins after the start bit, then a string of logic 1s preceding the stop bit may cause false recognition of an idle character. Beginning the count after the stop bit avoids false idle character recognition, but requires properly synchronized transmissions. If ILT is 1, then the idle character bit count begins after the stop bit. If ILT is 0, then the idle character bit count begins after start bit. To enable parity we set **PE** to 1. If parity is enabled, the SCI will insert a parity bit into the most significant position, and we specify the parity type with **PT**. If PT is 0, then an even number of ones in the data character causes the parity bit to be zero, and an odd number of ones causes the parity bit to be one. If PT is 1, then odd parity is selected. An odd number of ones in the data character causes the parity bit to be zero and an even number of ones causes the parity bit to be one. If parity is enabled, the receiver will test the parity of each incoming frame. Typically, we set M=1 along with PE=1 to create an 11-bit frame (1 start, 8 data, 1 parity, and 1 stop). Alternatively, we can set M=0 and with PE=1 to create a 10-bit frame (1 start, 7 data, 1 parity, and 1 stop).

The flags in the **SCISR1** register can be read by the software, but cannot be modified by writing to this register. **TDRE** is the Transmit Data Register Empty Flag. It is set by the SCI hardware if transmit data can be written to SCIDRL. If TDRE is zero, transmit data register contains previous data that has not yet been moved to the transmit shift register. Writing into the SCIDRL when TDRE is set will result in a loss of data. On the other hand, when this bit is set, the software can begin another output transmission by writing to SCIDRL. This flag is cleared by first reading SCISR1 with TDRE set followed by a SCIDRL write. **TC** is the Transmit Complete Flag. It is set if transmitter is idle (no data, preamble, or break transmission in progress). We can clear TC by reading SCISR1 with TC set followed by writing to SCIDRL. **RDRF** is the Receive Data Register Full bit. RDRF is set if a received character is ready to be read from SCIDR. We clear the RDRF flag by reading SCISR1 with RDRF set followed by reading SCIDRL. **IDLE** is the Idle Line Detected Flag. It is set if the RxD line is idle (10 or 11 consecutive logic ones). The IDLE flag is inhibited when RWU is set to one. It is cleared by reading SCISR1 with IDLE set followed by reading SCIDRL. Once cleared, IDLE is not set again until the RxD line has been active and becomes idle again.

Four error conditions can occur during generation of SCI input/output. Four bits (OR, NF, FE and PE) in the serial communications status register (SCISR1) indicate whether one of these error conditions exists. The overrun error (**OR**) bit is set when the next byte is ready to be transferred from the receive shift register to the SCDR and the SCDR is already full (RDRF bit is set), see Figure 7.32. When an overrun error occurs, the data that caused the overrun is lost and the data that was already in SCIDRL is not disturbed. The OR is cleared when the SCISR1 is read (with OR set), followed by a read of the SCIDRL. The noise flag (**NF**) bit is set if there is noise on any of the received bits, including the start and stop bits. In particular, each data bit is sampled three times and the NF bit is set if the three samples are not all the same. The NF bit is not set until the RDRF flag is set. The NF bit is cleared when the SCISR1

is read (with FE equal to 1) followed by a read of the SCIDRL. When no stop bit is detected in the received data character, the framing error (**FE**) bit is set. FE is set at the same time as the RDRF. If the byte received causes both framing and overrun errors, the processor only recognizes the overrun error. The framing error flag inhibits further transfer of data into the SCIDRL until it is cleared. The FE bit is cleared when the SCISR1 is read (with FE equal to 1) followed by a read of the SCIDRL. The parity flag (**PF**) bit is set when the parity enable bit (PE) is set and the parity of the received data does not match the parity type bit (PT). The PF bit is set during the same cycle as the RDRF flag but does not get set in the case of an overrun. We can clear PF by reading SCISR1 and then reading SCIDRL.

The **SCIDRL** register contains the data transmitted out and received in by the SCI device. Even though there are separate transmit and receive data registers, these two registers exist at the same I/O port address. Reads to SCIDRL access the eight bits of the read-only SCI receive data register. Writes to SCIDRL access the eight bits of the write-only SCI transmit data register. **R8** is Receive Data Bit 8. It is a read-only bit that contains the ninth bit of the receive data when M=1. **T8** is Transmit Data Bit 8. If M bit is set, T8 stores ninth bit in transmit data character. T8 can be set to 1 to implement a second stop bit. When using 9-bit data, it is necessary to read and write the SCI data register as one 16-bit register.

**Common error:** If we read or write the SCIDRL and SCIDRH registers in two separate 8-bit accesses, it is possible to confuse the MSbyte and the LSbyte between sequential frames.

The **SCISR1** register contains two mode control bits and one status bit. **BRK13** is the Break Transmit character length bit, which determines the transmit break length. If BRK13 is 1, then the break character is 13 or 14 bit long, and if it is 0, then the break Character is 10 or 11 bit long. **TXDIR** specifies the transmitter pin data direction in single-wire mode. It determines whether the TxD pin is going to be used as an input or output, in the single-wire mode of operation. If TXDIR is 1, then the TxD pin is an output in Single-Wire mode. If it is 0, then the TxD pin is an input in Single-Wire mode. **RAF** is the Receiver Active Flag. This flag is read only and is controlled by the receiver front end. It is set during the detection of a start bit. It is cleared when an idle state is detected or when the receiver circuitry detects a false start bit (generally due to noise or baud rate mismatch). If RAF is one, then a frame is being received. It is useful in half-duplex systems to avoid a collision.

**Observation:** The advances in RS232 interface hardware have reduced the probability of transmission errors to such an extent that most serial channels no longer implement parity checking.

**Observation:** There is no simple way for the transmitter to know whether the baud rates are not matched.

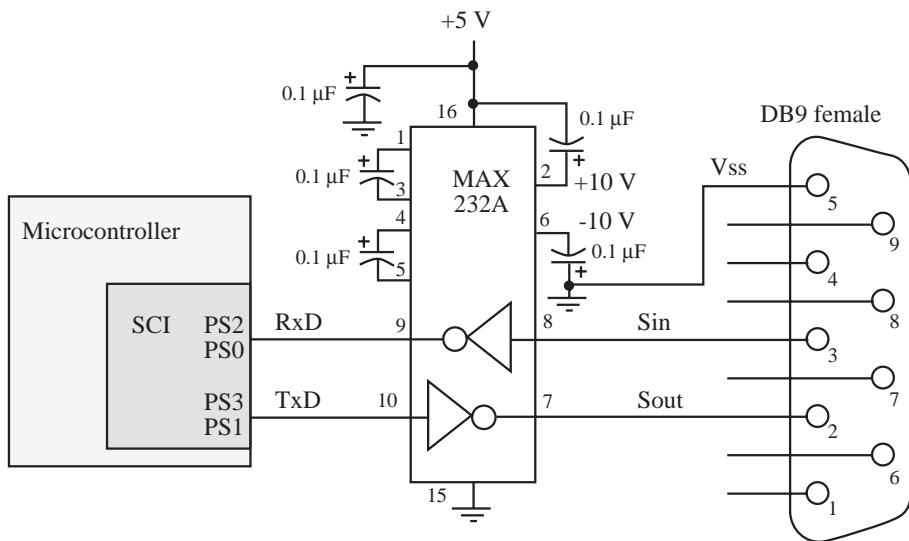
**Checkpoint 7.4:** In what simple way can the receiver software tell whether the baud rates are not matched?

**Example 7.1** Design a full-duplex serial port driver using interrupt synchronization.

**Solution** The objective of this section is to develop software to support bidirectional data transfer using interrupt synchronization. We could connect the microcontroller to another computer and use this channel to transfer data. For example, if we connect the DB9 cable to a serial port on a PC, we could run HyperTerminal on the PC and communicate with the microcontroller. A Maxim converter chip is used to generate the RS232 voltage levels, as

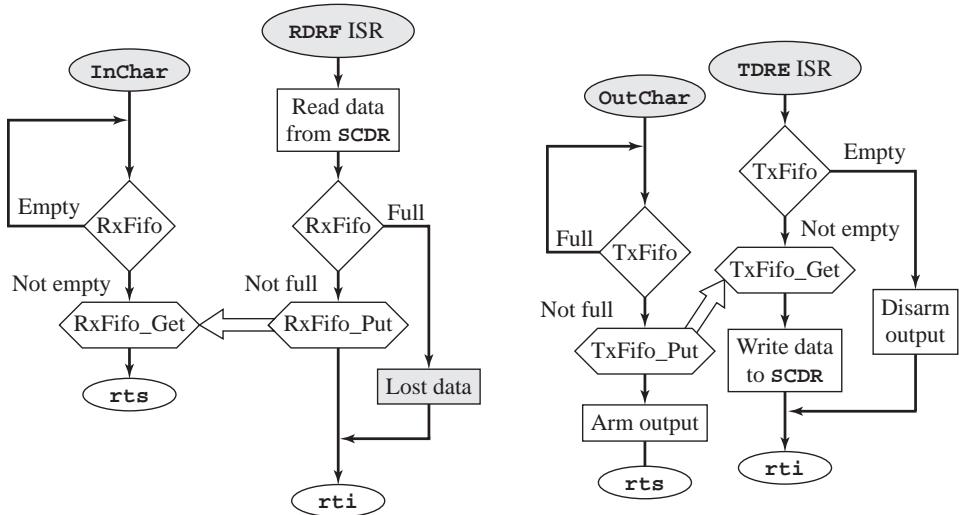
shown in Figure 7.33. The RS232 timing is generated automatically by the SCI. A simple device driver for this interface was developed in Chapter 3, and in this section we will redesign the device driver to use interrupts.

**Figure 7.33**  
Hardware interface  
implementing an  
asynchronous RS232  
channel.



The data flow graph for this interface was shown previously as Figure 4.20. This example can be found as `tut4` within **TExaS**. In order for data to be properly received, the baud rate must match with the other module, which in this interface will be 9600 bits/sec. Initially, the two FIFOs are cleared, and just the receiver is armed. The transmitter will be armed when data is available within the `SCI_OutChar` routine. A flowchart of this system is show in Figure 7.34. An interrupt occurs when new incoming data arrives in the receiver data register (`RDRF=1`). An interrupt also occurs when the transmit data register is empty (`TDRE=1`). `TDRE` is one, when the output channel is idle, needing the software to supply additional data. Notice that the transmit channel is disarmed when the `TxFifo` is empty and rearmed when new data is put into the `TxFifo`. When the `RxFifo` becomes full, then data is lost, but when the `TxFifo` becomes full, the main program simply waits for space to become available. The implementation of the FIFO was shown previously as Program 4.14. The software implementation of this system is presented as Program 7.1.

**Figure 7.34**  
FIFO queues can be used  
to pass data between  
threads.



```

SCI_Init jsr RxFifo_Init ;FIFO is empty
          jsr TxFifo_Init ;FIFO is empty
          movb #$2c,SCICR2 ;arm just RDRF
          movw #26,SCIBD   ;baud rate=9600
          cli
          rts
; Inputs: none Outputs: RegA is ASCII
SCI_InChar leas -1,s
          tsx           ;X->place
iloop  jsr RxFifo_Get ;A=0 if empty
          tbeq A,iloop
          pul a         ;A=character
          rts
; Inputs: RegA is ASCII Outputs: none
SCI_OutChar psha
oloop  ldaa 0,s      ;A=character
          jsr TxFifo_Put ;save in FIFO
          tbeq A,oloop
          movb #$AC,SCICR2 ;arm TDRE
          pul a
          rts
SCIhandler ldaa SCISR1
          bita #$20
          beq CkTDRE    ;Not RDRF set
          ldaa SCIDRL  ;ASCII character
          bsr RxFifo_Put
CkTDRE  ldaa SCISR1
          bpl sdone     ;Not TDRE set
          ldaa SCICR2  ;bit 7 is TIE
          bpl sdone     ;disarmed?
          leas -1,s
          tsx           ;X->place
          bsr TxFifo_Get
          pul b
          tbeq A,nomore
          stab SCIDRL  ;start output
          bra sdone
nomore  movb #$2C,SCICR2 ;disarm TDRE
sdone   rti
          org $FFD6
fdb    SCIhandler

```

```

void SCI_Init(void){
  RxFifo_Init(); // empty FIFOs
  TxFifo_Init();
  SCIBD = 26;    // 9600 bits/sec
  SCICR1 = 0;    // M=0, no parity
  SCICR2 = 0x2C; // enable, arm RDRF
asm cli        // enable interrupts
}
// Input ASCII character from SCI
// spin if RxFifo is empty
char SCI_InChar(void){ char letter;
  while(RxFifo_Get(&letter) == 0){};
  return(letter);
}
// Output ASCII character to SCI
// spin if TxFifo is full
void SCI_OutChar(char data){
  while(TxFifo_Put(data) == 0){};
  SCICR2 = 0xAC; // arm TDRE
}
// RDRF set on new receive data
// TDRE set on empty XMT register
interrupt 20 void SCIhandler(void){
char data;
  if(SCISR1&RDRF){
    RxFifo_Put(SCIDRL); // clears RDRF
  }
  if((SCICR2&0x80)&&(SCISR1&TDRE)){
    if(TxFifo_Get(&data)){
      SCIDRL = data; // clears TDRE
    }
    else{
      SCICR2 = 0x2c; // disarm TDRE
    }
  }
}

```

### Program 7.1

Software implementation of an interrupting SCI interface.

**Checkpoint 7.5:** How does the software clear the RDRF flag?

**Checkpoint 7.6:** How does the software clear the TDRE flag?

**Observation:** Data is lost when the RxFifo gets full.

**Common error:** Notice that the foregoing transmit device driver either acknowledges the interrupt by sending another character or disarms itself because the TxFifo is empty. The software will crash (infinite loop) if it returns from interrupt without acknowledging or disarming.

**Checkpoint 7.7:** Why didn't the initialization software arm TDRE?

**Checkpoint 7.8:** What bad thing would happen if the RDRF ISR waited for there to be room in the RxFifo, just as SCI\_OutChar waits for there to be room in the TxFifo?

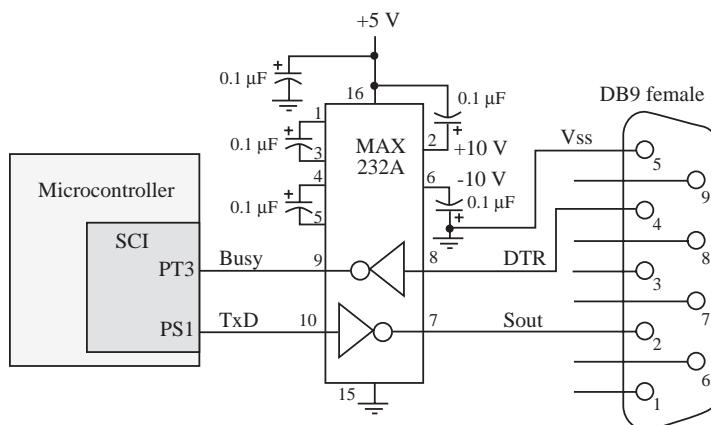
**Checkpoint 7.9:** Modify Program 7.1 so the baud rate is 1200 bits/sec.

**Example 7.2** Design a simplex printer driver with DTR synchronization.

**Solution** One problem with printers is that the printer bandwidth (the actual number of characters per second that can be printed) may be less than the maximum bandwidth supported by the serial channel. There are five conditions that might lead to a situation where the computer outputs serial data to the printer, but the printer isn't ready to accept the data. First, special characters may require more time to print (e.g., carriage return, line feed, tab, formfeed, and graphics). Second, most printers have internal FIFOs that could get full. If the FIFO is not full, then it can accept data as fast as the channel will allow, but when the FIFO becomes full, the computer should stop sending data. Third, the printer cable may be disconnected. Fourth, the printer may be deselected. Fifth, the printer power may be off. The output interfaces shown previously provide no feedback from the printer that could be used to detect and correct these five problems. There are two mechanisms, called flow control, to synchronize the computer with a variable rate output device. These two flow control protocols are called DTR and XON/XOFF.

The first method uses a hardware signal, DTR (pin 4 on the DB9 connector or pin 20 on the DB25 connector), as feedback from the printer to the microcomputer (see Figure 7.35). DTR is  $-12\text{ V}$  if the printer is busy and is not currently able to accept transmission. DTR is  $+12\text{ V}$  if the printer is ready and able to accept transmission. This mechanism can handle all five of the above situations. The computer input mechanism will handle the DTR protocol using additional software checking. With a standard RS232 interface when DTR is  $-12\text{ V}$ , the input line will be  $+5\text{ V}$  (which will stop the transmission if the TDR register is empty). Thus, when DTR is  $-12\text{ V}$ , transmission is temporarily suspended. When DTR is  $+12\text{ V}$ , the input line will be  $0\text{V}$  and transmission can proceed normally. In this design, input capture will be used to detect changes in the DTR signal.

**Figure 7.35**  
Hardware interface  
implementing a RS232  
simplex channel with  
DTR handshaking.



The DTR signal from the printer provides feedback information about the printer status. When DTR is  $-12\text{ V}$ , (PT3 is high), the printer is not ready to accept more data. In this case, our computer will postpone transmitting more frames. When DTR is  $+12\text{ V}$  (PT3 is low), the printer is ready to accept more data. At this point, the computer will resume transmission. The `Printer_OutChar(data)`, shown in Program 7.2, is called by the main program when it wishes to print.

```

Printer_Init
    jsr TxFifo_Init ;empty
    movw #26,SCIBD ;9600 bits/sec
    movb #0,SCICR1 ;M=0, no parity
    movb #$0C,SCICR2 ;disarm TDRE
    bclr TIOS,#$08 ;PT3 input capture
    bclr DDRT,#$08 ;PT3 is input
    movb #$80,TSCR1 ;enable TCNT
    bset TCTL4,#$C0 ;both rise fall
    bset TIE,#$08 ;Arm IC3
    movb #$08,TFLG1 ;initially clear
    cli ;enable
    rts
checkIC3
    brclr PTT,#$08,not ;0 if ready
    movb #$0C,SCICR2 ;busy, disarm
    bra done
not movb #$8C,SCICR2 ;ready, arm
done rts
Printer_OutChar ;A=character
    psha
loop ldaa 0,s ;A=character
    jsr TxFifo_Put ;save in FIFO
    tbeq A,loop ;A=0 if full
    bsr checkIC3
    pula
    rts
SCIhandler
    ldaa SCISR1
    bpl fini ;broken if no TDRE
    ldaa SCICR2 ;bit 7 is TIE
    bpl fini ;disarmed?
    leas -1,s
    tsx ;X->place
    bsr TxFifo_Get
    pulb
    tbeq A,none ;A=status
    stab SCIDRL ;start output
    bra fini
none movb #$2C,SCICR2 ;disarm TDRE
fini rti
IC3Han movb #$08,TFLG1 ;Ack
    bsr checkIC3 ;Arm if ready
    rti
    org $FFD6
    fdb SCIhandler
    org $FFE8
    fdb IC3Han

```

```

void Printer_Init(void){
    TxFifo_Init(); // empty FIFOs
    SCIBD = 26; // 9600 bits/sec
    SCICR1 = 0; // M=0, no parity
    SCICR2 = 0x0C; // enable disarm TDRE
    TIOS &=~0x08; // PT3 input capture
    DDRT &=~0x08; // PT3 is input
    TSCR1 = 0x80; // enable TCNT
    TCTL4 |= 0xC0; // both rise and fall
    TIE |= 0x08; // Arm IC3
    TFLG1 = 0x08; // initially clear
    asm cli // enable interrupts
}
void checkIC3(void){
    if(PTT&0x08) // PT3=1 if DTR=-12
        SCICR2 = 0x0C; // busy, so disarm
    else
        SCICR2 = 0x8C; // not busy, so arm
}
// Output ASCII character to Printer
// spin if TxFifo is full
void Printer_OutChar(char data){
    while(TxFifo_Put(data) == 0){};
    checkIC3();
}
// TDRE set on empty transmit register
interrupt 20 void SCIhandler(void){
char data;
    if((SCICR2&0x80)&&(SCISR1&TDRE)){
        if(TxFifo_Get(&data)){
            SCIDRL = data; // clears TDRE
        }
        else{
            SCICR2 = 0x2C; // disarm TDRE
        }
    }
}
// IC3 interrupt on any change of DTR
void interrupt 11 IC3Han(void) {
    TFLG1 = 0x08; // Ack, clear C3F
    checkIC3(); // Arm SCI if DTR=+12
}

```

### Program 7.2

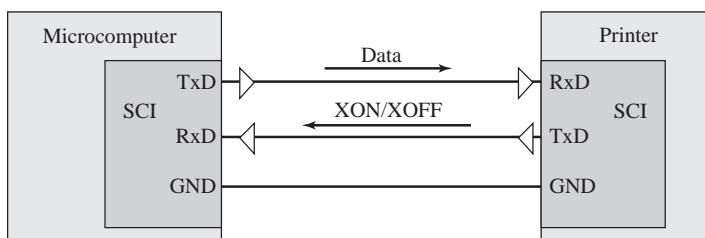
Software implementation of a printer interface with DTR synchronization.

### Example 7.3 Design a simplex printer driver with XON/XOFF synchronization.

**Solution** Another flow control technique is the XON/XOFF protocol. The hardware, which is a standard full-duplex serial channel, is described in Figure 7.36. This technique allows the printer to signal back to the computer that it cannot currently accept more data.

**Figure 7.36**

Hardware interface implementing a serial channel with XON/XOFF handshaking.

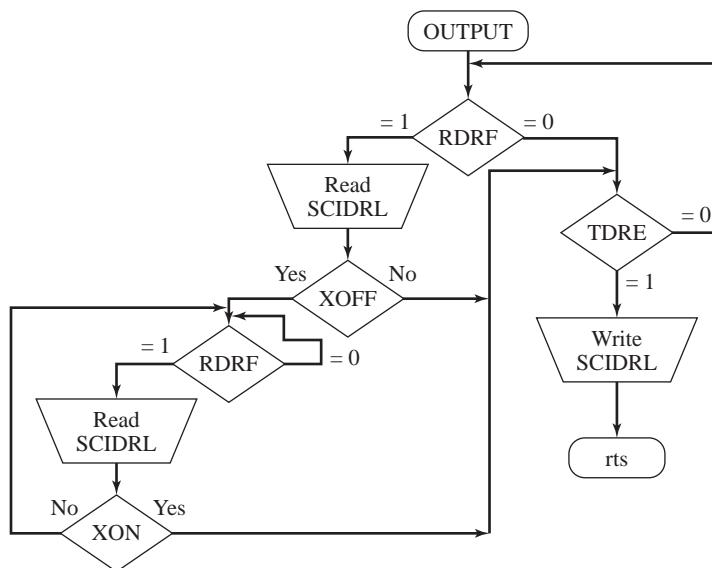


The advantage of XON/XOFF is that it uses a standard full-duplex serial channel. This is an example of a simplex communication *system* constructed on top of a full-duplex *channel*.

In this system we must assume the printer power is on, and the cable is properly connected. Most printers maintain a FIFO between the incoming serial data from the computer, and the actual printing hardware. When the printer FIFO is almost full (e.g., 80% full) or paper out, the printer will send a single XOFF (\$13) transmission back to the computer. Once the FIFO has more room (e.g., 20% full) or more paper, the printer will send a single XON (\$11) to the computer. Typically the computer assumes the printer FIFO is empty, so it can initially begin transmission without first receiving an XON. But, once the computer receives an XOFF, it must wait for an XON before it continues transmission. One disadvantage of XON/XOFF is that it cannot handle cable disconnection or printer power-off problems. Figure 7.37 shows a flowchart to implement the XON/XOFF protocol using busy-wait synchronization.

**Figure 7.37**

Software flowchart implementing a serial channel with XON/XOFF handshaking.



## 7.6 Synchronous Transmission and Receiving Using the SPI

### 7.6.1 SPI Fundamentals

Many of the Freescale embedded microcomputers include a SPI. The fundamental difference between a SCI, which implements an asynchronous protocol, and a SPI, which implements a synchronous protocol, is the manner in which the clock is implemented. Two devices communicating with asynchronous serial interfaces (SCI) operate at the same frequency (baud rate) but have two separate (not synchronized) clocks. Two devices communicating with synchronous serial interfaces (SPI) operate from the same clock (synchronized). Typically, the master device creates the clock, and the slave device(s) uses the clock to latch the data in or out. Before discussing the detailed operation of particular devices, we will begin with general features common to all devices. The Freescale SPI includes four

I/O lines. The slave select ( $\overline{SS}$ ) is an optional negative logic control signal from master to slave signifying the channel is active. The second line, SCK, is a 50% duty cycle clock generated by the master. The master-out slave-in (MOSI) is a data line driven by the master and received by the slave. The master-in slave-out (MISO) is a data line driven by the slave and received by the master. To work properly, the transmitting device uses one edge of the clock to change its output, and the receiving device uses the other edge to accept the data. Table 7.11 lists the I/O port locations of the synchronous serial ports for the various microcomputers discussed in this book.

**Table 7.11**

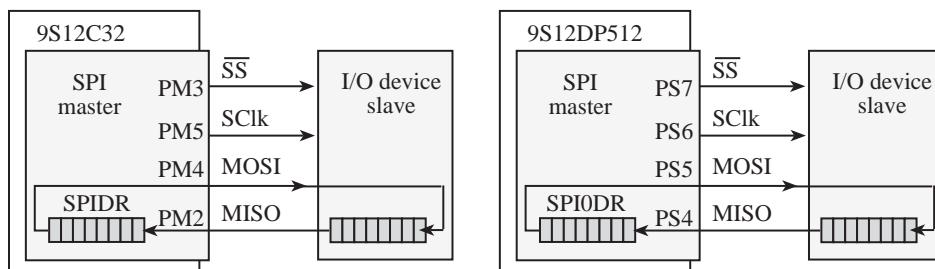
Synchronous serial port pins on various Freescale microcomputers.

Microcomputer	Pin for $\overline{SS}$	Pin for SCK	Pin for MOSI	Pin for MISO
9S12C32	PM3	PM5	PM4	PM2
9S12DP512 SPI0	PS7	PS6	PS5	PS4
9S12DP512 SPI1	PH3	PH2	PH1	PH0
9S12DP512 SPI2	PH7	PH6	PH5	PH4

The SPI allows the computer to communicate synchronously with peripheral devices and other microprocessors. The SPI system in the microcomputer can operate as a master or as a slave. In the SPI system the 8-bit data register, SPIDR, in the master and the 8-bit data register in the slave are linked to form a distributed 16-bit register. Figure 7.38 illustrates communication between master and slave. Typically, the microcontroller and the I/O device slave are so physically close that we do not use interface logic.

**Figure 7.38**

A synchronous serial interface between a microcomputer and an I/O device.



When a data transfer operation is performed, this 16-bit register is serially shifted eight bit positions by the SCK clock from the master so that the data are effectively exchanged between the master and the slave. Data written to the SPIDR register of the master are transmitted to the slave. Data in the slave register are transmitted to the master. The SPI is also capable of interprocessor communications in a multiple master system.

Common control features for the SPI module include:

- A baud rate control register used to select the transmission rate
- A mode bits in the control register to select master versus slave, clock polarity, clock phase
- Interrupt arm bit
- Ability to make the outputs open-drain (open-collector)

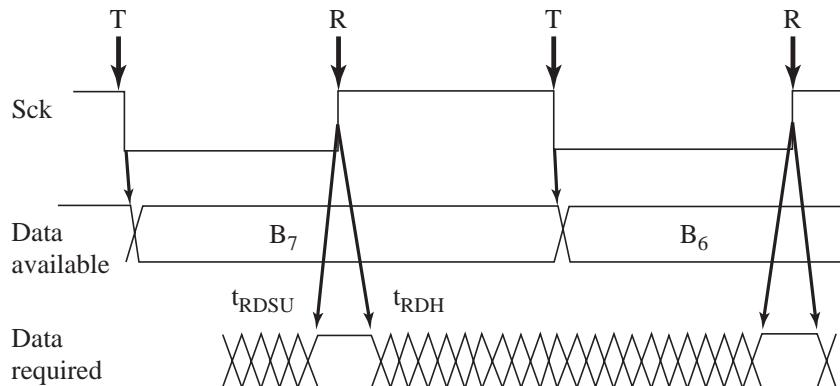
Common status bits for the SPI module include:

- SPIF, transmission complete
- WCOL, write collision
- MODF, mode fault

**Observation:** Because the clocks are shared, if you change the E clock frequency, the transfer rate will change, but the SPI still should operate properly.

The key to proper transmission is to select one edge of the clock (shown as T in Figure 7.39) to be used by the transmitter to change the output, and use the other edge (shown as R) to latch the data in the receiver. In this way data are latched during the time when they are stable. Data available is the time when the output data is actually valid, and data required is the time when the input data must be valid. For the communication to occur without error, the data available from the device that is driving the data line must overlap (start before and end after) the data required by the other device that is receiving the data. It is this overlap that will determine the maximum frequency at which synchronous serial communication can occur. The concepts of data available and data required were presented in Sections 3.14 and 3.15.

**Figure 7.39**  
Synchronous serial timing showing that the data available interval overlaps the data required interval.



The general idea of SPI communication can be illustrated by the following pseudocodes. Although it is possible to implement synchronous serial transmission on any computer with simple I/O pins in software using the bit-banging approach, these operations are implemented in hardware by the SPI interface.

TRANSMIT	Set n=7	Bit counter
TLOOP	On the fall of Sck, set Data=b <sub>n</sub>	Output data bit
	Set n=n-1	
	Goto TLOOP if n≥0	
	set Data=1	Idle output
RECEIVE	Set n=7	Bit counter
RLOOP	On the rise of Sck, read Data	
	Set b <sub>n</sub> =Data	Input data bit
	Set n=n-1	
	Goto RLOOP if n≥0	

**Observation:** Because the clocks are shared, if you change the E clock frequency, the transfer rate will change.

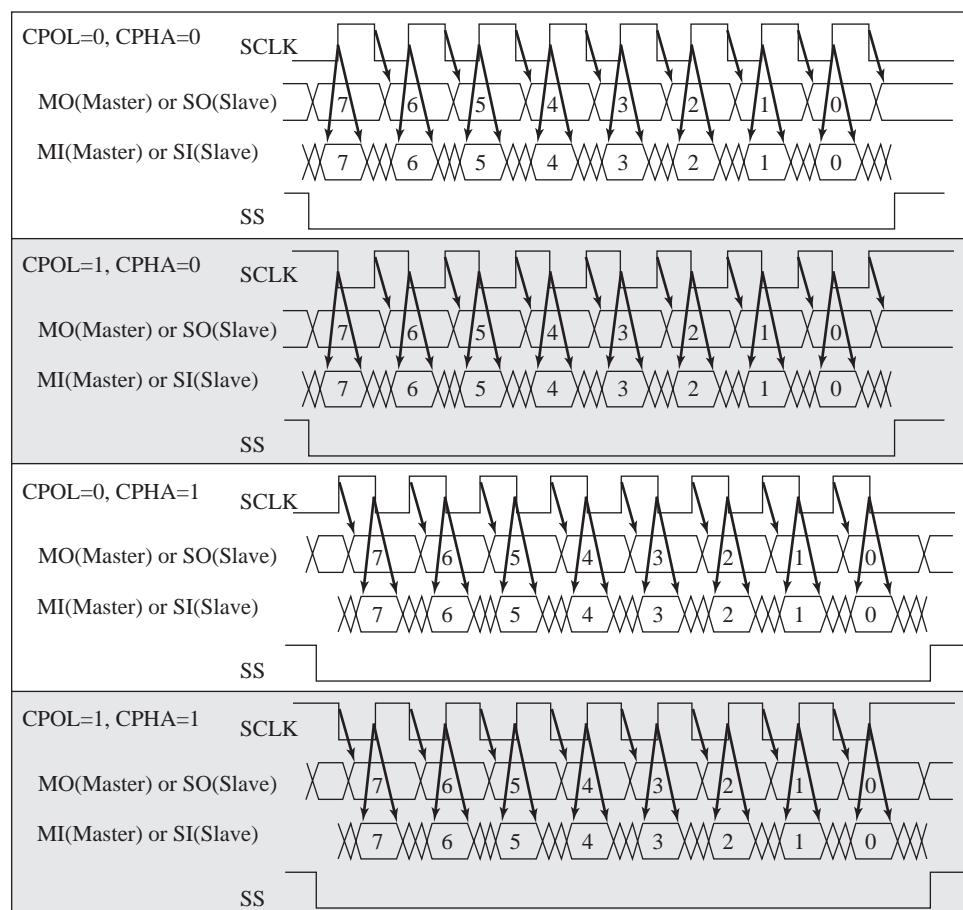
The SPI timing is shown in Figure 7.40. The SPI transmits 8-bit data at the same time as it receives input. In all modes, the SPI changes its output on the opposite edge of the clock as it uses to shift data in. There are three mode control bits (MSTR, CPOL, CPHA) that affect the transmission protocol. If the device is a master (**MSTR**=1), it generates the **SCLK**; data is output on the **MOSI** pin and input on the **MISO** pin. If the device is a slave (**MSTR**=0), the **SCLK** is an input, and data is received on the **MOSI** pin and transmitted on the **MISO** pin. The **CPOL** control bit specifies the polarity of the **SCLK**. In particular, the **CPOL** bit specifies the logic level of the clock when data is not being transferred. The **CPHA** bit affects the timing of the first bit transferred and received. If **CPHA** is 0, then the device will shift data in on the first (and 3rd, 5th, 7th, . . . etc.) clock edge. If **CPHA** is 1, then the device will shift data in on the second (and 4th, 6th, 8th, . . . etc.) clock edge.

In Figure 7.40, the data is shown with MSB transferred first, but the 9S12 has an option where the bits are transferred in the other order (LSB first.)

Next we will overview the specific SPI functions on the MC9S12C32. This section is intended to supplement rather than replace the Freescale manuals. When designing systems with a SPI, it is important to refer to the reference manual of your specific Freescale microcomputer.

**Figure 7.40**

Synchronous serial modes of the Freescale SPI interface.



## 7.6.2 MC9S12C32 SPI Details

The SPI port on the MC9S12C32 uses the four pins PM3= $\overline{SS}$ , PM5=SCLK, PM4=MOSI, and PM2=MISO (see Table 7.11). When the SPI is enabled (**SPE=1**), all pins that are defined by the configuration as inputs will be inputs regardless of the state of the DDRM bits for those pins. All pins that are defined as SPI outputs will be outputs only if the DDRM bits for those pins are set. If the 9S12 is the master, then we should set the DDRM register to make PM5, PM4, and PM3 outputs. PM2 will automatically be an input. A bidirectional serial pin is possible using the **BIDIROE** as the direction control. Table 7.12 shows the MC9S12C32 ports.

The SPI functions in three modes: run, wait, and stop. **Run mode** is the basic mode of operation. The SPI operation in **wait mode** is a configurable low-power mode, controlled by the **SPISWAI** bit. In wait mode, if the SPISWAI bit is clear, the SPI operates as in Run Mode. If the SPISWAI bit is set, the SPI goes into a power-conservative state, with the SPI clock generation turned off. If the SPI is configured as a master, any transmission in progress stops but is resumed after CPU goes into Run Mode. If the SPI is configured as a slave, reception and transmission of a byte continues, so that the slave stays synchronized

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$00D8	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBF	SPICR1
\$00D9	0	0	0	MODFEN	BIDIROE	0	SPISWAI	SPC0	SPICR2
\$00DA	0	0	0	0	0	SPR2	SPR1	SPR0	SPIBR
\$00DB	SPIF	0	SPTEF	MODF	0	0	0	0	SPISR
\$00DD	Bit 7	6	5	4	3	2	1	Bit 0	SPIDR
\$0250	0	0	PM5	PM4	PM3	PM2	PM1	PM0	PTM
\$0252	0	0	DDRM5	DDRM4	DDRM3	DDRM2	DDRM1	DDRM0	DDRM

**Table 7.12**  
MC9S12C32 SPI ports.

to the master. The SPI is inactive in **stop mode** for reduced power consumption. If the SPI is configured as a master, any transmission in progress stops, but is resumed after CPU goes into Run Mode. If the SPI is configured as a slave, reception and transmission of a byte continues, so that the slave stays synchronized to the master.

The module clock (same frequency as the E clock) is input to a divider series and the resulting SPI clock rate may be selected to be divided by 2, 4, 8, 16, 32, 64, 128, or 256. Three bits in the SPIBR register control the SPI clock rate. The SPIBR register determines the transfer rate. Table 7.13 shows the possible transmission rates for two different module clocks.

**Table 7.13**  
Bit rate selection for the synchronous serial port on the 9S12.

SPR2	SPR1	SPR0	Divisor	Module Clock=4 MHz		Module Clock=24 MHz	
				Frequency	Bit Time	Frequency	Bit Time
0	0	0	2	2 MHz	500 ns	12 MHz	83.3 ns
0	0	1	4	1 MHz	1 µs	6 MHz	166.7 ns
0	1	0	8	500 kHz	2 µs	3 MHz	333.3 ns
0	1	1	16	250 kHz	4 µs	1.5 MHz	666.7 ns
1	0	0	32	125 kHz	8 µs	750 kHz	1.33 µs
1	0	1	64	62.5 kHz	16 µs	375 kHz	2.67 µs
1	1	0	128	31.25 kHz	32 µs	187.5 kHz	5.33 µs
1	1	1	256	15.625 kHz	64 µs	93.75 kHz	10.67 µs

We use the **SPICR1** register to specify the SPI mode of operations. **SPE** is the SPI System Enable bit. We will turn this bit on whenever we wish to use the SPI. We set the **SSOE** bit to enable the  $\overline{SS}$  signal as shown in Figure 7.40. We clear the **SSOE** bit when we want to use PM3 as a regular I/O pin. The **SPIE** bit is the arm bit for SPIF, and **SPTIE** bit is the arm bit for SPTEF. These arm bits will be cleared because interrupts are not needed. We will set the **MSTR** to one so that the 9S12 becomes the master. **CPOL** and **CPHA** determine the SPI Clock Polarity and Clock Phase. These two bits are used to specify the protocol, as shown in Figure 7.40. All other bits not specifically mentioned will be cleared. **SPIF** is the SPI Interrupt Request bit. Even though we won't be using interrupts, this bit gets set after each byte is transferred, and will be used to implement the busy-waiting synchronization. In particular, SPIF is set after the eighth SCK cycle in a data transfer, and it is cleared by reading the **SPISR** register (with SPIF set) followed by an access (read or write) to the SPI data register. **SPTEF** is the Transmit Empty Interrupt Flag. If set, this bit indicates that the transmit data register is empty. To clear this bit and place data into the transmit data register, SPISR has to be read with SPTEF=1, followed by a write to SPIDR. Any write to the SPI Data Register without reading SPTEF=1, is effectively ignored.

The **SPIDR** 8-bit register is both the input and output register. A write to SPIDR allows a data byte to be queued and transmitted. For a SPI configured as a master, a queued data byte is transmitted immediately after the previous transmission has completed. The **SPTEF**

in the SPISR register indicates when the SPI Data Register is ready to accept new data. Reading the data can occur anytime from after the SPIF is set to before the end of the next transfer. If the SPIF is not serviced by the end of the successive transfers, those data bytes are lost, and the data within the SPIDR retains the first byte until SPIF is serviced. **LSBFE** is the LSB-First Enable bit. This bit does not affect the position of the MSB and LSB in the data register. Reads and writes of the data register always have the MSB in bit 7. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state. If LSBFE is 1, data is transferred least significant bit first. If LSBFE is 0, data is transferred most significant bit first.

The control bits **MODFEN** and **SSOE** affect the operation of the PM3 pin as defined in Table 7.14. **MODF** is the Mode Error Interrupt Status Flag. This bit is set if the  $\overline{SS}$  input becomes low when the SPI is configured as a master and mode fault detection is enabled, MODFEN bit of SPICR2 register is set. The flag is cleared automatically by a read of the SPI Status Register (with MODF set) followed by a write to the SPICR1.

**Table 7.14**

Mode selection for the synchronous serial port on the MC9S12C32.

<b>MODFEN</b>	<b>SSOE</b>	<b>Master Mode (MSTR=1)</b>	<b>Slave Mode (MSTR=0)</b>
0	0	PM3 not used with SPI	PM3 is $\overline{SS}$ input
0	1	PM3 not used with SPI	PM3 is $\overline{SS}$ input
1	0	PM3 is $\overline{SS}$ input with MODF feature	PM3 is $\overline{SS}$ input
1	1	PM3 is $\overline{SS}$ output	PM3 is $\overline{SS}$ input

Bidirectional modes are controlled by the bits **SPC0**, **BIDIROE**, and **MSTR**. These control bits determine the input/output configuration of the PM2 and PM4 as illustrated in Table 7.15 and Figure 7.41.

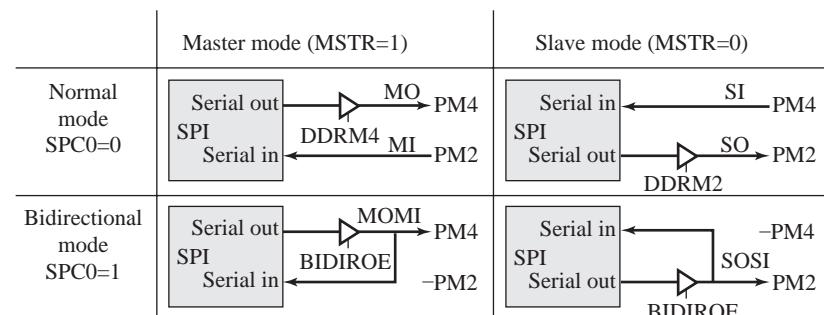
**Table 7.15**

Bidirectional modes for the SPI on the MC9S12C32.

<b>Pin Mode</b>	<b>MSTR</b>	<b>SPC0</b>	<b>BIDIROE</b>	<b>MISO</b>	<b>MOSI</b>
Normal	1	0	X	Master In	Master Out
Bidirectional	1	1	0	MISO Not used	Master In
Bidirectional	1	1	1	Not used	Master I/O
Normal	0	0	X	Slave Out	Slave In
Bidirectional	0	1	0	Slave In	MOSI Not used
Bidirectional	0	1	1	Slave I/O	Not used

**Figure 7.41**

Synchronous serial modes of the Freescale 9S12 SPI interface.



**Checkpoint 7.10:** How does the software clear the SPIF flag?

**Checkpoint 7.11:** What hardware interface and which modes do we use to create a bidirectional communication channel between two microcontrollers using their SPI ports?

In general, it takes four software steps in the master to perform a single 8-bit SPI communication. Even if we just want to input or just want to output, we need to perform all four steps:

1. Wait for **SPTEF** (e.g. read **SPISR** with **SPTEF** set)
2. Write data to **SPIDR** (this starts the SPI transfer)
3. Wait for **SPIF** (e.g., read **SPISR** with **SPIF** set) and
4. Read data from **SPIDR**.

### 7.6.3 9S12DP512 Module Routing Register

One of the confusing aspects of 9S12 family is its module routing register. On the 9S12DP512, the **MODRR** register allows the software to dynamically configure which port pins are used for the CAN0, CAN4, SPI0, SPI1, and SPI2 ports (see Table 7.16). Table 1.16 back in Chapter 1 defines a left to right priority. For example, if both CAN4 and I<sup>2</sup>C are enabled and mapped to PJ7-6, then the PJ7-6 will be CAN4 (not I<sup>2</sup>C), because it is higher priority (more to the left in Table 1.16). In general, the digital I/O is lowest priority and can be used only if all its special I/O functions are disabled.

CAN4 Routing				CAN0 Routing			
MODRR3	MODRR2	RXCAN4	TXCAN4	MODRR1	MODRR0	RXCAN0	TXCAN0
0	0	PJ6	PJ7	0	0	PM0	PM1
0	1	PM4 <sup>a</sup>	PM5 <sup>a</sup>	0	1	PM2 <sup>c</sup>	PM3 <sup>c</sup>
1	0	PM6 <sup>b</sup>	PM7 <sup>b</sup>	1	0	PM4 <sup>d</sup>	PM5 <sup>d</sup>
1	1	Reserved		1	1	PJ6	PJ7

<sup>a</sup>Routing of CAN4 to PM4 and PM5 only if CAN2 is disabled, and CAN0 not active or not routed here

<sup>b</sup>Routing of CAN4 to PM6 and PM7 only if CAN3 is disabled

<sup>c</sup>Routing of CAN0 to PM2 and PM3 only if CAN1 is disabled

<sup>d</sup>Routing of CAN0 to PM4 and PM5 only if CAN2 is disabled

SPI2 Routing					SPI1 Routing					SPI0 Routing				
MODRR6	MISO	MOSI	SCK	SS	MODRR5	MISO	MOSI	SCK	SS	MODRR4	MISO	MOSI	SCK	SS
0	PP4	PP5	PP7	PP6	0	PP0	PP1	PP2	PP3	0	PS4	PS5	PS6	PS7
1	PH4	PH5	PH6	PH7	1	PH0	PH1	PH2	PH3	1	PM2	PM4	PM5	PM3

**Table 7.16**

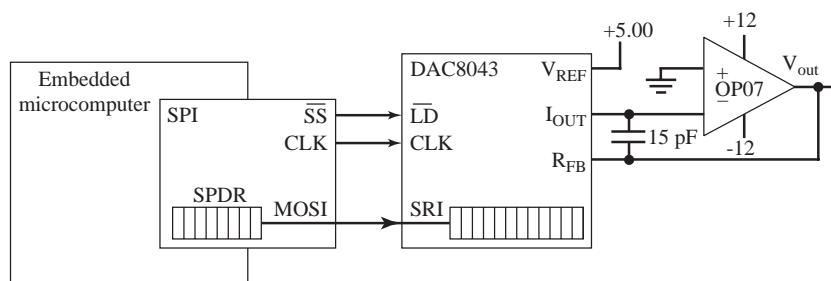
The MODRR register on the 9S12DP512 defines which pins implement CAN0, CAN4, SPI0, SPI1, and SPI2.

**Example 7.4** Interface the Analog Devices DAC8043 digital-to-analog converter to the 9S12.

**Solution** This first example shows the synchronous serial interface between the computer and an Analog Devices DAC8043 DAC. A DAC accepts a digital input (in our case a number between 0 and 4095) and creates an analog output (in our case a voltage between 0 and –5). Detailed discussion of DACs will be presented later in Chapter 11. Here in this section we will focus on the digital hardware and software aspects of the serial interface (Figure 7.42).

**Figure 7.42**

A 12-bit DAC interfaced to the SPI port.



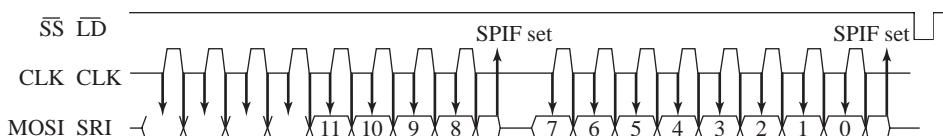
As with any SPI interface, there are basic interfacing issues to consider.

1. **Word size.** In this case we need to transmit 12 bits to the DAC. The DAC8043 data sheet suggests that we can embed the 12-bit data right-justified (it will ignore the first 4 bits that are sent) into a 16-bit transmission.
2. **Bit order.** The DAC8043 requires the most significant bits first.
3. **Clock phase, clock polarity.** There are two issues to resolve. Since the DAC8043 samples its serial input data on the rising edge of the clock, the SPI must change the data on the falling edge. CPOL=CPHA=0 and CPOL=CPHA=1 both satisfy this requirement. The second issue is which edge comes first, the rise or the fall. In this interface it probably doesn't matter.
4. **Bandwidth.** We look at the timing specifications of the DAC8043. The minimum clock low width of 120 ns means the shortest SPI period we can use is 250 ns.

The first 4 bits sent will be 0, followed by the 12 data bits that specify the analog output (Figure 7.43).

**Figure 7.43**

DAC8043 DAC serial timing.



Because we want the  $\overline{LD}$  signal to remain high for the entire 16-bit transfer then pulse low, we will implement it using the regular I/O pin functions. The ritual initializes the direction register, SPI mode, and bandwidth (Program 7.3). To change the DAC output, two 8-bit transmissions are sent (Program 7.4).

```

DAC_Init          ; PM3=LD=1
    bset DDRM,#$38 ; PM5=CLK=SPI clock out
    bset PTM,#$08   ; PM4=SRI=SPI master out
;bit SPICR1
; 7 SPIE = 0      no interrupts
; 6 SPE = 1       enable SPI
; 5 SPTIE= 0      no interrupts
; 4 MSTR = 1      master
; 3 CPOL = 0      output changes on fall,
; 2 CPHA = 0      clock normally low
; 1 SSOE = 0      PM3 regular output, LD
; 0 LSBF = 0      most sign bit first
    movb #$50,SPICR1
    movb #$00,SPICR2 ;normal mode
    movb #$00,SPIBR  ;2 MHz
    rts

```

```

void DAC_Init(void){ // PM3=LD=1
    DDRM |= 0x38; // PM5=CLK=SPI clock out
    PTM |= 0x08; // PM4=SRI=SPI master out
/* bit SPICR1
    7 SPIE = 0      no interrupts
    6 SPE = 1       enable SPI
    5 SPTIE= 0      no interrupts
    4 MSTR = 1      master
    3 CPOL = 0      output changes on fall,
    2 CPHA = 0      clock normally low
    1 SSOE = 0      PM3 regular output, LD
    0 LSBF = 0      most sign bit first */
    SPICR1 = 0x50;
    SPICR2 = 0x00; // normal mode
    SPIBR = 0x00; // 2 MHz
}

```

### Program 7.3

Software initialization for a DAC interface using the SPI.

```

send      ;RegA has data to send
brcclr SPISR,#$20,* ;1. wait SPTEF
staa   SPIDR       ;2. out data
brcclr SPISR,#$80,* ;3. wait SPIF
ldaa   SPISR       ;4. in data
rts

;Reg D has data to output to DAC
DAC_out
  bsr  send      ;msbyte
  tba
  bsr  send      ;lsbyte
  bclr PTM,#$08 ;PM3=LD=0
  bset PTM,#$08 ;PM3=LD=1
  rts

```

```

void send(unsigned char code){
  unsigned char dummy;
  while((SPISR&0x20)==0){};// wait SPTEF
  SPIDR = code;           // data out
  while((SPISR&0x80)==0){};// wait SPIF
  dummy = SPIDR;          // clear SPIF
}

void DAC_out(unsigned short code){
  unsigned char dummy;
  send(code>>8);        // msbyte
  send(code);             // lsbyte
  PTM &= ~0x08;            // PM3=LD=0
  PTM |= 0x08;             // PM3=LD=1
}

```

**Program 7.4**

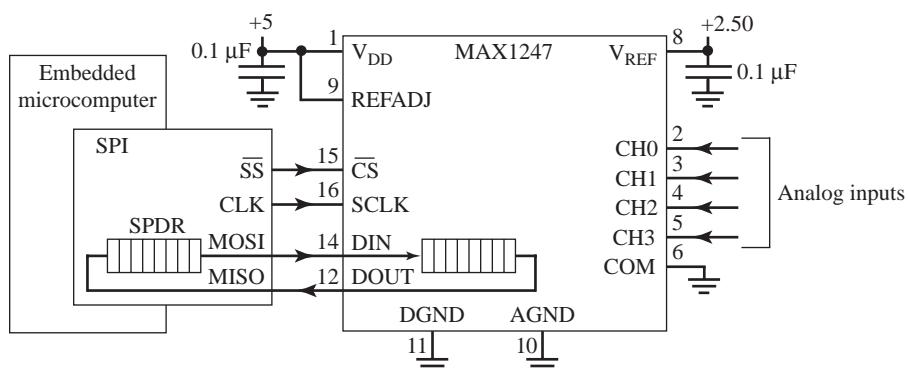
Function to send data to the DAC using the SPI.

**Example 7.5** Interface the Maxim MAX1247 analog to digital converter to the 9S12.

**Solution** This second SPI example shows the synchronous serial interface between the computer and a Maxim MAX1247 ADC. An ADC accepts an analog input (in our case a voltage between 0 and +2.5 V on one of the four analog inputs CH3–CH0) and creates a digital output (in our case a number between 0 and 4095). Detailed discussion of ADCs will also be presented later in Chapter 11. Here in this section we will focus on the hardware and software aspects of the serial interface (Figure 7.44).

**Figure 7.44**

A four-channel 12-bit ADC interfaced to the SPI port.

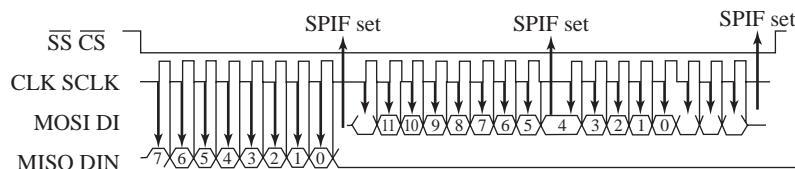


Again, there are basic interfacing issues to consider for this interface:

1. *Word size.* In this case we need to first transmit 8 bits to the ADC, then receive 12 bits back from the ADC. The MAX1247 data sheet suggests that it will embed the 12-bit data into two 8-bit transmissions.
2. *Bit order.* The MAX1247 requires the most significant bits first.
3. *Clock phase, clock polarity.* Since the MAX1247 samples its serial input data on the rising edge of the clock, the SPI must change the data on the falling edge. CPOL=CPHA=0 and CPOL=CPHA=1 both satisfy this requirement. We will use the CPOL=CPHA=0 mode as suggested in the Maxim data sheet.
4. *Bandwidth.* We look at the timing specifications of the MAX1247. The maximum SCLK frequency is 2 MHz, and the minimum clock low/high widths is 200 ns, so the shortest SPI period we can use is 500 ns.

The first 8 bits sent will specify the channel and ADC mode. After conversion, the 12 data bits result is returned. Notice that bit 7 of the mode select is always high, and the 12-bit ADC result is embedded into the middle of the two 8-bit transmissions. The software will shift the 16-bit data 3 bits to the right to produce the 0 to 4095 result (Figure 7.45).

**Figure 7.45**  
MAX 1247 ADC serial timing.



Because we want the  $\overline{CS}$  signal to remain low for the entire 24-bit transfer, we will implement it using the regular I/O pin functions. The ritual initializes the direction register, SPI mode, and bandwidth (Program 7.5).

```
ADC_Init          ;PM3=CS=1
    bset DDRM,#$38 ;PM5=SCLK=SPI clock out
    bclr DDRM,#$04 ;PM2=DOUT=SPI master in
    bset PTM,#$08 ;PM4=DIN=SPI master out
;bit SPICR1
; 7 SPIE = 0  no interrupts
; 6 SPE  = 1  enable SPI
; 5 SPTIE= 0  no interrupts
; 4 MSTR = 1  master
; 3 CPOL = 0  output changes on fall,
; 2 CPHA = 0  clock normally low
; 1 SSOE = 0  PM3 regular output, CS
; 0 LSBF = 0  most sign bit first
    movb #$50,SPICR1
    movb #$00,SPICR2 ;normal mode
    movb #$00,SPIBR ;2 MHz
    rts
```

```
void ADC_Init(void){ // PM3=CS=1
    DDRM |=0x38; // PM5=SCLK=SPI clock out
    DDRM &=~0x04; // PM2=DOUT=SPI master in
    PTM |=0x08; // PM4=DIN=SPI master out
/* bit SPICR1
    7 SPIE = 0  no interrupts
    6 SPE  = 1  enable SPI
    5 SPTIE= 0  no interrupts
    4 MSTR = 1  master
    3 CPOL = 0  output changes on fall,
    2 CPHA = 0  clock normally low
    1 SSOE = 0  PM3 regular output, CS
    0 LSBF = 0  most sign bit first */
    SPICR1 = 0x50;
    SPICR2 = 0x00; // normal mode
    SPIBR = 0x00; // 2 MHz
}
```

### Program 7.5

Software initialization for an ADC interface using the SPI.

Recall that when the software outputs to the SPI data register, the 8-bit register in the SPI is exchanged with the 8-bit register in the ADC. To read the ADC, three 8-bit transmissions are exchanged (Program 7.6). On the first exchange, the software specifies the channel and ADC mode, then the 12-bit ADC data are returned during the second and third transmission. All the ADC modes in the following `#define` statements implement unipolar voltage range, single-ended, and external clock:

```
#define CH0 0x9F
#define CH1 0xDF
#define CH2 0xAF
#define CH3 0xEF
```

```
spi   ;RegA both input and output data
    brclr SPISR,#$20,* ;1. wait SPTEF
    staa SPIDR ;2. out data
    brclr SPISR,#$80,* ;3. wait SPIF
    ldaa SPISR ;4. in data
    rts
```

```
unsigned char spi(unsigned char code){
    while((SPISR&0x20)==0){};// wait SPTEF
    SPIDR = code;           // data out
    while((SPISR&0x80)==0){};// wait SPIF
    return SPIDR;           // clear SPIF
}
```

```

;Input: Reg A is channel code
;Output: Reg D is 12-bit ADC result
ADC_In    bclr PTM,#$08 ; PM3=CS=0
          bsr spi      ;channel,mode
          clra
          bsr spi      ; msbyte of ADC
          psha        ;RegA = bits 15-8
          clra
          bsr spi      ; lsbyte of ADC
          tab         ;RegB = bits 7-0
          bset PTM,#$08 ;PM3=CS=1
          pul        ;RegD is bits 15-0
          lsr
          lsr
          lsr        ;right justify
          rts

```

```

unsigned short ADC_In(unsigned char code){
  unsigned short data;
  PTM &= ~0x08;           // PM3=CS=0
  spi(code);             // send channel,mode
  data = spi(0)<<8;     // msbyte of ADC
  SPIDR = 0;              // start SPI
  data += spi(0);         // lsbyte of ADC
  PTM |= 0x08;            // PM3=CS=1
  return data>>3;        // right justify
}

```

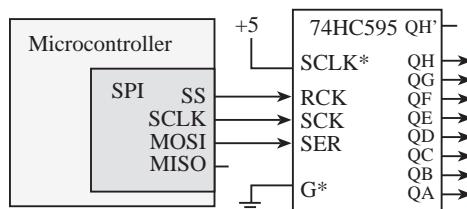
### Program 7.6

Software function to input from an ADC using the SPI.

### Example 7.6 Design an output port expander using a shift register and SPI.

**Solution** Sometimes we need more output pins than available on our microcontroller. In general, the proper design approach would be to upgrade to a microcontroller with more pins. However, in situations where we do not have the time or money to change microcontrollers, we can interface a 74HC595 shift register to the SPI port for a quick solution providing additional output pins. Basically, three pins of the SPI (SS, MOSI, and SCLK) will be converted to eight digital outputs **Q** on the 74HC595, as shown in Figure 7.46. Additional shift registers can be chained together (connect the **QH'** outputs of one to the **SER** inputs of the next) to provide additional outputs without requiring more 9S12 pins. The gate input, **G\***, of the 74HC595 is grounded so the eight **Q** outputs will be continuously driven. The SPI clock output is connected to the 74HC595 clock input (**SCK**), and the SPI data output is connected to the 74HC595 data input (**SER**). The SPI mode (CPOL = 0, CHPA = 0) is selected to the 9S12, changes the output data on the fall of the clock, and the 74HC595 shifts data in on the rise. Because the 74HC595 is fast (maximum clock 25 MHz and setup time of 20 ns), we run the SPI as fast as possible. After eight bits are transferred from the 9S12 to the 74HC595, software will create a rising edge of **RCK**, causing the new data to be latched into the 74HC595. If there is just one 74HC595 like Figure 7.46, we can use the automatic SS feature of the SPI to create a rising edge latch on **RCLK**. To enable this feature, we set bit 1 **SSOE** of **SPICR1** and bit 4 **MODFEN** of **SPICR2** as described in Table 7.14. If we were chaining multiple shift registers, we would not use the automatic SS feature, rather we would output all the data, then manually latch it in with

**Figure 7.46**  
Interface between the 9S12 and a 74HC595 shift register.



explicit outputs on **RCK**. In this solution, we perform one SPI transmission to change all eight bits of the port output (Program 7.7). The SS pulse, like Figure 7.40, occurs automatically and does not require software overhead to produce.

<pre> Port_Init     bset DDRM,#\$38 ;PM3,PM4,PM5 output ;bit SPICR1 ; 7 SPIE = 0    no interrupts ; 6 SPE  = 1    enable SPI ; 5 SPTIE= 0    no interrupts ; 4 MSTR = 1    master ; 3 CPOL = 0    output changes on fall, ; 2 CPHA = 0    clock normally low ; 1 SSOE = 1    PM3 automatic output ; 0 LSBF = 0    most sign bit first     movb #\$52,SPICR1     movb #\$10,SPICR2 ;SS automatic     movb #\$00,SPIBR ;Frequency is E/2     rts ;RegA has data to output to 74HC595 Port_Out     brclr SPISR,#\$20,* ;1. wait SPTEF     staa SPIDR          ;2. out data     brclr SPISR,#\$80,* ;3. wait SPIF     ldaa SPISR          ;4. in data     rts </pre>	<pre> void Port_Init(void){     DDRM  = 0x38; // PM3,PM4,PM5 output /* bit SPICR1     7 SPIE = 0    no interrupts     6 SPE  = 1    enable SPI     5 SPTIE= 0    no interrupts     4 MSTR = 1    master     3 CPOL = 0    output changes on fall,     2 CPHA = 0    clock normally low     1 SSOE = 1    PM3 automatic output     0 LSBF = 0    most sign bit first */     SPICR1 = 0x52;     SPICR2 = 0x10; // SS automatic     SPIBR = 0x00; // Frequency is E/2 } void Port_Out(unsigned char code){     unsigned char dummy;     while((SPISR&amp;0x20)==0){};// wait SPTEF     SPIDR = code;           // data out     while((SPISR&amp;0x80)==0){};// wait SPIF     dummy = SPIDR;          // clear SPIF } </pre>
--	--

### Program 7.7

Software to control an output parallel port expanded using the SPI.

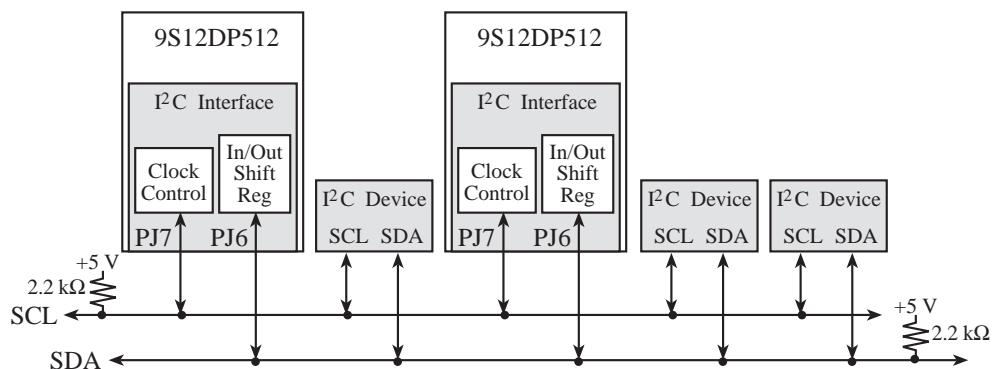
## 7.7 Inter-Integrated Circuit ( $I^2C$ ) Interface

### 7.7.1 The Fundamentals of the $I^2C$

Ever since microcontrollers have been developed, there has been a desire to shrink the size of an embedded system, reduce its power requirements, and increase its performance and functionality. Two mechanisms to make systems smaller are to integrate functionality into the microcontroller and to reduce the number of I/O pins. The inter-integrated circuit  $I^2C$  interface was proposed by Philips in the late 1980s as a means to connect external devices to the microcontroller using just two wires. The SPI interface has been very popular, but it takes three wires for simplex and four wires for full-duplex communication. In 1998, the  $I^2C$  Version 1 protocol became an industry standard and has been implemented into thousands of devices. The  $I^2C$  bus is a simple two-wire bi-directional serial communication system that is intended for communication between microcontrollers and their peripherals over short distances. This is typically, but not exclusively, between devices on the same printed circuit board—the limiting factor being the bus capacitance. It also provides flexibility, allowing additional devices to be connected to the bus for further expansion and system development. The interface will operate at bit rates of up to 100 kbps with maximum capacitive bus loading. The module can operate up to a baud rate of 400 kbps provided the  $I^2C$  bus slew rate is less than 100ns. The maximum interconnect length and the number of devices that can be connected to the bus are limited by a maximum bus capacitance of 400 pF in all instances. Version 2.0 supports a high-speed mode with a data rate up to 2.4 MHz. This section will focus on Version 1, because the 9S12 does not support Version 2.

Figure 7.47 shows a block diagram of a communication system based on the  $I^2C$  interface found in many 9S12 microcontrollers. The master/slave network may consist of

**Figure 7.47**  
Block diagram of an  $I^2C$  communication network.



multiple masters and multiple slaves. The **Serial Clock Line** (SCL) and the **Serial Data line** (SDA) are both bi-directional. Each line is open collector, meaning a device may drive it low or let it float. A logic high occurs if all devices let the output float, and a logic low occurs when at least one device drives it low. The value of the pull-up resistor depends on the speed of the bus. 4.7 k $\Omega$  is recommended for data rates below 100 kbps, 2.2 k $\Omega$  is recommended for standard mode, and 1 k $\Omega$  is recommended for fast mode.

**Checkpoint 7.12:** Why is the recommended pull-up resistor related to the bus speed?

The SCL clock is used in a synchronous fashion to communicate on the bus. Even though data transfer is always initiated by a master device, both the master and the slaves have control over the data rate. The master starts a transmission by driving the clock low, but if a slave wishes to slow down the transfer, it too can drive the clock low (called **clock stretching**). In this way, devices on the bus will wait for all devices to finish. Both address (from Master to Slaves) and information (bi-directional) are communicated in serial fashion on SDA.

The bus is initially idle where both SCL and SDA are both high. This means no device is pulling SCL or SDA low. The communication on the bus, which begins with a START and ends with a STOP, consists of five components:

**START (S)** is used by the master to initiate a transfer.

**DATA** is sent in 8-bit blocks and consists of:

    7-bit address and 1-bit direction from the master

    control code for master to slaves

    information from master to slave

    information from slave to master

**ACK (A)** is used by slave to respond to the master after each 8-bit data transfer.

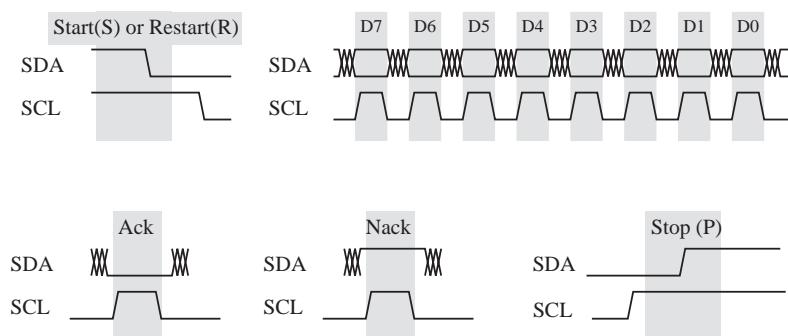
**RESTART (R)** is used by the master to initiate additional transfers without releasing the bus.

**STOP (P)** is used by the master to signal that the transfer is complete and the bus is free.

The basic timings for these components are drawn in Figure 7.48. For now, we will discuss basic timing, but we will deal with issues like stretching and arbitration later. A slow slave uses clock stretching to give it more time to react, and masters will use arbitration when two or more masters want the bus at the same time. An idle bus has both SCL and SDA high. A transmission begins when the master pulls SDA low, causing a START (S) component. The timing of a RESTART is the same as a START. After a START or a RESTART, the next eight bits will be an address (7-bit address plus 1-bit direction). There are 128 possible 7-bit addresses, however, 32 of them are reserved as special commands. The address is used to enable a particular slave. All data transfers are eight bits long, followed by a 1-bit acknowledge. During a data transfer, the SDA data line must be stable (high or low) whenever the SCL clock line is high. There is one clock pulse on SCL for

**Figure 7.48**

Timing diagrams of I<sup>2</sup>C components.



each data bit, the MSB being transferred first. Next, the selected slave will respond with a positive acknowledge (Ack) or a negative acknowledge (Nack). If the direction bit is 0 (write), then subsequent data transmissions contain information sent from master to slave. For a write data transfer, the master drives the RDA data line for eight bits, then the slave drives the acknowledge condition during the ninth clock pulse. If the direction bit is 1 (read), then subsequent data transmissions contain information sent from slave to master. For a read data transfer, the slave drives the RDA data line for eight bits, then the master drives the acknowledge condition during the ninth clock pulse. The STOP component is created by the master to signify the end of transfer. A STOP begins with SCL and SDA both low, then makes the SCL clock high, and ends by making SDA high. The rising edge of SDA while SCL is high signifies the STOP condition.

**Checkpoint 7.13:** What happens if no device sends an acknowledgement?

Figure 7.49 illustrates the case where the master sends 2 bytes of data to a slave. The shaded regions demarcate signals driven by the master, and the white areas show those times when the signal is driven by the slave. Regardless of format, all communication begins when the master creates a START component followed by the 7-bit address and 1-bit direction. In this example, the direction is low, signifying a write format. The 1<sup>st</sup> through 8<sup>th</sup> SCL pulses are used to shift the address/direction into all the slaves. In order to acknowledge the master, the slave that matches the address will drive the SDA data line low during the 9<sup>th</sup> SCL pulse. The 10<sup>th</sup> through 17<sup>th</sup> SCL pulses sends the data to the selected slave. The selected slave will acknowledge by driving the SDA data line low during the 18<sup>th</sup> SCL pulse. A second data byte is transferred from master to slave in the same manner. In this particular example, two data bytes were sent, but this format can be used to send any number of bytes, because once the master captures the bus it can transfer as many bytes as it wishes. If the slave receiver does not acknowledge the master, the SDA line will be left high (Nack). The master can then generate a STOP signal to abort the data transfer or a RESTART signal to commence a new transmission. The master signals the end of transmission by sending a STOP condition.

**Figure 7.49**

I<sup>2</sup>C transmission of two bytes from master to slave.

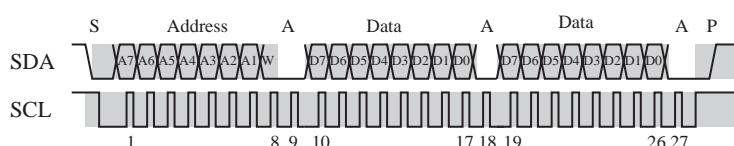
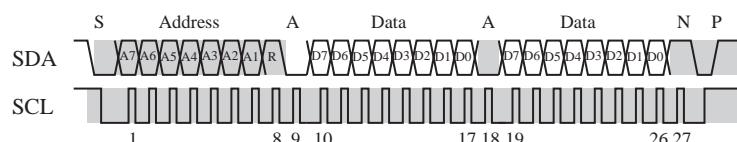


Figure 7.50 illustrates the case where a slave sends two bytes of data to the master. Again, the master begins by creating a START component followed by the 7-bit address and 1-bit direction. In this example, the direction is high, signifying a read format. During the 10<sup>th</sup> through 17<sup>th</sup> SCL pulses, the selected slave sends the data to the master. The selected slave can only change the data line while SCL is low and must be held stable while SCL is high.

**Figure 7.50**  
 $I^2C$  transmission of two bytes from slave to master.



The master will acknowledge by driving the SDA data line low during the 18<sup>th</sup> SCL pulse. Only two data bytes are shown in Figure 7.50, but this format can be used to receive as many bytes the master wishes. Except for the last byte, all data are transferred from slave to master in the same manner. After the last data byte, the master does not acknowledge the slave (Nack) signifying ‘end of data’ to the slave, so the slave releases the SDA line for the master to generate STOP or RESTART signal. The master signals the end of transmission by sending a STOP condition.

Figure 7.51 illustrates the case where the master uses the RESTART command to communicate with two slaves, reading one byte from one slave and writing one byte to the other. As always, the master begins by creating a START component followed by the 7-bit address and 1-bit direction. During the first start, the address selects the first slave and the direction is read. During the 10<sup>th</sup> through 17<sup>th</sup> SCL pulses, the first slave sends the data to the master. Because this is the last byte to be read from the first slave, the master will not acknowledge letting the SDA data float high during the 18<sup>th</sup> SCL pulse, so the first slave releases the SDA line. Rather than issuing a STOP at this point, the master issues a repeated start or RESTART. The 7-bit address and 1-bit direction transferred in the 20<sup>th</sup> through 27<sup>th</sup> SCL pulses will select the second slave for writing. In this example, the direction is low, signifying a write format. The 28<sup>th</sup> through 36<sup>th</sup> SCL pulses send the data to the second slave. During the 37<sup>th</sup> pulse the second slave pulls SDA low to acknowledge the data it received. The master signals the end of transmission by sending a STOP condition.

**Figure 7.51**  
 $I^2C$  transmission of one byte from the first slave and one byte to a second slave.

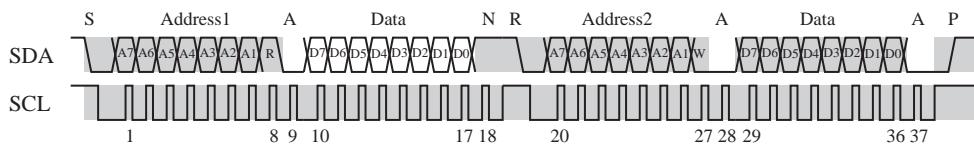


Table 7.17 lists some addresses that have special meaning. A write to Address 0 is a general call address, and is used by the master to send commands to all slaves. The 10-bit address mode gives two address bits in the first frame and eight more address bits in the second frame. The direction bit for 10-bit addressing is in the first frame.

**Table 7.17**  
 Special addresses used in the  $I^2C$  network.

Address	R/W	Description
0000 000	0	General call address
0000 000	1	Start byte
0000 001	x	CBUS address
0000 010	x	Reserved for different bus formats
0000 011	0	Reserved
0000 1xx	x	High speed mode
1111 0xx	x	10-bit address
1111 1xx	X	Reserved

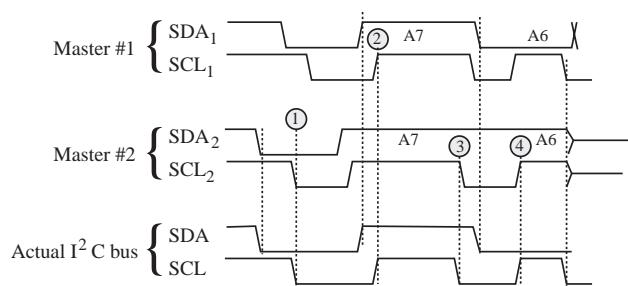
## 7.7.2 $I^2C$ Synchronization

The  $I^2C$  bus supports multiple masters. If two or more masters try to issue a START command on the bus at the same time, both clock synchronization and arbitration will occur. **Clock synchronization** is procedure that will make the low period equal to the longest

clock low period and the high is equal to the shortest one among the masters. Figure 7.52 illustrates clock synchronization, where the top set of traces is generated by the first master, and the second set of traces is generated by the second master. Since the outputs are open collector, the actual signals will be the wired-AND of the two outputs. Each master repeats these steps when it generates a clock pulse. It is during step 3 that the faster device will wait for the slower device.

1. Drive its SCL clock low for a fixed amount of time
2. Let its SCL clock float
3. Wait for the SCL to be high
4. Wait for a fixed amount of time, stop waiting if the clock goes low

**Figure 7.52**  
I<sup>2</sup>C timing illustrating  
clock synchronization  
and data arbitration.



Because the outputs are open collector, the signal will be pulled to a logic high by the  $2\ \Omega$  resistor only if all devices release the line (output a logic high). Conversely, the signal will be a logic low if any device drives it low. When masters create a START, they first drive SDA low, then drive SCL low. If a group of masters are attempting to create START commands at about the same time, then the wire-AND of their SDA lines has its 1 to 0 transition before the wire-AND of their SCL lines has its 1 to 0 transition. Thus, a valid START command will occur causing all the slaves to listen to the upcoming address. In the example shown in Figure 7.52, Master #2 is the first to drive its clock low. In general, the SCL clock will be low from the time the first master drives it low (time 1 in this example), until the time the last master releases its clock (time 2 in this example.) Similarly, the SCL clock will be high from the time the last master releases its clock (time 2 in this example), until the time the first master drives its clock low (time 3 in this example.)

The relative priority of the contending masters is determined by a **data arbitration** procedure. A bus master loses arbitration if it transmits logic “1” while another master transmits logic “0”. The losing masters immediately switch over to slave receive mode and stop driving the SCL and SDA outputs. In this case, the transition from master to slave mode does not generate a STOP condition. Meanwhile, a status bit is set by hardware to indicate loss of arbitration. In the example shown in Figure 7.52, Master #1 is generating an address with A7 = 1 and A6 = 0, while Master #2 is generating an address with A7 = 1 and A6 = 1. Between times 2 and 3, both masters are attempting to send A7 = 1, and notice the actual SDA line is high. At time 4, Master #2 attempts to make the SDA high (A6 = 1), but notices the actual SDA line is low. In general, the master sending a message to the lowest address will win arbitration.

The third synchronization mechanism occurs between master and slave. If the slave is fast enough to capture data at the maximum rate, the transfer is a simple synchronous serial mechanism. In this case the transfer of each bit from master to slave is illustrated by the following interlocked sequences (see Figure 7.48).

#### Master sequence

1. Drive its SCL clock low
2. Set the SDA line
3. Wait for a fixed amount of time
4. Let its SCL clock float

#### Slave sequence (no stretch)

5. Wait for the SCL to be high
6. Wait for a fixed amount of time
7. Stop waiting if the clock goes low

6. Capture SDA data on low to high edge of SCL

If the slave is not fast enough to capture data at the maximum rate, it can perform an operation called **clock stretching**. If the slave is not ready for the rising edge of SCL, it will hold the SCL clock low itself until it is ready. Slaves are not allowed to cause any 1 to 0 transitions on the SCL clock, but rather can only delay the 0 to 1 edge. The transfer of each bit from master to slave with clock stretching is illustrated by the following sequences:

#### Master sequence

1. Drive its SCL clock low
2. Set the SDA line
3. Wait for a fixed amount of time
4. Let its SCL clock float
5. Wait for the SCL clock to be high
6. Wait for a fixed amount of time
7. Stop waiting if the clock goes low

#### Slave sequence (clock stretching)

1. Wait for the SCL clock to be low
2. Drive SCL clock low
3. Wait until it's ready to capture
4. Let its SCL float
5. Wait for the SCL clock to be high
6. Capture the SDA data

Clock stretching can also be used when transferring a bit from slave to master:

#### Master sequence

1. Drive its SCL clock low
2. Wait for a fixed amount of time
4. Let its SCL clock float
5. Wait for the SCL clock to be high
6. Capture the SDA input
7. Wait for a fixed amount of time
8. Stop waiting if the clock goes low

#### Slave sequence (clock stretching)

1. Wait for the SCL clock to be low
2. Drive SCL clock low
3. Wait until next data bit is ready
4. Let its SCL float
5. Wait for the SCL clock to be high

**Observation:** Clock stretching allows fast and slow devices to exist on the same  $I^2C$  bus fast devices will communicate quickly with each other, but slow down when communicating with slower devices.

**Checkpoint 7.14:** Arbitration continues until one master sends a zero while the other sends a one. What happens if two masters attempt to send data to the same address?

### 7.7.3 9S12 $I^2C$ Details

Many 9S12 microcontrollers have an  $I^2C$  interface, but they implement just a subset of the standard. They support master and slave modes, can generate interrupts on START and STOP conditions, and allow  $I^2C$  networks with multiple masters. On the other hand, the 9S12 microcontrollers do not support general call, 10-bit addressing, or high speed mode. As shown in Figure 7.47, I/O pins PJ7 and PJ6 can be connected directly to an  $I^2C$  network. Because  $I^2C$  networks are intended to connect devices on the same PCB board, no special hardware drivers are required. Stop mode and wait mode are two low power states. Stop mode occurs when the device is turned off (**IBEN** = 0), and wait mode is a general state issued by the software when it executes a **wait** instruction. Table 7.18 lists the  $I^2C$  ports on the 9S12D family.

**Table 7.18**  
9S12  $I^2C$  ports.

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$00E0	ADR7	ADR6	ADR5	ADR4	ADR3	ADR2	ADR1	0	IBAD
\$00E1	IBC7	IBC6	IBC5	IBC4	IBC3	IBC2	IBC1	IBC0	IBFD
\$00E2	IBEN	IBIE	MS/SL	Tx/Rx	TXAK	RSTA	0	IBSWAI	ICR
\$00E3	TCF	IAAS	IBB	IBAL	0	SRW	IBIF	RXAK	IBSR
\$00E4	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	IBDR

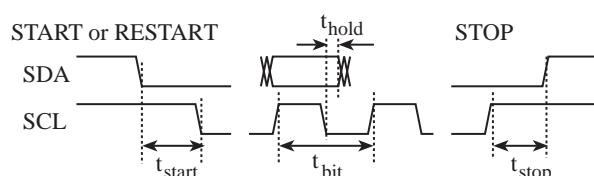
**IBAD** is the Bus Address Register. This register contains the address the 9S12 will respond to when addressed as a slave, thus, it is not the address sent on the bus during the address transfer. **IBCR** is the I<sup>2</sup>C control register, containing many of the bits that configure the I<sup>2</sup>C interface. **IBEN** is the I<sup>2</sup>C enable bit, which must be set to activate the interface. **IBIE** is the shared interrupt arm bit for the three flags **IAAS**, **TCF**, and **IBAL**. **MS/SL** is the master/slave bit, where 1 means master and 0 means slave. When this bit is changed from 0 to 1, a START signal is generated. When this bit is changed from 1 to 0, a STOP signal is generated. A STOP signal should only be generated if the **IBIF** flag is set. **MS/SL** is cleared without generating a STOP signal automatically when the master loses arbitration. The **Tx/Rx** bit specifies whether the next data transfer will be an output (equals 1) or an input (equals 0). When operating the interface as a slave and an address match occurs, the **Tx/Rx** bit should be set to match the **SRW** flag received during the address match. When sending an address as a master, **Tx/Rx** should be 1. When a master sends data to a slave, it specifies the R/W bit (bit 0 of the address frame), and sets **Tx/Rx** in the **IBCR**. **TXAK** specifies the value driven onto SDA during data acknowledge cycles for both master and slave receivers. The I<sup>2</sup>C module will always acknowledge address matches, provided it is enabled, regardless of the value of **TXAK**. **TXAK** is only used when the I<sup>2</sup>C Bus is a receiver, not a transmitter. When receiving data as a master or a selected slave, this bit determines if an acknowledgement will be sent during the ninth clock bit. 0 means an acknowledgement will be sent (Ack), and 1 means no acknowledgement will be sent (Nack). A repeated start (RESTART) will be sent if the master writes a 1 to the **RSTA** bit, provided this 9S12 is the current bus master. **RSTA** is a write-only bit, reads from this bit always return 0. Attempting a repeated start when the bus is owned by another master will result in loss of arbitration. If **IBSWAI** is 1, then the I<sup>2</sup>C device will halt during wait mode.

**IBFD** is the I<sup>2</sup>C Bus Frequency Divider Register, which determines the baud rate transferred as a master. The bit clock generator is implemented as a prescale divider—IBC7-6, prescaled shift register—IBC5-3 select the prescaler divider and IBC2-0 select the shift register tap point. The timing of the 9S12 I<sup>2</sup>C interface is derived from the bus clock. Table 7.19 presents the three fields of IBFD that define operating speed. Figure 7.53 defines the four timing intervals. **t<sub>start</sub>** is the delay from fall of SDA data to fall of SCL clock during a START or RESTART. **t<sub>bit</sub>** is the time to transfer one bit. **t<sub>hold</sub>** is the time after the fall of the clock that the data will remain valid. **t<sub>stop</sub>** is the delay from the rise of SCL

**Table 7.19**  
9S12 I<sup>2</sup>C timing components as specified by IBFD.

IBC7-6 MUL		IBC5-3				IBC2-0	
		scl2 start	scl2 stop	scl2 tap	tap2 tap	SCLTap	SDATap
00	1	000	2	7	4	1	
01	2	001	2	7	4	2	
10	4	010	2	9	6	4	
11	reserved	011	6	9	6	8	
		100	14	17	14	16	
		101	30	33	30	32	
		110	62	65	62	64	
		111	126	129	126	128	

**Figure 7.53**  
9S12 I<sup>2</sup>C timing intervals.



clock to rise of SDA data during a STOP. Table 7.20 gives the ratio of bus frequency to  $I^2C$  frequency for all possible values of **IBFD**. For example, if the bus frequency is 8 MHz, and we wish to create an  $I^2C$  clock frequency of 200 kHz, then we need a divider value of 40. From Table 7.20, we see **IBFD** could be chosen as \$07, \$0B, or \$40.

Let  $t_E$  be the period of the bus clock, then the four timing intervals are:

$$t_{\text{bit}} = t_E \cdot \text{MUL} \cdot \{2 \cdot (\text{scl2tap} + [(\text{SCLTap} - 1) \cdot \text{tap2tap}] + 2)\}$$

$$t_{\text{hold}} = t_E \cdot \text{MUL} \cdot \{\text{scl2tap} + [(\text{SDATap} - 1) \cdot \text{tap2tap}] + 3\}$$

$$t_{\text{start}} = t_E \cdot \text{MUL} \cdot [\text{scl2start} + (\text{SCLTap} - 1) \cdot \text{tap2tap}]$$

$$t_{\text{stop}} = t_E \cdot \text{MUL} \cdot [\text{scl2stop} + (\text{SCLTap} - 1) \cdot \text{tap2tap}]$$

<b>IBFD</b>	\$0-	\$1-	\$2-	\$3-	\$4-	\$5-	\$6-	\$7-	\$8-	\$9-	\$A-	\$B-
\$-0	20	48	160	640	40	96	320	1280	80	192	640	2560
\$-1	22	56	192	768	44	112	384	1536	88	224	768	3072
\$-2	24	64	224	896	48	128	448	1792	96	256	896	3584
\$-3	26	72	256	1024	52	144	512	2048	104	288	1024	4096
\$-4	28	80	288	1152	56	160	576	2304	112	320	1152	4608
\$-5	30	88	320	1280	60	176	640	2560	120	352	1280	5120
\$-6	34	104	384	1536	68	208	768	3072	136	416	1536	6144
\$-7	40	128	480	1920	80	256	960	3840	160	512	1920	7680
\$-8	28	80	320	1280	56	160	640	2560	112	320	1280	5120
\$-9	32	96	384	1536	64	192	768	3072	128	384	1536	6144
\$-A	36	112	448	1792	72	224	896	3584	144	448	1792	7168
\$-B	40	128	512	2048	80	256	1024	4096	160	512	2048	8192
\$-C	44	144	576	2304	88	288	1152	4608	176	576	2304	9216
\$-D	48	160	640	2560	96	320	1280	5120	192	640	2560	10240
\$-E	56	192	768	3072	112	384	1536	6144	224	768	3072	12288
\$-F	68	240	960	3840	136	480	1920	7680	272	960	3840	15360

**Table 7.20**

9S12  $I^2C$  clock divider values as specified by IBFD.

**Checkpoint 7.15:** Assuming a 24 MHz bus clock, what value can you program into **IBFD** to create a 100 kHz baud rate?

**IBSR** is the  $I^2C$  Bus Status Register. This status register is read-only, except that **IBIF**, **IAAS**, and **IBAL** can be cleared if the software writes a 1 to the corresponding bit position. **TCF** is the Data transferring bit. While one byte of data is being transferred, this bit is cleared. It is set by the falling edge of the ninth clock of a byte transfer. Note that this bit is only valid during or immediately following a transfer to the  $I^2C$  module or from the  $I^2C$  module. **IAAS** is addressed as a slave bit, which is set when its own specific address (**IBAD**) is matched with the calling address sent by another master. The **IAAS** flag will request an interrupt if the **IBIE** arm bit is set. After **IAAS** is set, the software should check the **SRW** bit and set its **Tx/Rx** mode accordingly. Writing **IAAS** clears this bit. **IBB** is the Bus busy bit, indicating the status of the bus. When a START signal is detected, the **IBB** is set. If a STOP signal is detected, **IBB** is cleared. A master should wait until **IBB** is 0, signifying the bus is idle before initiating a transfer. **IBAL** is the Arbitration Lost bit, which is set by hardware when the arbitration procedure is lost. Arbitration is lost in the following circumstances:

- SDA sampled low when the master drives a high during an address or data transmit cycle.
- SDA sampled low when master drives a high during the acknowledge bit of a data receive cycle.

- A START cycle is attempted when the bus is busy.
- A RESTART cycle is requested in slave mode.
- A STOP condition is detected when the master did not request it.

The **IBAL** bit must be cleared by software, by writing a one to it. **SRW** is Slave Read/Write bit, which indicates the value of the R/W command bit of the calling address sent from the master. This bit is only valid when the 9S12 is in slave mode; a complete address transfer has occurred with an address match and no other transfers have been initiated. Checking **SRW**, the CPU can select slave transmit/receive mode according to the command of the master. If **SRW** is 0, it means the master writing data to this 9S12 as a slave. Conversely, if **SRW** is 1, master wishes this 9S12 slave to transmit data back to the master. **IBIF** is the I<sup>2</sup>C Interrupt bit, which is set when one of the following conditions occurs:

- Arbitration lost (**IBAL** bit set)
- Byte transfer complete (**TCF** bit set)
- Addressed as slave (**IAAS** bit set)

These three conditions will request an interrupt if the **IBIE** arm bit is set. **IBIF** must be cleared by software, writing a one to it. **RXAK** is the Received Acknowledge bit, which is the value of SDA during the acknowledge bit of a bus cycle. If the **RXAK** is low, it indicates an acknowledge signal has been received during the ninth clock. If **RXAK** is high, it means no acknowledge signal is detected at the ninth clock.

**IBDR** is the I<sup>2</sup>C Bus Data I/O Register. In master transmit mode, when data is written to the **IBDR** a data transfer is initiated. As shown in Figures 7.20 through 7.23, the most significant bit is sent first. In master transmit mode (**MS/SL** = 1 and **Tx/Rx** = 1), writing this register initiates a 9-bit transmission, outputting to both SDA data and SCL clock. In master receive mode (**MS/SL** = 1 and **Tx/Rx** = 0), reading this register initiates next byte data receiving, where the 8-bit data from SDA input and the SCL clock is an output. When either a master or a slave is sending, the **TXAK** bit is send during the ninth clock. In slave mode, input/output functions are available only after an address match has occurred. Note that the **Tx/Rx** bit must correctly reflect the desired direction of transfer in master and slave modes for the transmission to begin. Reading the **IBDR** will return the last byte received while the IIC is configured in either master receive or slave receive modes. The **IBDR** does not reflect every byte that is transmitted on the I<sup>2</sup>C bus, nor can software verify that a byte has been written to the **IBDR** correctly by reading it back. In master transmit mode, the first byte of data written to **IBDR** following assertion of **MS/SL** is used for the address transfer and should comprise of the calling address (in position D7-D1) concatenated with the required R/W bit (in position D0).

### 7.7.4 9S12 I<sup>2</sup>C Single Master Example

The objective of this example is to present a low-level device driver for an I<sup>2</sup>C network where this 9S12 is the only master, as shown in Program 7.8. This simple example will employ busy-wait synchronization. **I2C\_Open** first enables the I<sup>2</sup>C interface, starting out in slave mode. Since this is the only master, it does not need a slave address (**IBAD**).

#### Program 7.8

9S12 I<sup>2</sup>C initialization in single master mode.

```
void I2C_Open(void){
    IBCR = 0x80; // enable, no interrupts, slave mode
    IBFD = 0x1F; // 100kHz, assuming 24 MHz bus clock
    IBSR = 0x02; // clear IBIF
}
```

Program 7.9 contains the function **I2C\_Send** that transmits two bytes to a slave, creating a transmission shown in Figure 7.49. In a system with multiple masters is should check to see if the bus is idle first. Because this system has just one master, the bus should be idle. By setting the **MS/SL** bit, the 9S12 will create a START condition. In a system with

**Program 7.9**

9S12 I<sup>2</sup>C transmission  
in single-master mode.

```
void I2C_Send(char slave, unsigned char data1, unsigned char data2){
    IBCR |= 0x30;           // send START
    IBDR = slave&0xFE;      // send address with D0=0 signifying write
    while((IBSR&0x02)==0){} // wait for the address to be sent
    IBSR = 0x02;            // clear IBIF
    IBDR = data1;           // send first byte
    while((IBSR&0x02)==0){} // wait for the data to be sent
    IBSR = 0x02;            // clear IBIF
    IBDR = data2;           // send second byte
    while((IBSR&0x02)==0){} // wait for the data to be sent
    IBSR = 0x02;            // clear IBIF
    IBCR &=~0x30;           // send STOP
}
```

multiple masters, it should check to see if it lost bus arbitration (**IBAL**). The slave address (with bit 0 equal to 0) will be sent. The two data bytes are sent, then the STOP is issued. If there is a possibility the slave doesn't exist, then this program could have checked **RXAX** after each transfer.

Program 7.10 contains the function **I2C\_Recv** that receives two bytes from a slave, creating a transmission shown in Figure 7.50. By setting the **MS/SL** bit, the 9S12 will create a START condition. During the first transfer, the **Tx/Rx** bit is 1, so the slave address (with bit 0 equal to 1) will be sent. During the second two transfers, the **Tx/Rx** bit is 0, so data flows into the 9S12. To trigger the first data reception, the software performs a dummy read on the **IBDR**. During the first data transfer TXAK is 0, creating a positive acknowledgement. Conversely, during the second data transfer TXAK is 1, creating a negative acknowledgement, signaling to the slave that this is the last data to be transferred. The two data bytes are received, then the STOP is issued. Notice that the last byte is captured after the STOP is issued, so that a third data transfer is not initiated.

**Program 7.10**

9S12 I<sup>2</sup>C reception in  
single-master mode.

```
unsigned short I2C_Recv(char slave){ unsigned char data1,data2;
    IBCR |= 0x30;           // send START
    IBDR = slave|0x01;      // send address with D0=1 signifying read
    while((IBSR&0x02)==0){} // wait for the address to be sent
    IBSR = 0x02;            // clear IBIF
    IBCR &= ~0x18;          // Tx/Rx=0, and TXAK=0
    data1 = IBDR;           // dummy read to initiate receiving
    while((IBSR&0x02)==0){} // wait for the data to be received
    IBSR = 0x02;            // clear IBIF
    IBCR |= 0x08;           // TXAK=1
    data1 = IBDR;           // capture first byte, initiate second
    while((IBSR&0x02)==0){} // wait for the address to be sent
    IBSR = 0x02;            // clear IBIF
    IBCR &=~0x38;           // send STOP
    data2 = IBDR;           // capture second byte
    return (data1<<8)+data2;
}
```

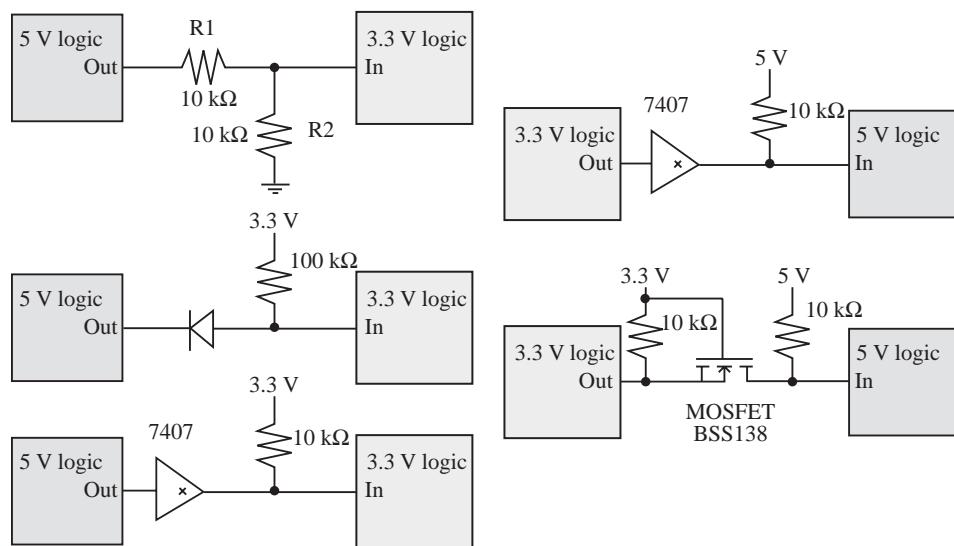
## 7.8. Logic Level Conversion

There are many 3.3 V devices we wish to interface to a 5 V microcontroller, and there are many 5 V devices we wish to interface to a 3.3 V microcontroller. This section will study various methods to convert one logic level to another. We begin with a 5 V output interfaced

to a 3.3 V input. Many 3.3 V inputs are 5 V tolerant, which means no special interface circuits are required. One of the simplest ways to convert 5 V logic into 3.3 V logic is to use a resistor divider as shown in Figure 7.54. A Schottky diode can also be used to convert 5 V into 3.3 V, and convert a 0.4 V into a 0.5 V. The Schottky diode must be fast and have a low voltage drop. The 7407 is another way to convert between logic families. When the 7407 input is 5 V, its output floats, and the 3.3 V pull-up makes a 3.3 V signal. When the 7407 input is low, its output is low.

**Figure 7.54**

Circuits to convert interface 5-V logic to 3.3-V logic.



Many 5 V inputs are 3.3 V tolerant, which means no special interface circuits are required. The 7407 can also be used to interface 3.3 V logic into 5 V logic. The  $V_{IH}$  of the 7407 is 2 V, so when the 7407 input is 3.3 V, its output floats, and the 5 V pull-up makes a 5 V signal. When the 7407 input is low, its output is low. A MOSFET, like the BSS138, is a popular method to convert logic levels because it is fast and efficient. Sparkfun makes a breakout board with resistor-divider and BSS138 circuits ([www.sparkfun.com/BOB-0874](http://www.sparkfun.com/BOB-0874)).

**Observation:** We can produce the same open collector behavior of any I/O port that has a direction register. We initialize the port by writing a zero to the data port. On subsequent accesses to the open collector port, we write the complement to the direction register. That is, if we want the I/O port bit to drive low, we set the direction register bit to 1, and if we want the I/O port bit to float (open collector), we set the direction register bit to 0.

## 7.9 Universal Serial Bus (USB)

### 7.9.1 Introduction

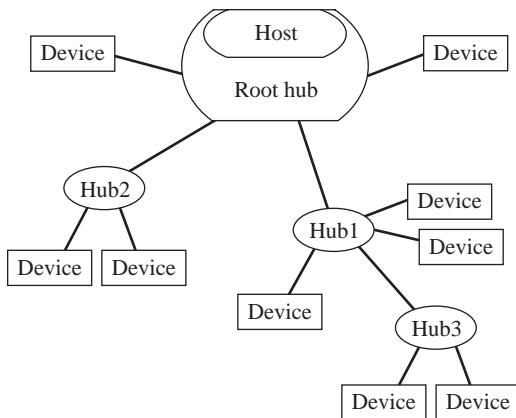
The Universal Serial Bus (USB) is a host-controlled, token-based high-speed serial network that allows communication between many devices operating at different speeds. The objective of this section is not to provide all the details required to design a USB interface; rather, it serves as an introduction to the network. There is 650-page document on the USB standard, which you can download from <http://www.usb.org>. In addition, there are quite a few web sites set up to assist USB designers, such as the one titled “USB in a NutShell” at <http://www.beyondlogic.org/usbnutshell/>. The standard is much more complex than the other networks presented in this chapter. Fortunately, however, there are a number of

USB products that facilitate incorporating USB into an embedded system. In addition, the USB controller hardware handles the low-level protocol. USB devices usually exist within the same room and are typically less than 4 meters from each other. USB 2.0 supports three speeds:

- High Speed—480 Mbits/s
- Full Speed—12 Mbits/s
- Low Speed—1.5 Mbits/s

The original USB Version 1.1 supported just full-speed mode and a low-speed mode. The Universal Serial Bus is host-controlled, which means the host regulates communication on the bus, and there can be only one host per bus. On the other hand, the On-The-Go specification, added in Version 2.0, includes a Host Negotiation Protocol that allows two devices to negotiate for the role of host. The USB host is responsible for undertaking all transactions and scheduling bandwidth. Data can be sent by various transaction methods using a token-based protocol. USB employs a tiered star topology, using a hub to connect additional devices. A hub is at the center of each star. Each wire segment is a point-to-point connection between the host and a hub or function, or a hub connected to another hub or function, as shown in Figure 7.55. Because the hub provides power, the hub can monitor power to each device by switching off a device drawing too much current without disrupting other devices. The hub can filter out high-speed and full-speed transactions so that lower-speed devices do not receive them. Because USB uses a 7-bit address, up to 127 devices can be connected.

**Figure 7.55**  
USB network topology.



The electrical specification for USB was introduced in Figure 7.18 and Table 7.5, using four shielded wires (+5V power, D+, D−, and ground). The D+ and D− are twisted-pair differential data signals. It uses **Non-Return-to-Zero Invert (NRZI)** encoding to send data with a sync field to synchronize the host and receiver clocks.

USB drivers will dynamically load and unload. When a device is plugged into the bus, the host will detect this addition, interrogate the device, and load the appropriate driver. Similarly, when the device is unplugged, the host will detect its absence and automatically unload the driver. The USB architecture comprehends four basic types of data transfers:

- **Control Transfers:** Used to configure a device at attach time and can be used for other device-specific purposes, including control of other pipes on the device.
- **Bulk Data Transfers:** Generated or consumed in relatively large quantities and have wide dynamic latitude in transmission constraints.

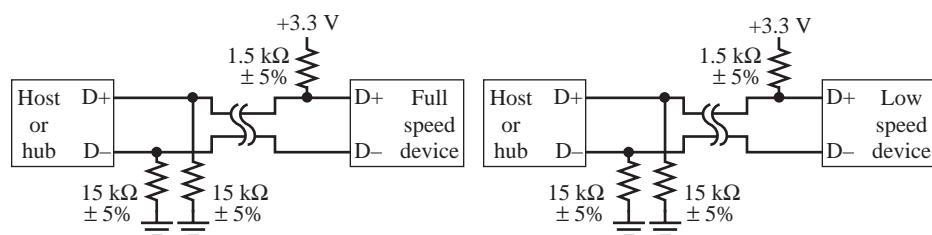
- **Interrupt Data Transfers:** Used for timely but reliable delivery of data—for example, characters or coordinates with human-perceptible echo or feedback response characteristics.
- **Isochronous Data Transfers:** Occupy a prenegotiated amount of USB bandwidth with a prenegotiated delivery latency. (Also called streaming real time transfers.)

Isochronous transfer allows a device to reserve a defined amount of bandwidth with guaranteed latency. This is appropriate for real-time applications such as in audio or video applications. An isochronous pipe is a stream pipe and is, therefore, always unidirectional. An endpoint description identifies whether a given isochronous pipe's communication flow is into or out of the host. If a device requires bidirectional isochronous communication flow, two isochronous pipes must be used, one in each direction.

A USB device indicates its speed by pulling either the D+ or D− line to 3.3 V, as shown in Figure 7.56. A pull-up resistor attached to D+ specifies full speed, and a pull-up resistor attached to D− means low speed. These device-side resistors are also used by the host or hub to detect the presence of a device connected to its port. Without a pull-up resistor, the host or hub assumes there is nothing connected. A high-speed device begins as a full-speed device (1.5 k to 3.3 V). Once it has been attached, it will do a high-speed chirp during reset and establish a high-speed connection if the hub supports it. If the device operates in high-speed mode, then the pull-up resistor is removed to balance the line.

**Figure 7.56**

Pull-up resistors on USB devices signal specify the speed.



Like most communication systems, USB is made up of several layers of protocols. Like the CAN network presented previously, the USB controllers will be responsible for establishing the low-level communication. Each USB transaction consists of three packets:

- **Token Packet** (header)
- **Optional Data Packet** (information)
- **Status Packet** (acknowledge)

The host initiates all communication, beginning with the token packet, which describes the type of transaction, the direction, the device address, and the designated endpoint. The next packet is generally a data packet carrying the information and is followed by a handshaking packet, reporting whether the data or token was received successfully, or whether the endpoint is stalled or not available to accept data. Data is transmitted least significant bit first. Some USB packets are shown in Figure 7.57. All packets must start with a sync field. The **sync** field is 8 bits long at low and full speed or 32 bits long for high speed and is used to synchronize the clock of the receiver with that of the transmitter. **PID** (Packet ID) is used

**Figure 7.57**

USB packet types.

Token packet	Sync	PID	ADDR	ENDP	CRC5	EOP
Data packet	Sync	PID	Data	CRC16	EOP	
Handshake packet	Sync	PID	EOP			
Start of frame	Sync	PID	Frame number	CRC5	EOP	

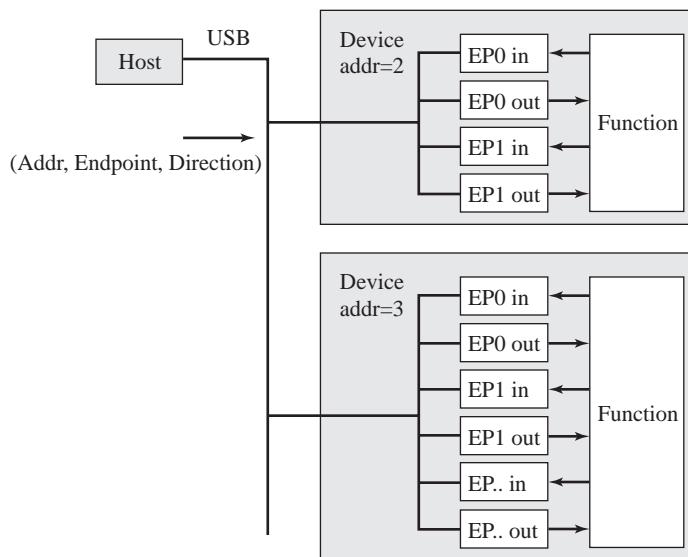
to identify the type of packet that is being sent, as shown in Table 7.21. The **address** field specifies which device the packet is designated for. Being 7 bits in length, it allows for 127 devices to be supported. Address 0 is not valid, as any device that is not yet assigned an address must respond to packets sent to address 0. The **endpoint** field is made up of 4 bits, allowing 16 possible endpoints. Low speed devices, however can have only two additional endpoints on top of the default pipe. **Cyclic Redundancy Checks** are performed on the data within the packet payload. All token packets have a 5-bit CRC, whereas data packets have a 16-bit CRC. EOP stands for **End of packet**. **Start of Frame** packets (SOF) consist of an 11-bit frame number that is sent by the host every 1ms ± 500ns on a full-speed bus or every 125 µs ± 0.0625 µs on a high-speed bus.

**Table 7.21**  
USB PID numbers.

Group	PID value	Packet identifier
Token	0001	OUT token, address + endpoint
	1001	IN token, address + endpoint
	0101	SOF token, start-of-frame marker and frame number
	1101	SETUP token, address + endpoint
Data	0011	DATA0
	1011	DATA1
	0111	DATA2 (high speed)
	1111	M DATA (high speed)
Handshake	0010	ACK handshake, receiver accepts error-free data packet
	1010	NAK handshake, device cannot accept data or cannot send data
	1110	STALL handshake, endpoint is halted or pipe request not supported
	0110	NYET (no response yet from receiver)
Special	1100	PREamble, enables downstream bus traffic to low-speed devices
	1100	ERR, split transaction error handshake
	1000	Split, high-speed split transaction token
	0100	Ping, high-speed flow control probe for a bulk/control endpoint

**USB functions** are USB devices that provide a capability or function such as a printer, zip drive, scanner, modem, or other peripheral. Most functions will have a series of buffers, typically 8 bytes long. **Endpoints** can be described as sources or sinks of data, shown as **EP0In**, **EP0Out**, and the like in Figure 7.58. As the bus is host-centric, endpoints occur at the end of the communications channel at the USB function. The host-software may send

**Figure 7.58**  
USB data flow model  
shows that the host  
communicates with  
endpoints in the device.



a packet to an endpoint buffer in a peripheral device. If the device wishes to send data to the host, the device cannot simply write to the bus, as the bus is controlled by the host. Therefore, it writes data to the endpoint buffer specified for input, and the data sits in the buffer until such time as the host sends an IN packet to that endpoint requesting the data. Endpoints can also be seen as the interface between the hardware of the function device and the firmware running on the function device.

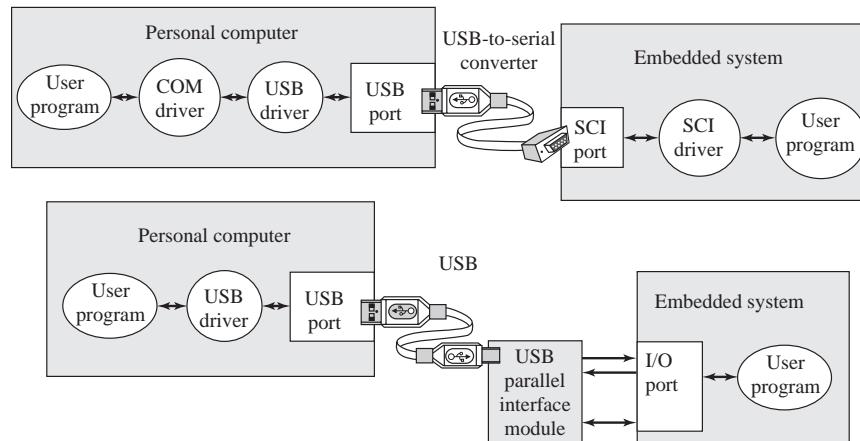
While the device sends and receives data on a series of endpoints, the client software transfers data through pipes. A **pipe** is a logical connection between host and endpoint(s). Pipes will also have a set of parameters associated with them, such as how much bandwidth is allocated to it, what transfer type (Control, Bulk, Iso, or Interrupt) it uses, a direction of data flow, and maximum packet/buffer sizes. **Stream pipes** can be used to send unformatted data. Data flows sequentially and a predefined direction, either in or out. Stream pipes will support bulk, isochronous, and interrupt transfer types. Stream pipes can be controlled by either the host or the device. **Message pipes** have a defined USB format. They are host-controlled and are initiated by a request sent from the host. Data is then transferred in the desired direction, dictated by the request. Therefore, message pipes allow data to flow in both directions but support only control transfers.

### 7.9.2 Modular USB Interface

There are two approaches to implementing a USB interface for an embedded system. In the modular approach, we will employ a USB-to-parallel or USB-to-serial converter. The modular approach is appropriate for adding USB functionality to an existing system. For about \$30, we can buy a converter cable with a USB interface to connect to the personal computer (PC) and a serial interface to connect to the embedded system, as shown in Figure 7.59. The embedded system hardware and software is standard RS232 serial. These systems come with PC device drivers so that the USB-serial-embedded system looks like a standard serial port (COM) to the PC software. The advantage of this approach is that software development on the PC and embedded system is simple. The disadvantage is that none of the power and flexibility of USB is utilized. In particular, the bandwidth is limited by the RS232 line, and the data stream is unformatted. Similar products are available that convert USB to the parallel port. Companies that make these converters include:

IOGear Inc.	<a href="http://www.iogear.com">http://www.iogear.com</a>
Wyse Technology	<a href="http://www.wyse.com">http://www.wyse.com</a>
D-Link Corporation	<a href="http://www.dlink.com">http://www.dlink.com</a>
Computer Peripheral Systems, Inc.	<a href="http://www.cpsc.com">http://www.cpsc.com</a>
Jo-Dan International, Inc.	<a href="http://www.jditech.com">http://www.jditech.com</a>

**Figure 7.59**  
Modular approach to  
USB interfacing.



The second modular approach is to purchase a USB parallel interface module. These devices allow you to send and receive data using parallel handshake protocols similar to the input/output examples in Chapter 3. They typically include an USB-enabled microcontroller and receive and transmit FIFO buffers. This approach is more flexible than the serial cable method, because both the microcontroller module and the USB drivers can be tailored and personalized. In particular, some modules allow you to burn PID and VID numbers into EEPROM. The advantages and disadvantages of this approach are similar to the serial cable, in that the data is unformatted and you will not be able to implement high bandwidth bulk transfers or negotiate for real-time bandwidth available with isochronous data transfers. Companies that make these modules include:

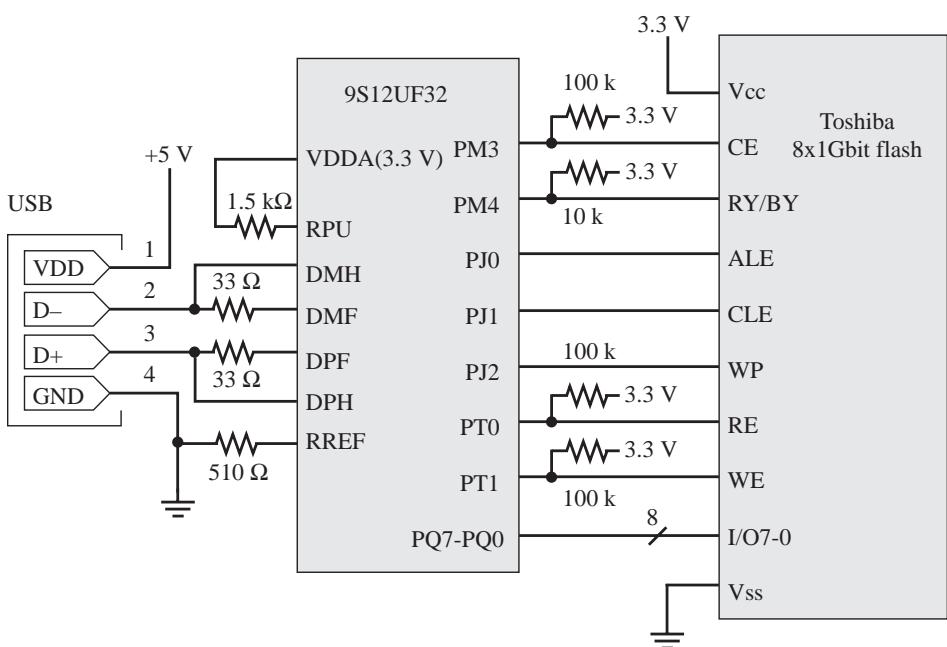
Future Technology Devices International Ltd.	<a href="http://www.ftdichip.com/">http://www.ftdichip.com/</a>
ActiveWire, Inc.	<a href="http://www.activewireinc.com">http://www.activewireinc.com</a>
DLP Design, Inc.	<a href="http://www.dlpdesign.com">http://www.dlpdesign.com</a>
Elexol Pty Ltd.	<a href="http://www.elexol.com">http://www.elexol.com</a>

### 7.9.3 Integrated USB Interface

The second approach to implementing a USB interface for an embedded system is to integrate the USB capability into the microcontroller itself. This method affords the greatest flexibility and performance, but requires careful software design on both the microcontroller and the host. During the past ten years, USB has been replacing RS232 serial communication as the preferred method for connecting embedded systems to the personal computer. Manufacturers of microcontrollers have introduced versions of their products with USB capability. Examples include the Microchip PIC18F2455, FTDI FT245BM, and the Texas Instruments LM3S3748. Figure 7.60 shows a USB-flash disk system designed using the 9S12UF32, which is capable of connecting directly to the USB bus. It has an internal voltage reference needed to create the 3.3 V levels. The flash disk is interfaced to parallel ports of the 9S12UF32. This microcontroller handles the USB 2.0 protocol, including full-speed and high-speed device functions. It has one default control IN/OUT endpoint and five independent configurable physical endpoints, with 64-byte buffers. It can perform high-speed isochronous data toggle communication. The detailed hardware and software for this system, which was developed by Freescale to demonstrate the capabilities of this chip, can be found on the Freescale web site as Reference Design “RDHCS12UF32TD.”

**Figure 7.60**

Thumbdrive system with USB interface built with a 9S12UF32 microcontroller.



## 7.10 Exercises

**7.1** For each term, give a definition in 32 words or less.

- |                 |                   |                   |
|-----------------|-------------------|-------------------|
| a) Asynchronous | h) Full-duplex    | o) NRZI           |
| b) Baud rate    | i) Frame          | p) Open collector |
| c) Bandwidth    | j) Framing error  | q) Overrun        |
| d) Break        | k) Half-duplex    | r) Positive logic |
| e) DCE          | l) Mark           | s) Simplex        |
| f) DTE          | m) Negative logic | t) Space          |
| g) Even parity  | n) NRZ            | u) Synchronous    |

**7.2** In 32 words or less, describe the similarities and differences between the following pairs of terms.

- |   |                        |
|---|------------------------|
| a) Baud rate versus bandwidth           | e) DS275 versus MAX232 |
| b) Positive logic versus negative logic | f) SCI versus SPI      |
| c) XON versus XOFF                      | g) NRZ versus NRZI     |
| d) Full-duplex versus half-duplex       | h) DTE versus DCE      |

**7.3** What fundamental electrical property is used to transfer digital data across a distance?

- |            |               |
|------------|---------------|
| a) Voltage | d) Frequency  |
| b) Current | e) Phase      |
| c) Energy  | f) Wavelength |

**7.4** Look up in the 9S12 data sheet how the SCI checks for noise. In particular, when is the noise flag set? How does software clear the noise flag?

**7.5** In 16 words or less, explain how you would use the **RAF** flag in **SCISR2** while implementing the half-duplex network in Figure 7.7.

**7.6** Draw a plot similar to Figure 7.2 for the binary data 00110111.

**7.7** Draw a plot similar to Figure 7.2 for the binary data 10011100.

**7.8** Consider a serial port operating with a baud rate of 10,000 bits per second. Draw the waveform occurring at the PS1 output (voltage levels are +5 and 0) when the ASCII ‘A’ (\$61) is transmitted on SCI0. The protocol is one start, eight data, and one stop bit. SCI0 is initially idle, and the software writes the \$61 to SCI0DRL at time = 0. Show the PS1 line before and after the frame, assuming the channel is idle before and after the frame.

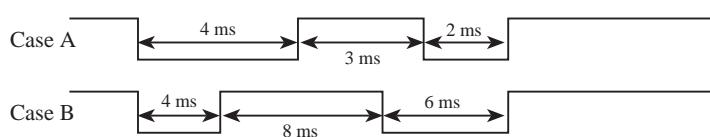
**7.9** Consider a serial port operating with a baud rate of 1000 bits per second. Draw the waveform occurring at the PS3 output (voltage levels are +5 and 0) when the ASCII ‘B’ (\$42) is transmitted on SCI1. The protocol is one start, eight data, and one stop bit. SCI1 is initially idle, and the software writes the \$42 to SCI1DRL at time = 0. Show the PS3 line before and after the frame, assuming the channel is idle before and after the frame.

**7.10** Assume the SCI baud rate is 1000 bits/sec and the protocol is one start, eight data, no parity, and one stop bit. What is the channel bandwidth? If we used two stop bits instead of one without changing the baud rate, would the bandwidth be higher, lower or the same?

**7.11** The data in Figure 7.61 was measured on a PS0 serial input, which we think is one frame, but it might be two frames. The serial format is one start, eight bit, and one stop bit.

- What is the baud rate in Case A?
- What data is being transferred in Case A? Give the number(s) in hexadecimal.
- What is the baud rate in Case B?
- What data is being transferred in Case B? Give the number(s) in hexadecimal.

**Figure 7.61**  
Serial transmission for  
Exercise 7.11.

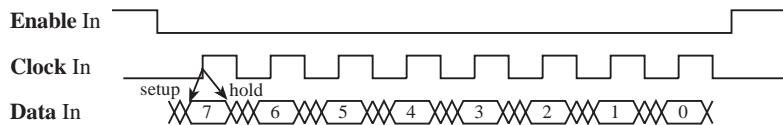


**7.12** RDRF interrupts are armed so that interrupts occur when new data arrives into the 9S12. Consider the situation in which a FIFO queue is used to buffer data between the RDRF ISR and the main program. The **SCIhandler** reads **SCIDRL** and saves the data by calling **Fifo\_Put**. When the main program wants input it calls **SCI\_InChar**, which in turn calls **Fifo\_Get**. Experimental observations show this FIFO is usually empty, and has at most three elements. What does it mean?

- a) The system is CPU bound
- b) Bandwidth could be increased by increasing FIFO size
- c) The system is I/O bound
- d) The FIFO could be replaced by a global variable
- e) The latency is small and bounded
- f) Interrupts are not needed in this system

**7.13** A slave device will be interfaced to the Master 9S12 using SPI. The timing is shown in Figure 7.62. There are three signals that will be outputs of the 9S12 and inputs to the device (**Enable**, **Clock**, and **Data**). The timing of the external device is shown below. What CPHA, CPOL mode should you use?

**Figure 7.62**  
Serial transmission for Exercise 7.13.



**7.14** Consider how long it will take Program 7.7 to output to the 74HC595. The **brclr**, **staa**, **brclr ldaa**, and **rts** instructions require 5, 3, 5, 3, 5 cycles to execute respectively. Why will Program 7.7 require more than  $5 + 3 + 5 + 3 + 5 = 21$  cycles to actually execute? Assuming a 4 MHz E clock, approximately how long will it take?

**D7.15** The objective is to build the serial port receive function using input capture and output compare (i.e., the signal is attached to PT1, not Port S). Assume the baud rate is 9600 bits/sec, resulting in a bit time of 104  $\mu$ s. Similar to SCI, the protocol is one start, eight data, no parity, and one stop bit. The TTL level serial input is connected to the input capture pin PT1. You can use output compare 2 channel as well. You may use the FIFO functions in Program 4.14 without showing their implementations. There will be three global error flags. Your initialization will clear the flags, and your communication programs will set the flags on the corresponding error:

<b>FalseStart</b>	set if there is a start edge, but the start bit is not zero
<b>FrameError</b>	set if the stop bit is not one
<b>Overrun</b>	set if the FIFO becomes full

The user program, which you do not write, can check and clear these error flags. Your solution will include an input function **Serial\_InChar**, virtually identical to **SCI\_InChar** in Program 7.1. It is a requirement that your input capture/output compare interrupt handlers accept the one start, eight data, and one stop bit transmission without executing any blind waits or backward jumps.

- a) Carefully draw the input signal when the ASCII character '5' is received. Draw the signal to scale. Assume the TCNT value is 1000 (decimal) when the first 1 to 0 transition occurs. Give the TCNT value for each transition of the input signal. Place arrows ( $\uparrow$ ) on the drawing exactly when each IC1 and OC2 interrupt will occur, giving the TCNT value at each interrupt.
- b) In addition to the three public global error flags, define any private global variables you need.
- c) Show the ritual that initializes variables, input capture, and output compare.
- d) Show the input capture 1 and output compare 2 interrupt handlers.
- e) Show the **Serial\_InChar** function with which the user can receive data.

**D7.16** The objective is to build the serial port transmit function using output compare (i.e., the signal is attached to PT0, not Port S). Assume the baud rate is 9600 bits/sec, resulting in a bit time of 104  $\mu$ s. Similar to SCI, the protocol is one start, eight data, no parity, and one stop bit. Make the TTL level serial output on PT0. You may use the FIFO functions in Program 4.14 without

showing their implementations. There will be no error flags. Your solution will include an output function `Serial_OutChar`, which operates similar to `SCI_OutChar` in Program 7.1. It is a requirement that your output compare interrupt handler transmits the one start, eight data, one stop bit frame without executing any blind waits or backward jumps.

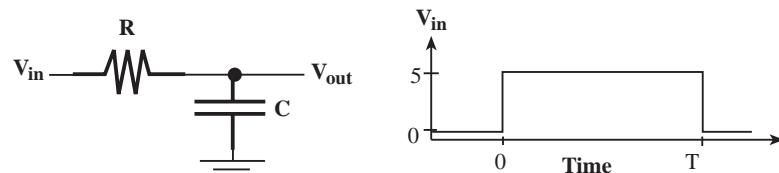
- Carefully draw the output signal when the ASCII character ‘5’ is transmitted. Draw the signal to scale. Assume the TCNT value is 1000 (decimal) when the first 1 to 0 transition occurs. Give the TCNT value for each transition of the output signal. Place arrows ( $\uparrow$ ) on the drawing exactly when each OC0 interrupt will occur, giving the TCNT value at each interrupt.
- Define any private global variables you need.
- Show the ritual that initializes variables and output compare.
- Show the OC0 interrupt handler.
- Show the `Serial_OutChar` function with which the user can transmit data.

- D.7.17** Interface an 8-bit serial device using SPI. The **control** pin is interfaced to PT2, and the **clock** and **data** signals are connected to the SPI. The **control** and **clock** signals are normally high. The SPI needs to clock data out on the falling edge of the clock. The **clock** and **data** are created by the real SPI, and the **control** signal is bit-banged. The maximum clock frequency is 1 MHz.
- Write a function that initializes the interface.
  - Write a function that outputs (transmits) one byte using the SPI port. First send eight bits of data, then make PT2 low, then make PT2 high again.

- D.7.18** This problem addresses the issue of capacitive loading on a high-speed serial transmission line like SPI. The SPI ports of two 9S12s are connected with a VERY long cable. We will model this cable as a single resistor in series with a capacitor, as shown in Figure 7.63. For this question, assume an ideal transmitter (output impedance of 0) and an ideal receiver (input impedance of infinity). Let the resistance **R** be  $1 \Omega$ , and the capacitance **C** be  $10 \text{ nF}$ . Consider a 5-V 100-ns pulse ( $T = 100 \text{ ns}$ ) on the output of the transmitter (labeled as  $V_{in}$ ) as might occur with a 5-Mbps SPI transmission. Derive an equation for  $V_{out}$  as a function of time for the first 100 ns. Show your work. Calculate values for  $V_{out}$  at time equals 0 and time equals  $100 \text{ ns}$ . Create a  $V_{in}$  versus time sketch similar to the right of Figure 7.63, and add a sketch of  $V_{out}$  on this same plot. Show both 0 to time  $T$  and time  $T$  to time  $2T$ .

**Figure 7.63**

Circuit model for Exercises D7.18 and D7.19.

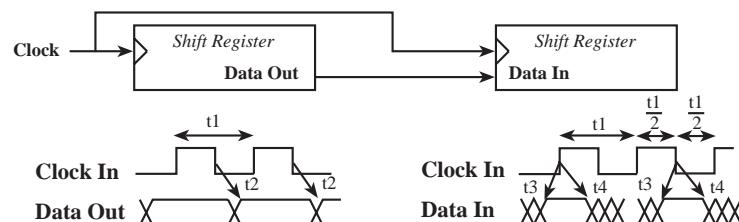


- D.7.19** Solve exercise D7.18 with  $R = 10 \Omega$ ,  $C = 100 \text{ nF}$  and  $T = 10 \text{ ns}$ . Justify whether or not this system will work?

- D7.20** The output of one shift register is connected to the input of a second shift register, as shown in Figure 7.64. The two registers are controlled by the same 50% duty cycle **Clock**. The period of the clock is  $t_1$ . The data is shifted out on the falling edge of **Clock**. The time  $t_2 = [10 \text{ ns min}, 200 \text{ ns max}]$  is the delay from the falling edge of **Clock** until when the output is valid. The data is shifted into the second register on the rising edge of **Clock**. The time  $t_3 = 50 \text{ ns}$  is the time before the rising edge of **Clock** that the data must be valid. The time  $t_4 = 20 \text{ ns}$  is the time after that same rising edge of **Clock** that the data must continue to be valid. What is the smallest  $t_1$  clock period that reliably transverses data from one shift register to the other?

**Figure 7.64**

Circuit and timing for Exercise D7.20.



**D7.21** Design an automatic baud rate selector. The input capture system will be used to measure the first three positive logic pulse widths and the first three negative logic widths. To implement this problem on another computer, simply connect to an available input capture. Your software will then choose the smallest pulse width as the communication bit time. For example, if the smallest bit time is 400 E cycles (which falls between 625 and 313), then the baud rate should be 4800. Finally, your software will initialize the SCI to communicate at that correct baud rate (eight bit data, one start, one stop and no parity). Specify the E clock rate. You may assume the baud rate is 600, 1200, 2400, 4800, or 9600.

- Show the hardware connections between a three-wire  $\pm 12$  V RS232 full-duplex line (TxD, RxD, and Gnd) and the microcomputer. Any RS232 interface chip may be used. Give chip numbers but not pin numbers.
- Show the appropriate data structure that holds the information of the above table. The organization of this information should be easy to understand and easy to change (e.g., add more baud rates, change of E clock frequency). This data structure will eliminate the need for a complex `if then` or `switch` code in Part c.
- Show the C software function that automatically selects the correct baud rate and initializes the SCI. Gadfly synchronization will be used. Input/output functions for the SCI are not required.

**D7.22** You will build a low-cost computer-based logic analyzer. It will be able to collect 2048 8-bit digital samples at 80 MHz triggered by your software. You are given the following hardware components (you may use other 74HC digital devices, too).

A 20-MHz clock generator (black box with 50% duty cycle digital output)  
A 2048 9-bit hardware FIFO (CY7C429, IDT7203, AM7203, or LH540203)

The FIFO contains a 2048 by 9 bit RAM and implements the classic FIFO functions. You will only use eight of the nine data pins. To reset the FIFO, you pull its **\*RS** line low. This will clear the FIFO. When the reset **\*RS** is high, you can perform a put operation by toggling the **\*W** line low, then high. The 9 bits on the input data lines **D8-D0** are stored (put) in the FIFO on the rising edge of the **\*W** line. When the reset **\*RS** is high, you can perform a get operation by toggling the **\*R** line low, then high. The 9 bits are removed (get) from the FIFO on the rising edge of the **\*R** line. These 9 bits are available on the output data lines **O8-O0** when the **\*R** line is low. There are three negative logic FIFO status outputs that are available:

full flag	<b>*FF</b>	0 means full
half flag	<b>*HF</b>	0 means more than half full
empty flag	<b>*EF</b>	0 means empty

- Show the hardware connections to your single-chip microcomputer. A RS232 channel will connect your logic analyzer system to a personal computer. The only external connections to your logic analyzer system are the eight FIFO data inputs. For some cool features that are possible with this approach, see the Circuit Cellar, *INK*, Vol. 89, December 1997, pp. 46–49. Other features discussed in this article, but not to be implemented here, are computer-controlled sampling rate, optional external clock, and external reset to the device under test so that the FIFO and external circuit are started together.
- Show the main program that initializes the SCI to 9600 baud, one stop, no parity, then loops:
  - Resets the FIFO
  - Waits for any character to be received on the SCI input (start command from the PC)
  - Allows the FIFO to fill with 8-bit data at 20 MHz
  - Waits for the FIFO to be full
  - Reads all 2048 bytes from the FIFO and transmits them out the SCI output one at a time

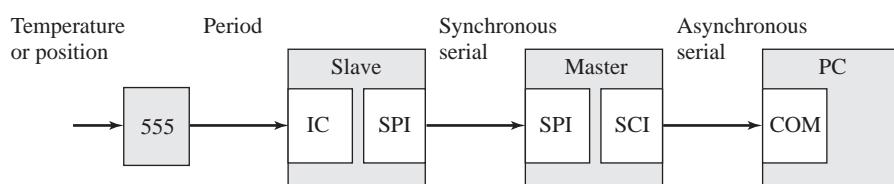
This computer is dedicated to this task, so you must show all the software for the computer. The program never quits, repeats the loop over and over. Use gadfly or interrupt synchronization, whichever is most appropriate.

## 7.11 Lab Assignments

**Lab 7.1** The overall objective is to redesign the Lab 6.2 or 6.3 data acquisition system to employ two microcontrollers. The slave microcontroller will perform the data acquisition (position or temperature), and the master microcontroller will interface to the PC. The two microcontrollers will be linked using their SPI ports, as shown in Figure 7.65. The master microcontroller will fetch data from the slave and transmit it to the PC, using SCI interrupt synchronization. The user will interact with a PC running HyperTerminal.

**Figure 7.65**

Data flow graph for Lab 7.1



**Lab 7.2** The overall goal is to develop an interrupting SCI device driver that implements fixed-point input/output. The fixed-point constant is 0.001. The full-scale range is from 0 to 65.534. The Fix\_InDec function should provide for flexible operation. The input/output specifications of Fix\_InDec are illustrated in Table 7.22. The operator creates the input by typing on the keyboard, and the output is a number passed back as the return parameter of the function.

**Table 7.22**

Examples of 16-bit unsigned decimal fixed point input with  $\Delta=10^{-3}$ .

User Types in to Fix_InDec	Actual Value	Return Parameter of Fix_InDec()
0	0.000	0
.2	0.200	200
50.5	50.500	50500
12.48	12.480	12480
0.0023	0.002	2
1.4595	1.460	1460
1.4604	1.460	1460
6	6.000	6000
65.534	65.534	65534

Notice that Fix\_InDec rounds the input to the closest fixed-point result (e.g., 1.4595 rounds to 1.460, and 1.4604 also rounds to 1.460). Some numbers such as 1.2345678 might be considered illegal because they cause overflow of intermediate results. In the comments of your software, please discuss why you chose your particular implementation method over the other available choices. Please handle the backspace character, allowing the operator to erase characters. In particular, you are free to use iterative or recursive algorithms. You are free to modify the prototypes as well as handle illegal inputs in any way you feel is appropriate. You must detect illegal input, but you have a choice as to how your system responds to the illegal input. One possibility for handling an illegal number would be to return 65535, which you could define as an illegal number. A second possibility for when an illegal number is typed is to output an error message, and require the operator to enter the number again. The input/output specifications for Fix\_OutDec are illustrated in Table 7.23.

**Table 7.23**

Examples of 16-bit unsigned decimal fixed-point output with  $\Delta=10^{-3}$ .

Input to Fix_OutDec	Output of Fix_OutDec
\$0000 0	0.000
\$0032 50	0.050
\$00FA 250	0.250
\$0781 1921	1.921
\$3039 12345	12.345
\$7FFF 32767	32.767
\$FFFE 65534	65.534

**Lab 7.3** The overall goal is to interface a DS1620 temperature controller to the 9S12. Create the following functions to control the device. When reading the temperature, you will have to start a conversion and wait for the conversion to be complete

```
void DS1620_Init(void); // initializes SPI connections to the 9S12  
void DS1620_WriteTH(short data); // sets the high setpoint temp  
void DS1620_WriteTL(short data); // sets the low setpoint temp  
unsigned short DS1620_ReadT(void); // reads the current temp
```

Perform experiments to determine the accuracy of the temperature measurement. Interface three LEDs to TH TL TCOM outputs, and perform experiments to verify the temperature controller in the DS1620 is operating properly.

**Lab 7.4** The overall objective of this lab is to design, implement, and test an output port expander. You will use three I/O pins of the 9S12 and four 74HC595 shift registers. You will design hardware and software that supports four 8-bit output ports. The output ports do not need to be readable. Measure how long it takes for the 9S12 to perform outputs to all 32 bits.

**Lab 7.5** The overall objective of this lab is to design, implement, and test an input port expander. You will use three I/O pins of the 9S12 and four 74HC165 shift registers. You will design hardware and software that supports four 8-bit input ports. The input ports do not need to be latched by an external signal. Measure how long it takes for the 9S12 to perform inputs from all 32 bits.

**Lab 7.6** The objective of this lab is to design a digital clock using a DS1307 external clock chip. The first step is to interface the clock chip to the microcontroller using an I<sup>2</sup>C network. The second step is to design low-level drivers to allow the microcontroller to send and receive data from the DS1307. The next software layer includes functions such as **SetTime**, **FormatTime**, and **ReadTime**. The highest level is a main program that implements a digital clock using an LED or LCD display. Two, three, or four momentary switches will be used to control the operation of the digital clock.

**Lab 7.7** The objective of this lab is to design a digital thermometer using a DS1631A external thermometer chip. The first step is to interface the thermometer chip to the microcontroller using an I<sup>2</sup>C network. The second step is to design low-level drivers to allow the microcontroller to send and receive data from the DS1631A. The next software layer includes functions such as **SetMode** and **ReadTemperature**. The highest level is a main program that implements a digital thermometer using a LED or LCD display. Two or three momentary switches will be used to control the operation of the digital thermometer.

# 8 Parallel Port Interfaces

## Chapter 8 objectives are to:

- ❖ Develop fundamental concepts associated with the physical behavior of the device
- ❖ Design the hardware interface between the device and the parallel port
- ❖ Write low-level device drivers that perform basic I/O with our device
- ❖ Discuss interfacing issues associated with performing the I/O as a background interrupt process

**C**hapter 8 deals with external devices that we connect to the parallel I/O port of our computer. In particular, we will interface switches, keyboards, single LEDs, LED displays, single LCDs, LCD displays, relays, and motors. Starting with this chapter and running through Chapter 11, we make the shift away from the details of the microcomputer itself and discuss devices external to the computer. Later, in Chapters 12 through 14, the pieces (microcomputer and external devices) are combined to design embedded systems.

## 8.1 Input Switches and Keyboards

### 8.1.1 Interfacing a Switch to the Computer

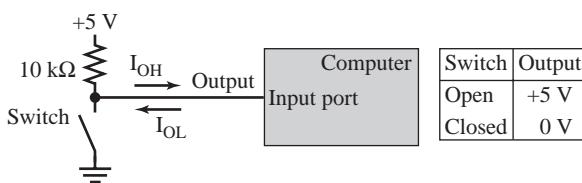
We begin with the simple switch interface (Figure 8.1). To convert the mechanical signal into an electric signal, a resistor pull-up is used. When the switch is open, the output is pulled to +5 V. The amount of current this output can source (its equivalent  $I_{OH}$ ) is determined by the resistor value. The smaller the resistor, the larger the  $I_{OH}$ . When the switch is closed, the output is forced to ground. The amount of current this output can sink (its equivalent  $I_{OL}$ ) is huge, limited only by the capacity of the switch to conduct current. The resistor does not affect the equivalent  $I_{OL}$ . A smaller resistor does waste current when the switch contact is closed.

In TTL digital logic we usually pull-up to +5 V rather than pulling down to zero because in most situations  $I_{OL}$  needs to be larger than  $I_{OH}$ . CMOS digital logic is an exception to this rule. Either pull-up or pull-down could be used for CMOS. When the switch in the pull-down circuit of Figure 8.2 is open, the output is pulled to ground. The amount of current this output can sink (its equivalent  $I_{OL}$ ) is determined by the resistor value. The smaller the resistor, the larger the  $I_{OL}$ . When the switch is closed, the output is forced to +5 V. The amount of current this output can source (its equivalent  $I_{OH}$ ) is huge, limited only by the capacity of the switch to conduct current. The resistor does not affect the equivalent  $I_{OH}$ . Notice that the polarity of the interface is reversed in the pull-down interface as compared to the pull-up case.

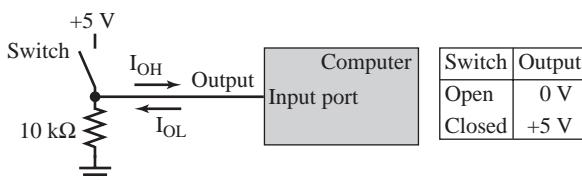
Most ports on the 9S12 support both internal pull-ups and pull-downs (see Table 3.3). To use internal pull registers, the pin must be selected as an input. To use Port AD as a digital port, the corresponding bits in the ATDDIEN must be set. Positive or negative

**Figure 8.1**

A negative logic switch interface.

**Figure 8.2**

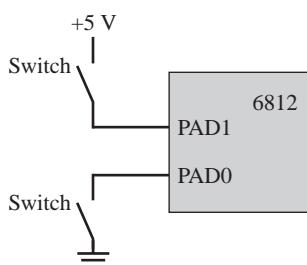
A positive logic switch interface.



logic switch interfaces can be implemented on the 9S12 without a resistor, as shown in Figure 8.3, where PAD1 is configured to have a pull-down resistor, and PAD0 is configured to have a pull-up resistor.

**Figure 8.3**

The MC9S12C32 supports internal pull-up or pull-down.



The software initialization sets bits in the Port Pull Select Register (e.g., PPSAD, PPSJ, PPSP, PPSM, PPSS, PPST) to select pull-up or pull-down. For each port pin that is enabled for pull-up or pull-down, the corresponding bit in the Port Pull Select Register determines whether it is pull-up(0) or pull-down(1). For each of these port pins there is a bit in the Pull Enable Register (e.g., PERAD, PERJ, PERP, PERM, PERS, PERT), which we set to enable the pull-up or pull-down function. It is good programming practice to first set the PPSx register, then set the PERx register so that temporary glitches are avoided. Program 8.1 will initialize Port AD with pull-up on PAD0 and pull-down on PAD1, as needed for the interface shown in Figure 8.3.

### Program 8.1

The MC9S12C32 Port AD initialization.

```
// MC9S12C32
// PortAD bit 1 is connected to a switch to +5, using internal pull-down
// PortAD bit 0 is connected to a switch to 0, using internal pull-up
void PortAD_Init(void){
    ATDDIEN |= 0x03; // PAD1-0 digital I/O
    DDRAD &= ~0x03; // PAD1-0 inputs
    PPSAD |= 0x02; // pull-down on PAD1
    PPSAD &= ~0x01; // pull-up on PAD0
    PERAD |= 0x03; // enable pull-up and pull-down
}
```

**Checkpoint 8.1:** Write the software that configures all of Port M as inputs with internal pull-ups.

### 8.1.2 Hardware Debouncing Using a Capacitor

The mechanical properties of a switch strongly affect the mechanical response to an applied force according to the following second-order differential equation. The three terms arise from Newton's Second Law internia, friction, and the spring constant, respectively.

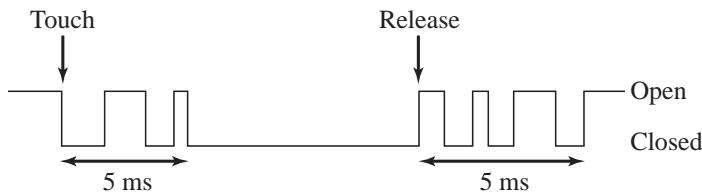
$$F = m \frac{d^2x}{dt^2} + K_d \frac{dx}{dt} + Kx$$

where  $m$  is the switch mass,  $K_d$  is the friction coefficient, and  $K$  is the spring constant. The damping ratio can be calculated as

$$\xi = \frac{K_d}{2\sqrt{K \cdot m}}$$

Depending on the relative values for  $m$ ,  $K_d$ , and  $K$ , the step response (i.e., what happens when you put your finger on the button) will be underdamped ( $\xi > 1$  causes ringing) or overdamped ( $\xi < 1$  causes a delayed but smooth rise). Most inexpensive switches are underdamped. Hence, they will bounce both when touched and when released. Typical bounce times range from 1 to 25 ms. Ideally, the switch resistance is zero (actually about  $0.1 \Omega$ ) when closed and infinite when open (Figure 8.4).

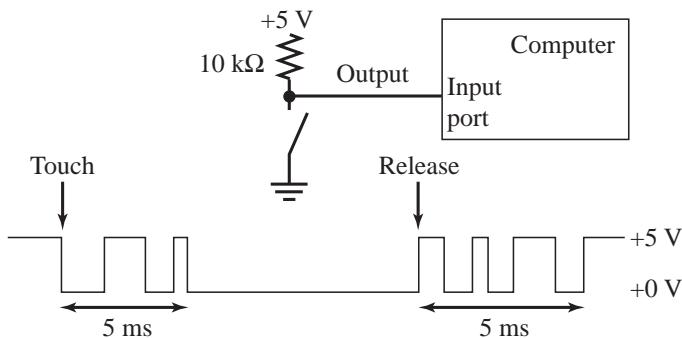
**Figure 8.4**  
Switch timing showing bounce on touch and release.



**Checkpoint 8.2:** What is the natural solution to the above differential equation with no friction ( $K_d = 0$ )?

More expensive switches can be purchased with proper damping. These oil-filled switches do not bounce. Because it is easy to add software to solve the bounce problem, most keyboard devices use inexpensive switches that bounce. With the circuit having just a pull-up resistor, the electric output will bounce because the mechanical input bounces (Figure 8.5).

**Figure 8.5**  
Switch bounce can be seen on the voltage signal.

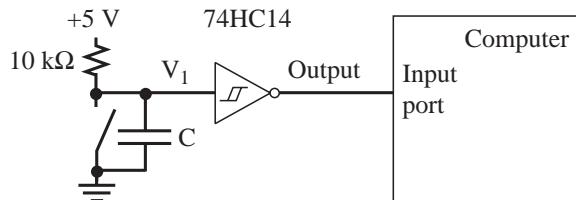


It may or may not be important to debounce the switch. For example, if we are counting the number of times the operator pushes the button, then we must debounce so that one push results in incrementing just once. We will debounce our standard computer keyboard so that when the operator types the letter A into her word processor, only one A is entered into the file. On the other hand, if the switch position specifies some static condition, and the operator sets the switch before she turns on the computer, then debouncing is not necessary.

We will study both hardware and software mechanisms to debounce the switch. The simplest hardware method is to use a capacitor across the switch (Figure 8.6). The capacitor value is chosen large enough so that the input voltage does not exceed the 0.7-V threshold of the NOT gate while it is bouncing. A detailed discussion about selecting the capacitor value will be presented later in this section.

**Figure 8.6**

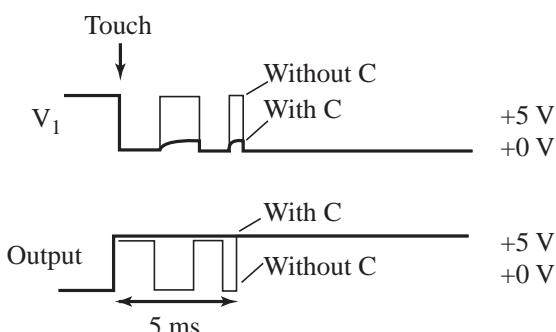
Switch bounce removed with a capacitor (this is a bad circuit).



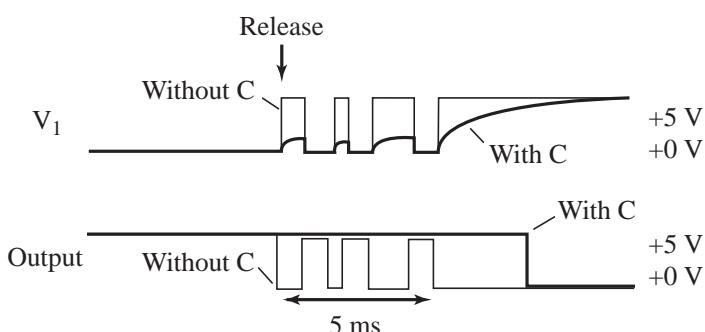
The touch timing with and without the capacitor is as shown in Figure 8.7. Notice that there is no delay between the touching of the switch and the transition of the signal output. The release timing with and without the capacitor is shown in Figure 8.8. There is a significant delay from the release of the switch until the fall of the output. To repeat, the capacitor is chosen such that the input voltage does not exceed threshold during the bouncing (Figure 8.9).

**Figure 8.7**

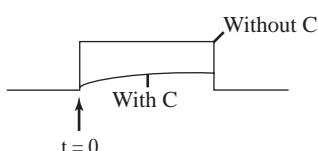
Switch touch bounce is removed by the capacitor.

**Figure 8.8**

Switch release bounce is also removed by the capacitor.

**Figure 8.9**

Timing used to calculate capacitor value.



The voltage rise during a bounce interval when the switch is open is given by

$$V \geq 5 - 5e^{-t/RC}$$

In this example,  $R = 10 \text{ k}\Omega$ , and the bounce time is 5 ms. Thus, we will choose  $C$  such that

$$0.7 \geq 5 - 5e^{-5 \text{ ms}/(10 \text{ k}\Omega \cdot C)}$$

$$0.86 \leq e^{-5 \text{ ms}/(10 \text{ k}\Omega \cdot C)}$$

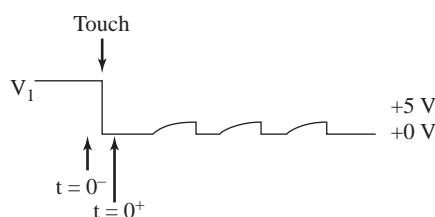
$$\ln(1.16) \geq 5 \text{ ms}/(10 \text{ k}\Omega \cdot C)$$

$$C \geq 5 \text{ ms}/(10 \text{ k}\Omega \cdot \ln(1.16)) = 3.3 \mu\text{F}$$

One problem with the above interface is the instantaneous current that occurs when the switch bounces closed. At  $t = 0^-$ , there is a charge on the capacitor. At  $t = 0^+$ , the energy has been discharged and the voltage is zero. Theoretically, this can occur only if the current at  $t = 0$  is infinite (Figure 8.10).

**Figure 8.10**

A spark will occur because a large current occurs when the switch is touched.

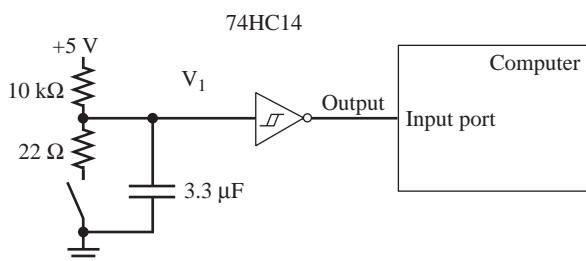


In reality, the current is not infinite, but it is large enough to cause a spark. These sparks will produce carbon deposits on the switch that will build up until the switch no longer works. To limit the current (thereby eliminating the sparks), a  $22 \Omega$  resistor is placed in series with the switch. The value  $22 \Omega$  was chosen to be much smaller than the  $10 \text{ k}\Omega$ , but much larger than the  $0.1 \Omega$  contact resistance of the switch.

If an input switch is closed, its resistance will be about  $0.1 \Omega$ , and the output of the 74HC14 will be high (a logic 1). If an input switch is open, its resistance will be infinite, and the output of the 74HC14 will be low (a logic 0). If you connect the output of this switch interface to a computer parallel port input, then the computer can read the state (on/off) of the switch (Figure 8.11).

**Figure 8.11**

A hardware interface that removes the bounce (good circuit compared to Figure 8.6).

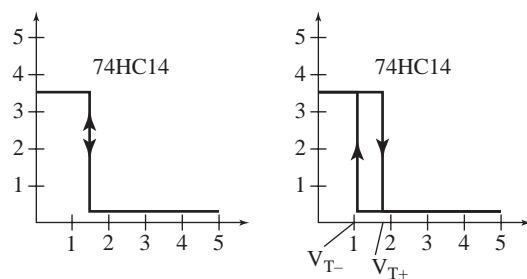


The Schmitt trigger 74HC14 is used to prevent multiple transitions when the switch is released. The difference between a regular NOT gate (74HC04) and the Schmitt trigger NOT gate (74HC14) is hysteresis (Figure 8.12). Hysteresis is required because the input of the NOT gate in the transition range is

$$0.7 \leq V_1 < 2.0 \text{ V}$$

for a long period. Normally the time during which a digital input is in transition is on the order of a few nanoseconds. The rise and fall time for a 24-MHz 9S12 E clock is on the order

**Figure 8.12**  
Input/output relationship showing the difference between a 74HC04 and a 74HC14.



of 1 ns. For faster clocks, the transition times become even shorter. But in this application because of the  $3.3 \mu\text{F}$  capacitor, the time in the transition range,  $\Delta t$ , is a huge 12 ms! In particular,

$$\Delta t = t_2 - t_1$$

where

$$0.7 = 5 - 5 e^{-t_1/(10 \text{ k}\Omega \cdot 3.3 \mu\text{F})}$$

and

$$2.0 = 5 - 5 e^{-t_2/(10 \text{ k}\Omega \cdot 3.3 \mu\text{F})}$$

Thus

$$t_1 = 10 \text{ k}\Omega \cdot 3.3 \mu\text{F} \cdot \ln(1.16) = 5 \text{ ms}$$

and

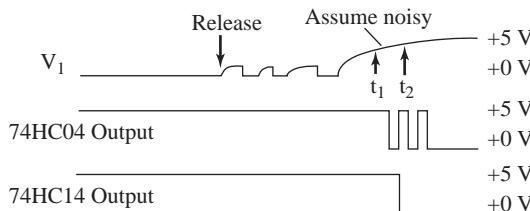
$$t_2 = 10 \text{ k}\Omega \cdot 3.3 \mu\text{F} \cdot \ln(1.67) = 17 \text{ ms}$$

Thus

$$\Delta t = 12 \text{ ms}$$

If the input stays in the transition range for a long time with just a little noise, then the output of a regular digital gate will toggle with the noise. The hysteresis removes the extra transitions that might occur with a regular gate (Figure 8.13).

**Figure 8.13**  
Timing showing why a 74HC14 is used instead of a regular digital gate.



**Observation:** With a capacitor-based debounced switch, there is no delay between the closing of the switch and the rising edge at the computer input.

**Observation:** With a capacitor-based debounced switch, there is a large delay (over 10 ms) between the opening of the switch and the falling edge at the computer input.

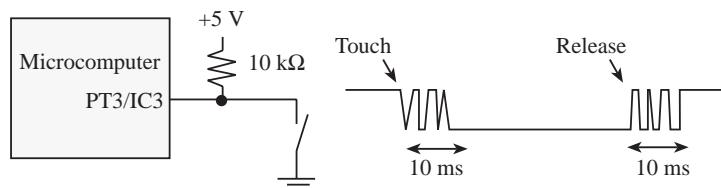
**Checkpoint 8.3:** Is the time duration of the bounce a function of the value of the pull-up resistor?

### 8.1.3 Software Debouncing

It will be less expensive to remove the bounce using software methods. It is appropriate to use a software approach because the software is fast compared to the bounce time. Typically we use the pull-up resistor to convert the switch position into a TTL-level digital signal. The 9S12 supports internal pull-up resistors, which can reduce the component count and simplify manufacturing. The 9S12 key wake-up could have been used in place of the input capture (Figure 8.14).

**Figure 8.14**

Switch interface.



In these examples, we assume switch bounce is less than 10 ms.

**Example 8.1** Given the switch circuit in Figure 8.14, write two functions: one that waits for the switch to be pressed, and the other that waits for the switch to be released. The solution should handle switch bounce.

**Solution** We will implement software debouncing using a combination of busy-wait and blind-cycle synchronization (Program 8.2 and Figure 8.15). This approach is appropriate when the computer is dedicated to this interface and does not need to perform any other functions while the routines are running. The initialization makes the pin an input and

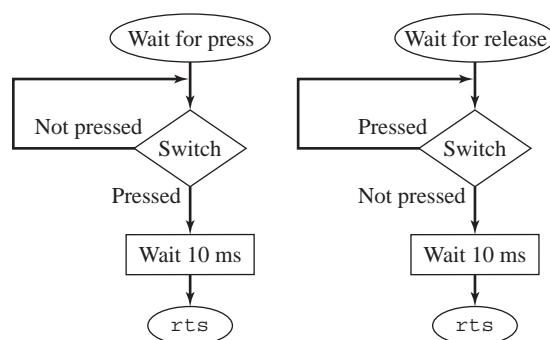
<pre> Switch_Init     jsr Timer_Init ;enable TCNT     bclr DDRT,#\$08 ;PT3 is input     rts Switch_WaitPress     brset PTT,#\$08,* ; Loop here until switch is pressed     ldy #1     jsr Timer_Wait10ms     ; wait for switch to stop bouncing     rts Switch_WaitRelease     brclr PTT,#\$08,* ; Loop here until switch is released     ldy #1     jsr Timer_Wait10ms     ; wait for switch to stop bouncing     rts </pre>	<pre> void Switch_Init(void) {     Timer_Init(); // enable TCNT     DDRT &amp;= ~0x08; // PT3 is input } void Switch_WaitPress(void){     while((PTT&amp;0x08){};     // Loop here until switch is pressed     Timer_Wait10ms(1);     // wait for switch to stop bouncing }  void Switch_WaitRelease(void){     while((PTT&amp;0x08)==0){};     // Loop here until switch is released     Timer_Wait10ms(1);     // wait for switch to stop bouncing } </pre>
---	---

**Program 8.2**

Switch debouncing using software.

**Figure 8.15**

Software flowcharts for debouncing the switch.



enables the timer (see Program 2.10). Basically, the software first waits for the appropriate edge, then waits an additional 10 ms until the bouncing interval is over.

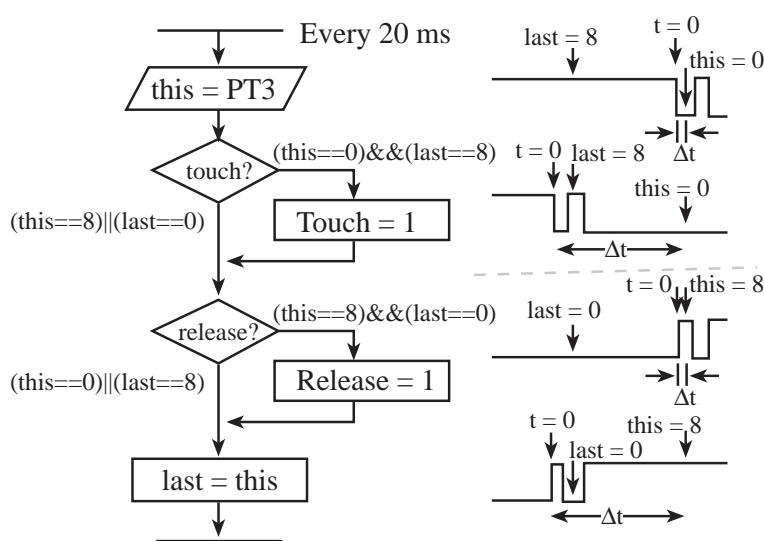
**Performance Tip:** When building complex systems, removing backward jumps often creates a more efficient and effective system.

**Example 8.2** Given the switch circuit in Figure 8.14, create two semaphores: one for touch and one for release. Signal the corresponding semaphores each time the switch is pressed and released. The solution should handle switch bounce and run in the background as an interrupt service routine.

**Solution** Basically, there will be two shared global variables: `Touch` and `Release`. Whenever the switch is touched, the semaphore `Touch` will be set to 1 (signaled). Similarly, whenever the switch is released, the semaphore `Release` will be set to 1 (signaled). The user program, not shown here, will wait on these semaphores, clearing them to 0. Whenever we pass parameters from the foreground to/from the background using shared global variables, we need to define the variables as `volatile`. This tells the compiler the value of the variable can be changed outside the immediate scope of the executing software. For example, an interrupt can occur changing the value.

This solution uses periodic polling to debounce the switch, as described in Figure 8.16. Similar to the last example, we expect the bounce to last up to 10 ms. The input signal oscillates (see Figure 8.14) when the switch is touched or released. In this solution, we will use an output-compare periodic interrupt to observe the switch every 20 ms. Any period longer than 10 ms would be okay. Since the time difference between when we look at the switch is larger than 10 ms, we are guaranteed to observe the bouncing event no more than once per touch or once per release. Fortunately, if we observe the switch during the time it is bouncing, it does not matter whether the input we observe is high or low. In other words, the sequence of digital values as seen by the periodic ISR during a touch will have exactly one transition from 1 to 0 (e.g., ...1111100000...), where the 1 to 0 change occurs after the touch. Similarly, the sequence of digital values as seen by the ISR during a release will have exactly one transition from 0 to 1 (e.g., ...0000011111...). A periodic interrupt every 20 ms will properly debounce the switch, but it will introduce a maximum delay of 30 ms, from when the switch is actually touched until the time the semaphore is signaled. The latency is labeled as  $\Delta t$  in Figure 8.16. To see how it might have a latency of 0 to 30 ms, define the time the switch is touched as  $t = 0$ . The switch bounces for 10 ms after the touch. In the best case, the periodic interrupt occurs right after  $t = 0$ , and the input is zero. The semaphore is set almost immediately after the touch. In the worst case however, if the periodic interrupt

**Figure 8.16**  
Periodic polling used to debounce a switch.



happens near the end of the 10 ms bounce event and captures a 1, the signal will not be observed as 0 until the next interrupt occurring 30 ms after the touch. Normally, this latency is inconsequential compared to how fast people can touch and release a switch.

Program 4.31 is extended, in Program 8.3, to include the task of observing the switch every 20 ms. *Last* is a private variable with permanent allocation containing the status of the switch 20 ms ago. *this* is a private variable with temporary allocation containing the status of the switch now. For both of these variables, 8 means not pressed, and 0 means pressed. A touch has occurred if the input was high on the last interrupt and low during this interrupt. Similarly, a release has occurred if the input was low on the last interrupt and high during this interrupt. The C implementation also shows how we could use this solution to solve the Example 8.1.

```

        org $3800
last    rmb 1      ;switch 20ms ago
Touch   rmb 1      ;set to 1 on touch
Release rmb 1      ;set to 1 on release
        org $4000
Switch_Init sei       ;make atomic
        bclr DDRT,#$08 ;PT3 is input
        movb #$80,TSCR1 ;enable TCNT
        movb #$02,TSCR2 ;lus
        bset TIOS,#$08 ;activate OC3
        bset TIE,#$08  ;arm OC3
        ldd TCNT ;time now
        addd #50 ;first in 50us
        std TC3
        clr Touch
        clr Release
        movb #$08,last
        cli      ;enable IRQ
        rts
OC3han movb #$08,TFLG1 ;ack      [4]
        ldd TC3          [3]
        addd #20000       [2]
        std TC3          ;next in 20 ms [2]
        ldaa PTT          [3]
        anda #$08 ;this  [1]
        psha             [2]
        eora #$08 ;not(this) [1]
        anda last ;not(this)&last[3]
        beq skip1         [1-3]
        movb #1,Touch ;signal [4-0]
skip1   ldaa last          [3]
        eora #$08 ;not(last) [1]
        anda 0,s ;not(last)&this[3]
        beq skip2         [1-3]
        movb #1,Release ;signal[4-0]
skip2   pula              [3]
        staa last ;last = this [3]
        rti               [8]
        org $FFE8
        fdb OC3han ;vector

```

```

unsigned char volatile Touch;
unsigned char volatile Release;
void Switch_Init(void){
    asm sei           // Make atomic
    DDRT &= ~0x08;   // PT3 is input
    TSCR1 = 0x80;
    TSCR2 = 0x02;   // 1 MHz TCNT
    TIOS |= 0x08;   // activate OC3
    TIE |= 0x08;   // arm OC3
    TC3 = TCNT+50; // first in 50us
    Touch = Release = 0;
    asm cli           // enable IRQ
}

void interrupt 11 OC3han (void){
    unsigned char this;
    unsigned char static last = 0x08;
    TFLG1 = 0x08;   // acknowledge C3F
    TC3 = TC3+20000; // next in 20 ms
    this = PTT&0x08; // read switch
    if((this^0x08)&last) Touch = 1;
    if((last^0x08)&this) Release = 1;
    last = this;
}

// Solutions to Example 8.1
void Switch_WaitPress(void){
    while(Touch==0){}; // wait for press
    Touch = 0; // set up for next time
}
void Switch_WaitRelease(void){
    while(Release==0){}; // wait
    Release = 0; // set up for next time
}

```

### Program 8.3

Switch debouncing using periodic polling software.

The advantage of periodic polling is it is easy to extend. For example, Program 8.3 can be adapted to interface any number of switches without the need for input capture or key wake-up capabilities on the port pins to which the switches are attached. If you are designing a computer keyboard or a consumer appliance, a latency of 30 ms can be tolerated. However, we will not classify this latency as real time. In safety-critical applications, such as air bag deployment, aircraft controls, medical devices, and military devices, we will need to interface switches with lower latency. Control systems also require low-latency input devices. We will learn in Chapter 13 that delays in a control system can cause the system to become unstable.

### Example 8.3 Solve Example 8.2 again, but with minimal latency.

**Solution** To minimize latency, we need to use an edge-triggered interrupt. We could use either key wake-up or input capture. The advantage of key wake-up is 9S12 microcontrollers have many lines that support key wake-up. However, input capture has the advantages of being a vectored interrupt and being able to trigger on both rising and falling edges. The solution in Program 8.4 uses input capture to know exactly when the switch is changed and output compare to delay past the bouncing interval. The algorithm is described in the flowchart in Figure 8.17. Just like Example 8.2, `Last` is a private variable with permanent allocation, containing the previous status of the switch (8 means released and 0 means touched). We need a reliable method to set `Last` at times we know the switch is not bouncing. Input capture is configured to generate an interrupt on either edge. When an edge-triggered interrupt has occurred, we know the switch is bouncing. So, 20 ms after an edge-triggered interrupt, we know the switch is not bouncing. In other words, we know the switch has begun to bounce at the input capture interrupt and is no longer bouncing at the output compare interrupt. Therefore, we update the variable `Last` by inputting from `PTT` during the OC5 interrupt. If `Last` is equal to 8 and we get an edge-triggered interrupt, we can reliably assume the switch has just been touched. Conversely, if `Last` is equal to 0 and we get an edge-triggered interrupt, we can reliably assume the switch has just been released.

```

org $3800
Last rmb 1 ;previous switch
Touch rmb 1 ;set to 1 on touch
Release rmb 1 ;set to 1 on release
    org $4000
Switch_Init sei ;make atomic
    bclr DDRT,#$08 ;PT3 is input
    movb #$80,TSCR1 ;enable TCNT
    movb #$02,TSCR2 ;lus
    bclr TIOS,#$08 ;activate IC3
    bset TIOS,#$20 ;activate OC5
    bset TIE,#$08 ;arm IC3
    bclr TIE,#$20 ;disarm OC5
    bset TCTL4,#$C0 ;both edges
    clr Touch
    clr Release
    ldaa PTT ;PT3 is input
    anda #$08 ;8 means release
    staa Last ;0 means touch
    cli ;enable IRQ
    rts

```

```

unsigned char volatile Touch;
unsigned char volatile Release;
unsigned char static Last; // previous
void Switch_Init(void){
    asm sei           // Make atomic
    DDRT &= ~0x08;   // PT3 is input
    TSCR1 = 0x80;    // enable
    TSCR2 = 0x02;    // lus clock
    TIOS |= 0x20;    // OC5
    TIOS &= ~0x08;   // IC3
    TIE |= 0x08;    // arm IC3
    TIE &= ~0x20;   // disarm OC5
    TFLG1 = 0x28;   // clear C5F,C3F
    TCTL4 |= 0xC0;   // both edges
    Last = PTT&0x08; // switch state
    Touch = Release = 0;
    asm cli          // enable IRQ
}
// occurs on a touch or release
void interrupt 11 IC3han(void){
    if(Last)

```

*continued on p. 400*

*continued from p. 399*

```

IC3han ldaa Last ;was 8->touch      [3]
    beq skip1 ;was 0->release [1-3]
    movb #1,Touch ;signal      [4-0]
    bra skip2      [3-0]
skip1 movb #1,Release ;signal      [0-4]
skip2 bclr TIE,#$08 ;disarm IC3 [4]
    bset TIE,#$20 ;arm OC5      [4]
    ldd TCNT      [3]
    addd #20000      [2]
    std TC3      ;OC5 in 20 ms [2]
    movb #$28,TFLG1 ;ack      [4]
    rti          [8]
OC5han ldaa PTT ;PT3 is input      [3]
    anda #$08 ;8 means release [1]
    staa Last ;0 means touch      [3]
    bset TIE,#$08 ;arm IC3      [4]
    bclr TIE,#$20 ;disarm OC5 [4]
    movb #$28,TFLG1 ;ack      [4]
    rti          [8]
    org $FFE4
    fdb OC5han ;vector
    org $FFE8
    fdb IC3han ;vector

```

```

        Touch = 1; // touch occurred
    else
        Release = 1; // release occurred
    TIE &= ~0x08; // disarm IC3
    TIE |= 0x20; // arm OC5
    TC5 = TCNT+20000; // 20 ms later
    TFLG1 = 0x28; // clear C5F,C3F
}
// occurs 20 ms after touch or release
void interrupt 13 OC5han(void){
    Last = PTT&0x08; // switch state
    TIE |= 0x08; // arm IC3,
    TIE &= ~0x20; // disarm OC5
    TFLG1 = 0x28; // clear C5F,C3F
}

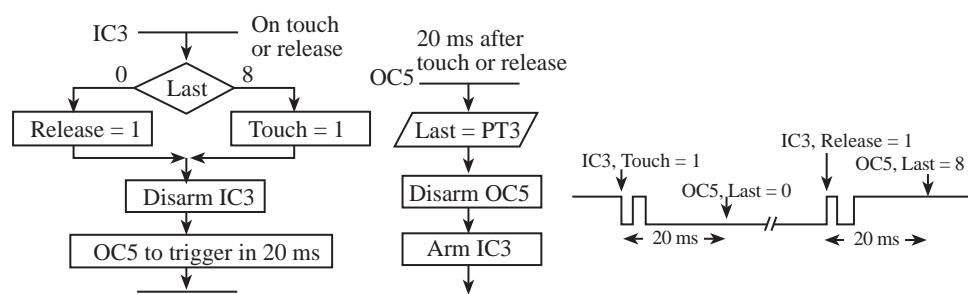
```

#### Program 8.4

Low-latency switch interface using input capture and output compare.

**Figure 8.17**

Input capture and output compare used to debounce a switch.



The ISRs in Programs 8.3 and 8.4 have no backward jumps. This means the time to execute the ISR will be short and bounded. The *software overhead* of an interface can be estimated by counting the cycles required to execute the software. In Program 8.3, we count from 48 to 50 cycles to execute the ISR. Adding in the nine cycles for the context switch, we estimate the overhead of Program 8.3 to be 57 to 59 cycles every 20 ms. Running at 4 MHz, this corresponds to 59/80000 or about 0.1%.

In Program 8.4, a touch causes two interrupts, one input capture (9 + 38 cycles) and one output compare (9 + 27 cycles), requiring a total of 83 cycles to execute. Because of the different branch pattern, a release requires only 82 cycles to execute. The C version runs a little faster, requiring 81 cycles for touch and 80 cycles for release. Notice the overhead for Program 8.4 is much less than for Program 8.3, because periodic polling requires running the ISR every 20 ms regardless of whether or not the switch has changed.

The latency in Program 8.4 between when the switch is touched and when the semaphore is signaled is only nine cycles for the context switch and 10 more cycles to run the ISR up until the point the semaphore is signaled. As always, the latency will be affected by

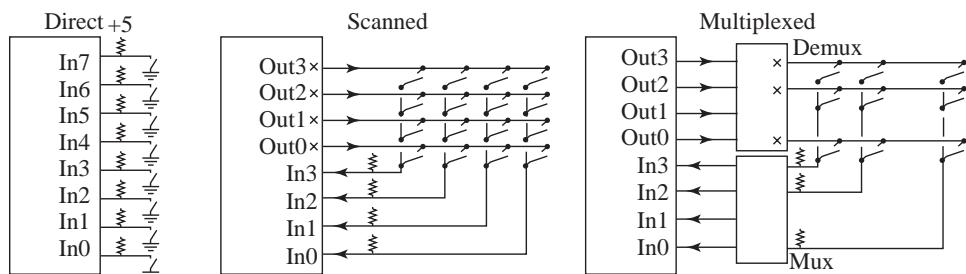
other software that runs with interrupts disabled. As we can see from Figure 8.17, this output compare and this input capture will not affect the latency of each other, because the OC5 occurs 20 ms after the IC3.

**Observation:** With a software-based debounced switch, the signal arrives at the computer input without delay, but software delays may or may not occur at either touch or release.

### 8.1.4 Basic Approaches to Interfacing Multiple Keys

In this section we attempt to interface as many switches as possible to a single 8-bit parallel port, and we will consider three interfacing schemes (Figure 8.18). In a *direct interface* we connect each switch to a separate microcomputer input pin. For example, using just one 8-bit parallel port, we can connect eight switches using the direct scheme. An advantage of this interfacing approach is that the software can recognize all 256 ( $2^8$ ) possible switch patterns. If the switches were remote from the microcomputer, we would need a nine-wire cable to connect it to the microcomputer. In general, if there are  $n$  switches, we would need  $n/8$  parallel ports and  $n + 1$  wires in our remote cable. This method will be used when there are a small number of switches or when we must recognize multiple and simultaneous key presses. Examples include music keyboards and the shift and control option keys. As illustrated in the Music (Example 8.4), implementing an interrupt-driven interface requires a lot of key wake-up ports. Therefore, if interrupt synchronization is required, it may be more appropriate to utilize periodic polling interrupt synchronization.

**Figure 8.18**  
Three approaches to interfacing multiple keys.



In a *scanned interface* the switches are placed in a row/column matrix (Figure 8.19). The  $\times$  at the four outputs signifies an open collector (an output with two states, HiZ and low). The computer drives one row at a time to zero while leaving the other rows at HiZ. By reading the column, the software can detect if a key is pressed in that row. The software

**Figure 8.19**  
Multiple keys are implemented by placing the switches in a matrix. (Notice there are fewer wires in the cable than there are keys.)



Courtesy of Jonathan Valvano.

“scans” the device by checking all rows one by one. The open-collector functionality will be implemented by toggling the direction register. Table 8.1 illustrates the sequence to scan the four rows.

**Table 8.1**

Scanning patterns for a 4 by 4 matrix keyboard.

Row	Out3	Out2	Out1	Out0
3	0	HiZ	HiZ	HiZ
2	HiZ	0	HiZ	HiZ
1	HiZ	HiZ	0	HiZ
0	HiZ	HiZ	HiZ	0

The direction register can be toggled to simulate the two output states, HiZ/0, of open-collector logic (see Exercise D1.36). This method can interface many switches with a small number of parallel I/O pins. In our example situation, the single 8-bit I/O port can handle 16 switches with only an eight-wire cable. The disadvantage of the scanned approach over the direct approach is that it can handle situations only where zero, one, or two switches are simultaneously pressed. This method is used for most of the switches in our standard computer keyboard. Recall that the shift, alt, and control keys are interfaced with the direct method. We can “arm” this interface for interrupts by driving all the rows to 0. The key wake-up or input capture mechanism can be used to generate interrupts on touch and release. Because of the switch bounce, an interrupt will occur when any of the keys changes.

In a *multiplexed interface*, the computer outputs the binary value defining the row number, and a hardware decoder will output the 0 on the selected row and HiZs on the other rows. The decoder must have open-collector outputs (illustrated again by the  $\times$  in the Figure 8.18 circuit). The computer outputs the sequence \$00, \$10, \$20, \$30 . . . , \$F0 to scan the 16 rows, as shown in Table 8.2.

**Table 8.2**

Scanning patterns for a multiplexed 16 by 16 matrix keyboard.

Row	Computer Output				Decoder Output			
	Out3	Out2	Out1	Out0	15	14	...	0
15	1	1	1	1	0	HiZ	...	HiZ
14	1	1	1	0	HiZ	0	...	HiZ
...	...	...	...	...	...	...	...	...
1	0	0	0	1	HiZ	HiZ	...	HiZ
0	0	0	0	0	HiZ	HiZ	...	0

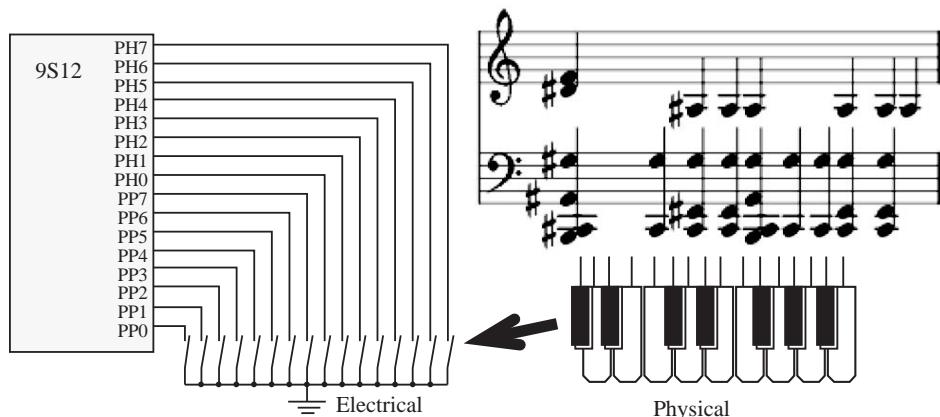
In a similar way, the column information is passed to a hardware encoder that calculates the column position of any 0 found in the selected row. One additional signal is necessary to signify the condition that no keys are pressed in that row. Since this interface has 16 rows and 16 columns, we can interface up to 256 keys! We could sacrifice one of the columns to detect the no key pressed in this row situation. In this way, we can interface 240 keys ( $15 \cdot 16$ ) on the single 8-bit parallel port. If more than one key is pressed in the same row, this method will detect only one of them. Therefore, we classify this scheme as being able to handle only 0 or one key pressed. Applications that can utilize this approach include touch screens and touch pads because they have a lot of switches but are interested only in the 0 or 1 touch situation. Implementing an interrupt-driven interface would require too much additional hardware. In this case, periodic polling interrupt synchronization would be appropriate.

**Example 8.4** Interface a 16-key keyboard as part of an electronic piano. The goal is to record the time and key pattern as the musician plays on the keyboard. The desired time resolution is 10 ms. The bounce time is less than 10 ms.

**Solution** In this direct interface, we connect each key up to a separate computer input pin so that we can distinguish all  $2^{16}$  possible key combinations. In order to establish time, we will need a periodic interrupt. Since we need a resolution of 10 ms, it makes sense to implement periodic polling at 10 ms. The variable Time holds the current time in 10 ms units. The bounce is less than 10 ms, so touching or releasing a key will cause exactly one entry into the recording buffer. To save cost, we will use the internal pull-up resistors by clearing bits **PPSH/PPSP** and setting bits in **PERH/PERP**. Most of the 9S12 ports can be used as simple inputs with pull-up, so it is not necessary to use key wake-up ports. The key pattern (this) will be compared to the value from the previous interrupt (Last), and the time and pattern will be recorded on each change. The variable Count contains the number of recorded key strokes. In the assembly version, Count is incremented by 2 in order to simplify access to the 16-time arrays. The assembly ISR requires from 40 to 58 cycles to execute (plus the nine cycles for the context switch), so the overhead of periodic polling will be low. The C ISR requires a maximum 86 cycles to execute, illustrating in some cases we can optimize for speed by writing in assembly.

**Figure 8.20**

Multiple keys are interfaced directly to input ports.



<pre>         org \$0800 ;DP512 Count  rmb 2 ;0,2,4,...998 Time   rmb 2 ;0.00 to 655.35 sec TimeBuf rmb 1000 ;in 0.01sec KeyBuf rmb 1000 ;pattern Last   rmb 2 ;value last ISR         org \$4000 Key_Init sei ;make atomic         clr DDRH ;PTH is input         clr PPSH ;pull-up         movb #\$FF,PERH ;enable         clr DDRP ;PTP is input         clr PPSP ;pull-up         movb #\$FF,PERP ;enable         movb #\$80,TSCR1 ;enable TCNT         movb #\$02,TSCR2 ;lus         bset TIOS,#\$08 ;activate OC3         bset TIE,#\$08 ;arm OC3         ldd TCNT ;time now         addd #50 ;first in 50us         std TC3         movw #0,Count         movw #0,Last     </pre>	<pre> unsigned short volatile Count; unsigned short volatile Time; unsigned short TimeBuf[500]; unsigned short KeyBuf[500]; unsigned short static Last; void Key_Init(void){     asm sei           // Make atomic     DDRH = 0x00;      // PTH is input     PPSH = 0x00;      // pull up     PERH = 0xFF;      // enable     DDRP = 0x00;      // PTP is input     PPSP = 0x00;      // pull up     PERP = 0xFF;      // enable     TSCR1 = 0x80;     TSCR2 = 0x02;      // 1 MHz TCNT     TIOS  = 0x08;      // activate OC3     TIE  = 0x08;      // arm OC3     TC3 = TCNT+50; // first in 50us     Count = 0;         // empty buffer     Last = 0;     Time = 0;     asm cli           // enable IRQ } </pre>
--	--

*continued on p. 404*

*continued from p. 403*

```

    movw #0,Time
    cli           ;enable IRQ
    rts
OC3han ldy Time          [3]
    iny          [1]
    sty Time   ;0.01 sec [3]
    ldaa PTH      [3]
    ldab PTP   ;this [3]
    cpd Last      [3]
    beq skip   ;skip if same[1-3]
    std Last   ;for next [3-0]
    ldx Count      [3-0]
    cpx #1000     [2-0]
    bhs skip   ;skip if full[1-0]
    std KeyBuf,x ;save key [3-0]
    sty TimeBuf,x ;save time[3-0]
    leax 2,x      [2-0]
    stx Count      [3-0]
skip  movb #$08,TFLG1 ;ack [4]
    ldd TC3       [3]
    addd #10000    [2]
    std TC3   ;next in 10 ms [3]
    rti          [8]
    org $FFE8
    fdb OC3han ;vector

```

```

// interrupt every 10 ms
void interrupt 11 OC3han (void){
unsigned short this;
    Time++; // the time now
    this = (PTH<<8)+PTP;
    if(this != Last){
        Last = this;
        if(Count<500){
            TimeBuf[Count] = Time;
            KeyBuf[Count] = this;
            Count++; // record time,key
        }
    }
    TFLG1 = 0x08; // acknowledge C3F
    TC3 = TC3+10000; // next in 10 ms
}

```

### Program 8.5

Keyboard logging using periodic polling software.

**Checkpoint 8.4:** How many 9S12 pins will it take to interface an 88-key piano?

In order to save I/O lines, the scanned approach can be used. In general, if we have  $n + m$  I/O ports, we can make  $n$  rows (open collector outputs from the computer) and  $m$  columns (inputs to the computer with pull-up resistors) and interface a keyboard with  $n \cdot m$  keys. The scanned keyboard operates properly if

1. No key is pressed.
2. Exactly one key is pressed.
3. Exactly two keys are pressed.

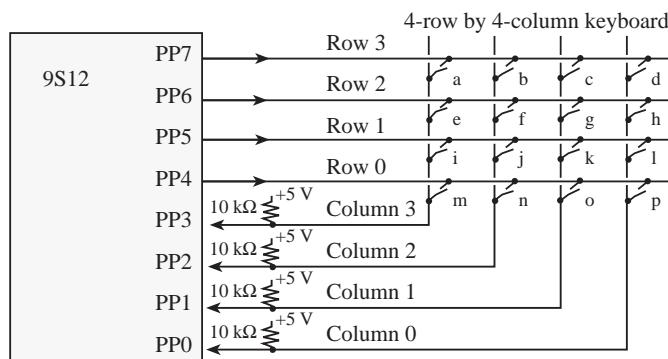
With a scanned approach, we give up the ability to detect three or more keys pressed simultaneously. If three keys are pressed in a “L” shape, then the fourth key that completes the rectangle will appear to be pressed. Therefore, special keys like the shift, control, option, and alt are not placed in the scanned matrix but rather are interfaced directly—each to a separate input port, like the piano keyboard example.

**Example 8.5** Interface a 16-key matrix keyboard. There will be either one key touched or no keys touched. The bounce time is less than 10 ms. Keys are input in the background and should be passed to the foreground using an FIFO queue.

**Solution** This matrix keyboard divides the sixteen keys into four rows and four columns, as shown in Figure 8.21. Each key exists at a unique row/column location. It will take eight I/O pins to interface the rows and columns. Any output port on the 9S12 could have been used to interface the rows. To scan the matrix, the software will drive the rows one at a time

**Figure 8.21**

A matrix keyboard interfaced to the microcomputer.



with open collector logic, then read the columns. The open collector logic, with outputs HiZ and 0, will be created by toggling the direction register on the four rows. Actual  $10\text{ k}\Omega$  pull-up resistors will be placed on the column inputs (PP3-PP0) rather than configured internally, because the internal pull-ups are not fast enough to handle the scanning procedure. The computer will respond to a touch using a falling-edge key wake-up mechanism on the four columns. We also could have used Port T input capture to interface the columns.

Program 8.6 shows the initialization software. The data structure will assist in the scanning algorithm, and it provides a visual mapping from the physical layout of the keys to the ASCII code produced when touching that key. The structure also makes it easy to adapt this solution to other keyboard interfaces. The output compare interrupt will be used to debounce the switches. Output compare is turned on, but it is initially disarmed.

<pre> ScanTab  fcb  \$80,"abcd" ;top row         fcb  \$40,"efgh"         fcb  \$20,"ijkl"         fcb  \$10,"mnop" ;bottom row         fcb  0  Arm         movb #\$F0,DDRP ;PP7-PP4 output low         movb #\$0F,PIEP ;arm         movb #\$0F,PIFP ;clear flags         bclr TIE,#\$08 ;disarm OC3         rts  Key_Init         clr  PPSP      ;falling edge         jsr  Fifo_Init         clr  PTP       ;PP7-PP4 oc output         bsr  Arm       ;PP3-PP0 inputs         movb #\$80,TSCR1 ;enable TCNT         movb #\$02,TSCR2 ;lus         bset TIOS,#\$08 ;activate OC3         cli         rts </pre>	<pre> const struct Row {     unsigned char direction;     unsigned char keycode[4]; } typedef const struct Row RowType; RowType ScanTab[5] = {     { 0x80, "abcd" }, // row 3     { 0x40, "efgh" }, // row 2     { 0x20, "ijkl" }, // row 1     { 0x10, "mnop" }, // row 0     { 0x00, "      " }};  void Arm(void){     DDRP = 0xF0; // PP7-PP4 output low     PIEP = 0x0F; // arm key wakeup PP3-PP0     PIFP = 0x0F; // clear flags     TIE &amp;= ~0x08; // disarm OC3 }  void Key_Init(void){     PPSP = 0; // falling edge PP3-PP0     Fifo_Init();     PTP = 0; // PP7-PP4 oc output     Arm(); // PP3-PP0 inputs     TSCR1 = 0x80;     TSCR2 = 0x02; // 1 MHz TCNT     TIOS  = 0x08; // activate OC3     asm cli } </pre>
--	---

### Program 8.6

Initialization software for a matrix keyboard.

Program 8.7 shows the scanning software. The scanning sequence is listed in Table 8.3. There are two steps to scan a particular row.

1. Select that row by driving it low, while the other rows are HiZ.
2. Read the columns to see if any keys are pressed in that row.

0 means the key is pressed

1 means the key is not pressed

**Table 8.3**

Patterns for a 4 by 4 matrix keyboard.

DDRP	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0
\$80	0	HiZ	HiZ	HiZ	a	b	c	d
\$40	HiZ	0	HiZ	HiZ	e	f	g	h
\$20	HiZ	HiZ	0	HiZ	i	j	k	l
\$10	HiZ	HiZ	HiZ	0	m	n	o	p

```
; Returns RegA ASCII key pressed,
;      RegY number of keys pressed
;      Y=0 if no key pressed
Key_Scan ldy #0      ;Number pressed
          ldx #ScanTab
loop    ldab 0,x      ;row select
          beq done
          stab DDRP  ;select row
          nop      ;time to settle
          brset PTP,#$01,notPP0
          ldaa 4,x  ;code for column 0
          iny
notPP0  brset PTP,#$02,notPP1
          ldaa 3,x  ;code for column 1
          iny
notPP1  brset PTP,#$04,notPP2
          ldaa 2,x  ;code for column 2
          iny
notPP2  brset PTP,#$08,notPP3
          ldaa 1,x  ;code for column 3
          iny
notPP3  leax 5,x  ;Size of entry
          bra loop
done    rts
```

```
/* Returns ASCII code for key pressed,
   Num is the number of keys pressed
   both equal zero if no key pressed */
unsigned char Key_Scan(short *Num){
RowType *pt; unsigned char column,key;
short j;
(*Num) = 0;
key = 0;      // default values
pt = &ScanTab[0];
while(pt->direction){
  DDRP = pt->direction; // one output
asm nop
  column = PTP; // read columns
  for(j=3; j>=0; j--){
    if((column&0x01)==0){
      key = pt->keycode[j];
      (*Num)++;
    }
    column>>=1; // shift into position
  }
  pt++;
}
return key;
}
```

### Program 8.7

Scanning software for a matrix keyboard.

It is important to observe column and row signals on a dual trace oscilloscope while running the software at full speed, because it takes time for the correct signal to appear on the column after the row is changed. In some cases, a software delay should be inserted between setting the row and reading the column (shown as the `nop` instruction in Program 8.7). The length of the delay you will need depends on the size of the pull-up resistor and any stray capacitance that may exist in your circuit.

Program 8.8 will use a combination of key wake-up and output compare interrupts to perform input in the background. When arming for interrupts (`Arm` function), we set all four rows to output 0. In this way, a falling-edge key wake-up interrupt will occur on any key touched. When a key wake-up interrupt occurs, we will disarm key wakeup and arm an output compare to trigger in 10 ms. It is during the output compare ISR we scan the matrix. If there is exactly one key, we enter it into the FIFO. An interrupt may occur on release due to bounce. However, 10 ms after the release, when we scan during the OC handler, the `Key_Scan` function will return a num of 0, and we will ignore it.

```

;occurs on touch
KeyWakeP clr PIEP ;disarm
    bset TIE,#$08    ;arm OC3
    ldd TCNT
    addd #10000
    std TC3      ;OC3 in 10 ms
    movb #$08,TFLG1 ;ack
    movb #$0F,PIFP  ;clear flags
    rti
;occurs 10ms after touch
OC3han bsr Key_Scan
    cpy #1
    bne skip  ;ignore 0 or 2,3...
    jsr Fifo_Put
skip   jsr Arm   ;setup for next key
    rti
    org $FF8E
    fdb KeyWakeP ;vector
    org $FFE8
    fdb OC3han ;vector

// occurs on a touch
void interrupt 56 KeyWakeP(void){
    PIEP = 0x00; // disarm key wakeup
    TIE |= 0x08; // arm OC3
    TC3 = TCNT+10000; // 10 ms later
    TFLG1 = 0x08; // clear C3F
    PIFP = 0x0F; // ack
}
// occurs 10ms after touch
void interrupt 11 OC3han (void){
    unsigned char this;
    short num;
    this = Key_Scan(&num);
    if(num == 1){
        Fifo_Put(this);
    }
    Arm(); // enable for next key
}

```

**Program 8.8**

Keyboard input using key wake-up and output compare.

**Checkpoint 8.5:** Using a method similar to Example 8.5, how many 9S12 pins will it take to interface a 100-key keyboard ?

One of the limitations of Program 8.8 is *two-key rollover*. When people type very fast, they sometimes type the next key before they release the first key. For example, when the operator types the letters “hlp” slowly with one finger, the keyboard status goes in this sequence:

<none>, <h>, <none>, <l>, <none>, <p>, <none>

Conversely, if the operator types quickly, there can be two-key rollover, which creates this sequence:

<none>, <h>, <hl>, <l>, <lp>, <p>, <none>

where <hl> means both keys ‘h’ and ‘l’ are touched. Two-key rollover means the keyboard does not go through a state where no keys are touched between typing the ‘h’ and the ‘l’. Because the ‘h’ ‘l’ ‘p’ are in the same column of Figure 8.21, there will not be a falling edge on PP0 when the ‘l’ or ‘p’ keys are typed, and consequently, these keys will not be recorded. If we want to handle two-key rollover, we should use periodic polling and scan the keyboard every 20 ms.

In order to interface a keyboard with many keys, we need the multiplexed approach, using a decoder and encoder, as shown on the right side of Figure 8.18. If you look back at the row pattern in Table 8.3, you see four possible output patterns. In Example 8.5, four output pins are used to generate the patterns, but we could be more efficient and generate four output patterns with just two output pins. With a hardware decoder (or demultiplexer) we could generate the  $2^n$  patterns with **n** digital output pins of the microcontroller. Similarly, the column information is processed by an encoder, such that  $2^m$  columns are converted to **m** signals. These **m** signals are interfaced to input pins on the microcontroller. We will need a way to detect the presence or absence of any keys in that row. We could add additional status signal, so with **n** + **m** + 1 I/O pins, we can interface a keyboard with  $(2^m) \cdot (2^n)$  keys. A better way to tell whether or not any keys are pressed is eliminate one of the columns, using only  $(2m - 1)$  columns. The code for the last column could signify no key. In this scheme, with **n** + **m** I/O pins we can interface a keyboard with  $(2^m - 1) \cdot (2^n)$  keys. The keyboard is scanned in the usual manner; it is just

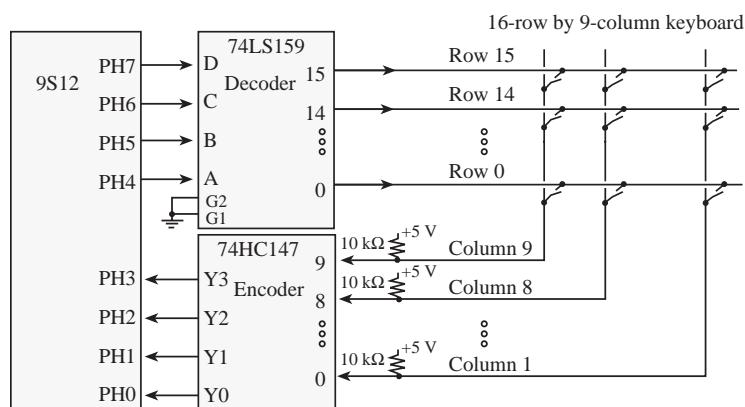
that the hardware reduces the number of I/O ports needed. If two keys are pressed in the same row, the multiplexer will tell us about only one, so we lose the ability to detect two keys pressed simultaneously. That is, the multiplexed keyboard operates properly if

1. No key is pressed.
2. Exactly one key is pressed.

**Example 8.6** Interface a 144-key keyboard matrix keyboard. There will be either one key touched or no keys touched. The bounce time is less than 10 ms. Keys are input in the background and should be passed to the foreground using an FIFO queue.

**Solution** In order to interface 144 switches, the decoder/encoder interface is used. In this approach, the keys are again divided into rows and columns. This keyboard has 16 rows and 9 columns, giving 144 keys, as shown in Figure 8.22. The computer specifies the row number (0 to 15) by outputting to the four most significant bits to Port H. There are no special features of Port H used, so any 8-bit I/O port could have been used. Since the bounce time is 10 ms, we will execute periodic polling every 20 ms. Table 8.2 shows the row patterns created during scanning. The 4 to 16 decoder, 74LS159, has open collector outputs and will drive exactly one row to zero, and the other rows will be HiZ. The 74147 is a 10 to 4 line priority encoder. If all nine inputs of the 74147 are high (i.e., no key pressed in this row), then the least significant 4-bit signals will be 1111. If exactly one key is pressed in the row, then exactly one “0” will exist on the 74147’s inputs, and the 74147 output will be the negative logic location of the column number. For example, if a key in column 4 is pressed and that row is selected, then the 74147 “4” input will be 0 and Port H bits 3-0 will be 1011. If two or more keys are pressed in the same row, then bits 3-0 will be the highest column number pressed (again in negative logic). For example, if the keys in column 3 and 7 are both pressed, then the 74147, “3” and “7” inputs will both be 0, but Port H bits 3-0 will signify only the “7” 1000. As mentioned earlier, a decoded/encoded interface cannot handle more than one key pressed simultaneously.

**Figure 8.22**  
A decoded/encoded interface of a keyboard with 144 keys.



The software solution is presented as Program 8.9. To scan the keyboard, we output the row number on PH7-PH4, then look at the column number on PH3-0. If PH3-0 is 1111, no key is pressed in that row. Similar to Example 8.5, we may need to add a short delay after setting the row and before reading to column to allow the hardware to settle (shown as the `nop`). The length of this delay can be determined using a two-channel oscilloscope on the pairs of Port H pins while the software is running at full speed. Periodic polling interrupt synchronization is appropriate for this interface, because a lot of hardware would be necessary to trigger an interrupt directly on a key touch. The background thread will pass data to the foreground using an FIFO queue. If the switch happens to be bouncing at the time of the interrupt, the 74HC147 outputs may contain incorrect patterns (e.g., representing

<pre> OneAgo rmb 1 ;key 1 interrupt ago TwoAgo rmb 1 ;key 2 interrupts ago Key_Init sei     ldaa #\$F0 ;PH7-PH4 outputs     staa DDRH ;PH3-PH0 inputs     bset TIE,#\$08 ;arm OC3     bset TIOS,#\$08 ;activate OC3     movb #\$80,TSCR1 ;enable TCNT     movb #\$02,TSCR2 ;lus     ldd TCNT     addd #50     std TC3 ; first in 50us     movb #\$08,TFLG1 ;clr C3F     clr OneAgo     clr TwoAgo     jsr Fifo_Init     cli     rts  ; returns RegA=code (0 for none) KeyScan clrA ;=0 means no     clr PTH ;row=0 loop  nop     ldab PTH ;read columns     andb #\$0F     cmpb #\$0F ;\$0F means no     beq none     ldab PTH ;bits 7-4 = rows     eorb #\$0F ;bits 3-0 = 1-9     tba ;found one none   ldab PTH     addb #\$10 ;next row     stab PTH     bcc loop     rts OC3han bsr KeyScan     ldab OneAgo     cmpb TwoAgo     beq skip     cmpa OneAgo ;same as last?     bne skip     psha     jsr Fifo_Put ;new key typed     pula skip   stab TwoAgo     staa OneAgo     ldd TC5     addd #20000     std TC5 ;every 20ms     movb #\$08,TFLG1 ;ack C3F     rti     org \$FFE8     fdb OC3han ;vector </pre>	<pre> void Key_Init(void){     asm sei           // PH7-PH4 outputs     DDRH = 0xF0;     // PH3-PH0 inputs     TIE  = 0x08;    // Arm OC3     TIOS  = 0x08;   // enable OC3     TSCR1 = 0x80;     TSCR2 = 0x02;   // 1 MHz TCNT     TC3 = TCNT+50; // first in 50us     TFLG1 = 0x08;   // clear C3F     Fifo_Init();     asm cli }  // returns key code // if a key is pushed // bits 7-4 are row, 0 to 15 // bits 3-0 are column, 1 to 9 // returns 0 if no key pressed unsigned char KeyScan(void){     unsigned char key,row;     key = 0; // means no key pressed     for(row=0; row&lt;16; row++){         PTH = row&lt;&lt;4; // Select row         asm nop         if((PTH&amp;0x0F) != 0x0F){             key= PTH^0x0F;         }     }     return(key); }  // interrupt every 20 ms void interrupt 11 OC3han (void){     unsigned char static OneAgo; // 20ms ago     unsigned char static TwoAgo; // 40ms ago     unsigned char new;     new = KeyScan(); // Current pattern     if((new == OneAgo)&amp;&amp;(OneAgo != TwoAgo)){         Fifo_Put(new);     }     TwoAgo = OneAgo;     OneAgo = new;     TFLG1 = 0x08; // acknowledge C3F     TC3 = TC3+20000; // next in 20 ms } </pre>
--	---

### Program 8.9

Software for a decoded/encoded keyboard with 144 keys.

patterns that are neither the old nor the new status). Therefore, we will put a key into the FIFO after we see it twice and if it is different from the key we saw two interrupts ago. In this example, there is a 40 to 60 ms time delay (latency) between the key press and when the key is put into the FIFO. The static variables `OneAgo` and `TwoAgo` will be initialized to 0 by the compiler.

**Checkpoint 8.6:** How should the Ctrl, Alt, and Shift keys be interfaced within a standard PC keyboard? How should the remaining 100 keys be interfaced?

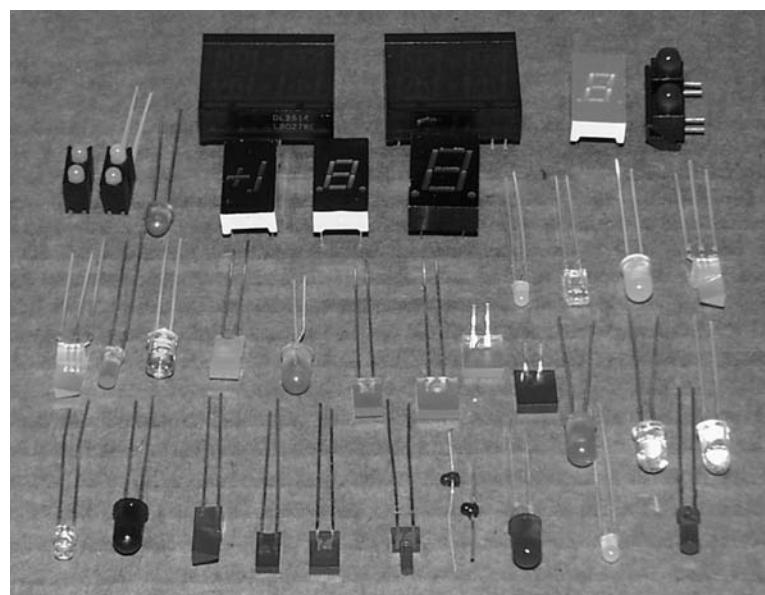
**Checkpoint 8.7:** How would you interface a touchpad that has the equivalent of over one million keys?

## 8.2 Output LEDs

Similar to the keyboard interfaces in the previous section, we will develop LED interfaces in the direct, scanned, and multiplexed categories (Figure 8.23 and 8.24). A direct interface has a unique computer output pin for each LED. In this way, a single output port can control eight LED segments. Once the output port is set, no software action is required to maintain the direct interface. All possible “on/off” patterns can be generated by the direct interface. The scanned interface organizes the LEDs in a matrix with rows and columns, and each row and each column has a unique output pin. In Figure 8.24 current sources are used to drive the rows and current sinks are connected to the columns. If the LEDs are configured in a 4 by 4 matrix, a single output port can control 16 LED segments. Software executed on a continuous and regular basis will be required to maintain the display. All possible on/off patterns can be generated by the scanned interface. The multiplexed display is also organized in a matrix with rows and columns but has decoding hardware so that there are more rows and columns than there are computer output pins. Theoretically, as shown in Figure 8.24, it is possible for a single output port to control  $16 \cdot 16$ , or 256, LED segments. Again, software executed on a continuous and regular basis will be required to maintain the display. Depending on the configuration of the decoder, not all possible on/off patterns can be generated by the multiplexed interface. Because of the maximum allowable LED current limitations (the details to be presented later in the section), it will not be practical to have a 16 by 16 LED matrix. Nevertheless, the multiplexed approach is common, especially for 7-segment and 15-segment LED displays. We will

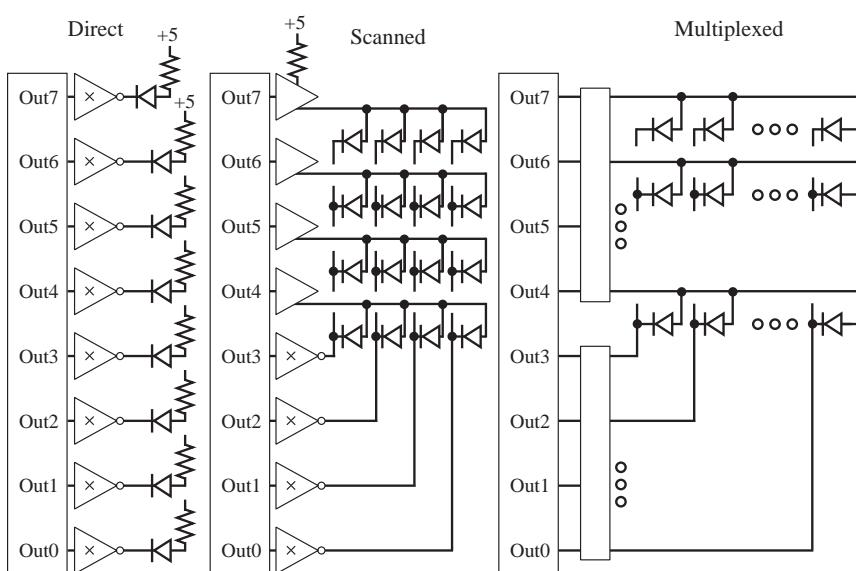
**Figure 8.23**

LEDs come in a wide variety of shapes, sizes, colors, and configurations.



Courtesy of Jonathan Valvano.

**Figure 8.24**  
Three approaches to interfacing multiple LEDs.

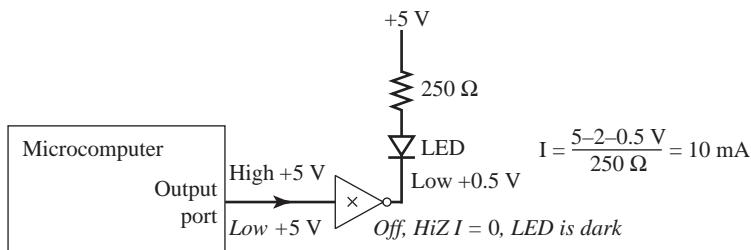


also develop an LED interface that uses a multiplexed approach but performs the scanning operations in hardware so that periodic software maintenance is not required.

### 8.2.1 Single LED Interface

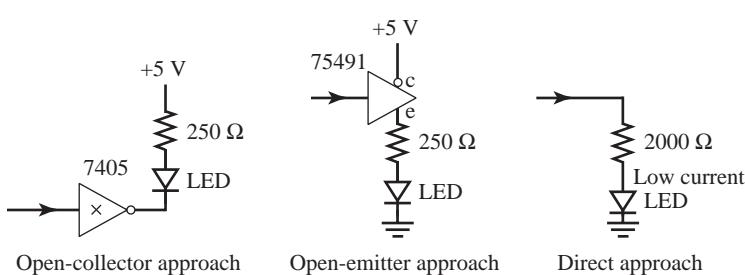
If you connect a microcomputer output pin to the LED interface, then the microcomputer can control the state (on/off) of the LED. If you make the microcomputer output high (a logic 1), the output of the 7405 will be low, current will flow through the LED, and it will be lit. The LED voltage will be about 2 V when it is lit. If you make the computer output low (a logic 0), the output of the 7405 will be off (HiZ, not driven, high impedance, disconnected), no current will flow through the LED, and it will be dark. The use of the 7405 provides the necessary current (about 10 mA) to activate the light. The  $250\ \Omega$  resistor is selected to control the brightness of the light (Figure 8.25).

**Figure 8.25**  
A single LED interface.



We can use either open-collector logic or open-emitter logic as current switches. The current on some LEDs is so low (e.g., 1 mA) that they can be connected directly to an output port, as shown in Figure 8.26. When the open-collector output is 0, it will sink current to

**Figure 8.26**  
Three approaches to controlling the current to an LED.



ground (turning on the LED, relay, stepper coil, solenoid, etc.). When the open-collector output is floating, it will not sink any current (turning off the LED, relay, stepper coil, solenoid, etc.). Similarly, when the 75491 input is high, the open-emitter output transistor is active, sourcing current into the LED. When the 75491 input is low, the open-emitter output transistor is off, making the LED current zero. Table 8.4 provides the output low currents for some typical open-collector devices. Table 8.5 provides the output source currents for some typical open-emitter devices.

Darlington switches like the ULN-2061 through ULN-2077 and MOSFETs like the IRF-540 can be used either in open-collector mode to sink current or in open-emitter mode to source current. For all the devices the actual output voltage depends on the output current. What is shown in Tables 8.4 and 8.5 is the output voltage at maximum output current. For the transistor devices, the output voltages/currents also depend on the input currents.

**Table 8.4**

Output parameters for various open-collector gates.

Family	Example	$V_{OL}^*$	$I_{OL}$
Standard TTL	7405	0.4 V	16 mA
Schottky TTL	74S05	0.5 V	20 mA
Low-power Schottky TTL	74LS05	0.5 V	8 mA
High-speed CMOS	74HC05	0.33 V	4 mA
High-voltage output TTL	7406	0.7 V	40 mA
High-voltage output TTL	7407	0.7 V	40 mA
Silicon monolithic IC	75492	0.9 V	250 mA
Silicon monolithic IC	75451-75454	0.5 V	300 mA
Darlington switch	ULN-2074	1.4 V	1.25 A
MOS field-effect transistor (MOSFET)	IRF-540	Varies	28 A

\* Voltage at maximum  $I_{OL}$ .

**Table 8.5**

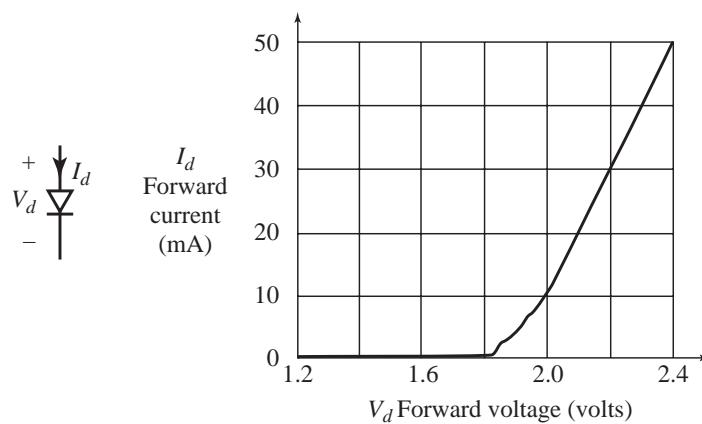
Output parameters for various open-emitter gates.

Family	Example	$V_{CE}$	$I_{CE}$
Silicon monolithic IC	75491	0.9 V	50 mA
Darlington switch	ULN-2074	1.4 V	1.25 A
MOSFET	IRF-540	Varies	28 A

For scanned and multiplexed LED interfaces, we will use both current sources and sinks. In the following interface circuits we assume the desired LED setpoint is 2 V and 10 mA. Since the LED is a diode, the voltage and current relationship is quite nonlinear. The voltage/current relationship for the LTP-1057A and LTP-1157A dot matrix LED displays is plotted in Figure 8.27.

**Figure 8.27**

Typical voltage/current response of a LED.



To prevent the LED from overheating, we must limit the electric power by using a current-limiting series resistor, as shown in Figure 8.28. Table 8.6 illustrates typical LED specifications.

**Table 8.6**

Absolute maximum rating for LTP-1057A and LTP-1157A 5 by 7 dot matrix displays.

Parameter	Red	Green	Yellow	Orange	Units
Maximum power	55	75	60	75	mW
Peak forward current	160	100	80	100	mA
Max continuous current	25	25	20	25	mA

The LED power can be calculated from its voltage and current.

$$P_d = V_d \cdot I_d$$

The resistor value is chosen to establish the desired voltage/current ( $V_d/I_d$ ) operating point for the LED. For the 75492 open-collector circuit, the resistor is calculated as (Figure 8.28)

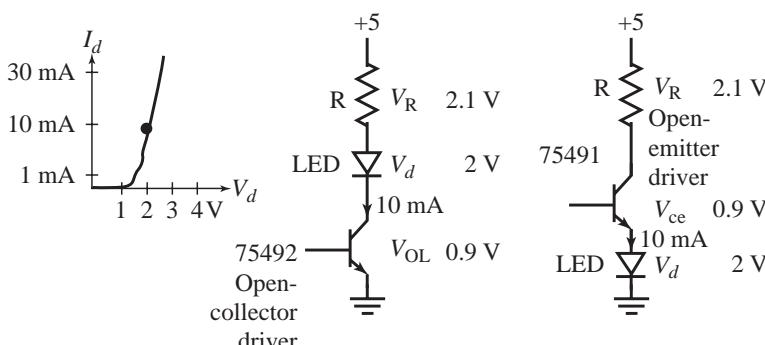
$$R = (5 - V_d - V_{OL})/I_d = (5 - 2 - 0.9)/10 \text{ mA} = 210 \Omega$$

Similarly for the 75491 open-emitter circuit, the resistor is calculated as (Figure 8.28)

$$R = (5 - V_d - V_{ce})/I_d = (5 - 2 - 0.9)/10 \text{ mA} = 210 \Omega$$

**Figure 8.28**

Calculating the resistor used in the LED interface.



**Checkpoint 8.8:** Using the open-collector approach, what resistor value would you need for a 2.5 V, 20 mA LED?

**Checkpoint 8.9:** What resistor value do you need in the direct approach LED interface of Figure 8.26 if the LED voltage is 2 V and current is 1 mA, assuming  $V_{OH}$  is 4.5 V?

## 8.2.2 Seven-Segment LED Interfaces

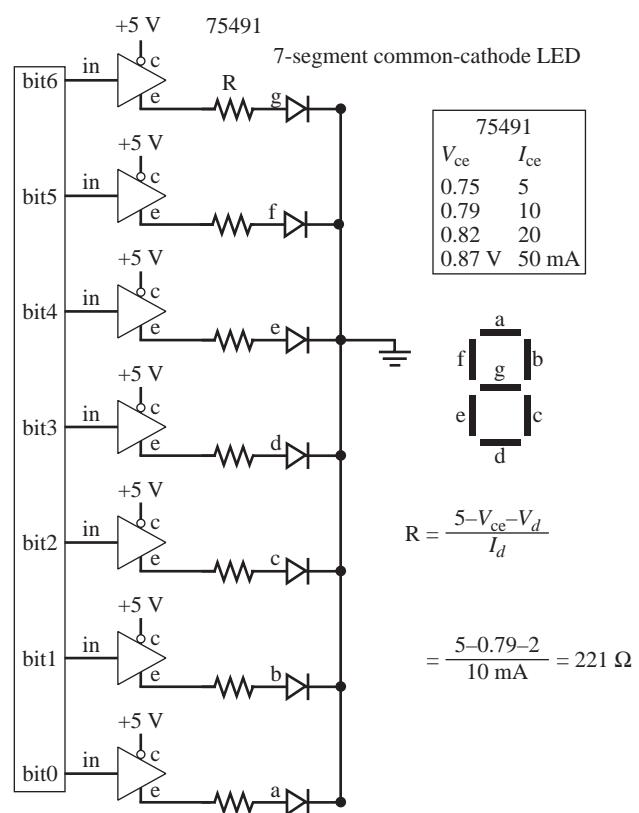
When creating LEDs that display numbers, it is appropriate to use common-cathode or common-anode seven-segment LED modules. Some modules have an eighth or ninth segment to display a right-side and/or left-side decimal point. The circuit in Figure 8.29 shows a direct interface to a single seven-segment common-cathode LED display. It is called common cathode because the seven cathodes are connected. We label it direct because there is a separate computer output pin for each LED segment. The software simply writes to the output port to control the seven segments. A current sink is required to interface a common-anode display. It is called common anode because the seven anodes are connected (Figure 8.30).

**Common error:** If you try to replace the seven individual resistors in either of the two circuits in Figures 8.29 and 8.30 with a single resistor on the “common” side, then the brightness of each LED segment will be a function of how many LEDs are on. For example, if only one segment is on, it will be very bright; if all seven are on, then they will be dim.

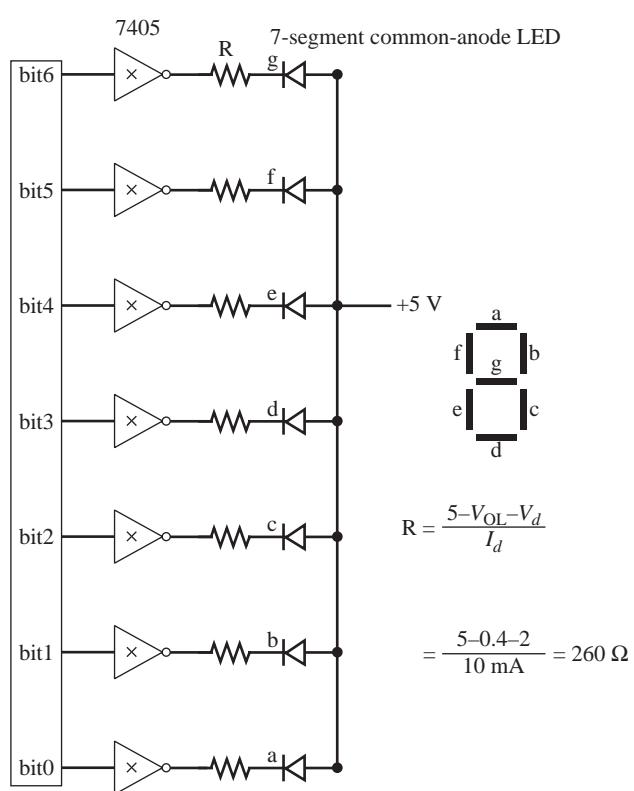
**Observation:** The current-limiting resistors should be placed on the individual LED connections and not on the “common” connection.

**Figure 8.29**

Seven-segment common-cathode LED interface.

**Figure 8.30**

Seven-segment common-anode LED interface.



**Example 8.7** Design a three-decimal digital LED display capable of displaying the numbers from 000 to 999.

**Solution** The 21 LED segments will be interfaced in a 3-column by 7-row rectangular matrix. There will be a column for each digit and a row for each of the segments a, b, c, d, e, f, g. The software will utilize a simple 3-byte variable to contain the current value to be displayed. For example, if the value “456” is to be displayed, the main program will set the 24-bit global to \$666D7C, as explained in Table 8.7. Our interrupt software will read this global and output the appropriate signals to the output ports (Figure 8.31).

**Figure 8.31**  
Seven-segment common-anode LED's.

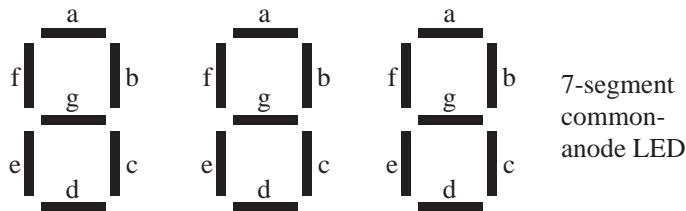


Table 8.7 gives the conversion between decimal digit and seven-segment binary code assuming segments g, f, e, d, c, b, a are mapped into bits 6, 5, 4, 3, 2, 1, 0. This interface can also be used to create hexadecimal displays.

**Table 8.7**  
Patterns for a seven-segment display.

Digit	Segments	Binary code
0	f, e, d, c, b, a	%00111111=\$3F
1	b, c	%00000110=\$06
2	g, e, d, b, a	%01011011=\$5B
3	g, d, c, b, a	%01001111=\$4F
4	g, f, c, b	%01100110=\$66
5	g, f, d, c, a	%01101101=\$6D
6	g, f, e, d, c	%01111100=\$7C
7	c, b, a	%00000111=\$07
8	g, f, e, d, c, b, a	%01111111=\$7F
9	g, f, c, b, a	%01100111=\$67
A	g, f, e, c, b, a	%01101111=\$6F
b	g, f, e, d, c	%01111100=\$7C
C	f, e, d, a	%00111001=\$39
d	g, e, d, c, b	%01011110=\$5E
E	g, f, e, d, a	%01111001=\$79
F	g, f, e, a	%01110001=\$71

It is easy to interface PNP transistors to the microcontroller in a current sourcing mode, because a digital low will create a large emitter-based voltage, turning it on. To turn the PNP off, we make the output high, producing small or no emitter-based voltage. Each 2N2907 PNP transistor will source current for one column (0, 30, 60,..., 210 mA, depending on how many LEDs will be active in that column). Resistor R1 is chosen small enough to drive the 2N2907 into saturation when the digital output is low. The 2N2907 ICE needs up to  $7 \times 30 = 210$  mA, and the current gain is 100, so the  $I_B$  should be 2.1 mA or higher. When the port output is low, the voltage ( $V_{OL}$ ) is about 0.5 V. Saturation occurs when the  $V_{EB}$  voltage is above 0.8 V. If the 2N2907 is in saturation the  $V_B$  is 4.2 V ( $5 - V_{EB}$ ), so

$R_1$  should be less than  $(4.2 - 0.5 \text{ V})/2.1 \text{ mA} = 1.76 \text{ k}\Omega$ . The  $R_1$  resistors and PNP transistors could be replaced with an open-emitter driver like the ULN-2074. I suggest using  $R_1 = 470 \Omega$ , to make sure the PNP is completely in saturation.

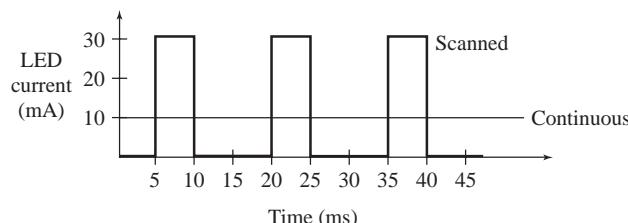
It is easy to interface NPN transistors to the microcontroller in a current-sinking mode, because a digital high will create a large base-emitter voltage, turning it on. To turn the NPN off, we make the output low, producing small or no base-emitter voltage. Each 2N2222 NPN transistor will sink current for one row (0 or 30 mA depending on whether or not that LED will be active). Resistor  $R_2$  is chosen small enough to drive the 2N2222 into saturation when the digital output is high. The 2N2222  $I_{CE}$  needs to be 30 mA, and the current gain is 100, so the  $I_B$  should be 0.3 mA or higher. Saturation occurs when the  $V_{BE}$  voltage is above 0.8 V. If the port output is high, the voltage ( $V_{OH}$ ) is about 4.5 V. If the 2N2222 is in saturation, the  $V_B$  is 0.8 V ( $V_{BE} = 0.8 \text{ V}$ ), so  $R_2$  should be less than  $(4.5 - 0.8 \text{ V})/0.3 \text{ mA} = 12 \text{ k}\Omega$ . A rule of thumb with transistors in saturated mode is to make the base resistors (e.g.,  $R_1$  and  $R_2$ ) two or five times smaller than the calculation suggests, making sure the transistors do saturate. For this interface, I suggest  $R_2 = 2.7 \text{ k}\Omega$ . The  $R_2$  resistors and NPN transistors could be replaced with any of open-collector drivers from Table 8.4 with  $I_{OL}$  greater than 30 mA.

Resistor  $R_3$  controls the LED current. The voltage across  $R_3$  will be +5 V minus  $V_{CE}$  of the 2N2907 minus the diode voltage minus  $V_{CE}$  of the 2N2222. It is good engineering design to actually measure these voltages because the data sheets are only approximate. If the desired LED operating point is  $V_d$ ,  $I_d$ , select  $R_3 = (5 - V_{CE} - V_d - V_{CE})/I_d$ .

Assume the desired LED operating point is 10 mA, 2 V. Since the display will be scanned, we will operate each active LED at 30 mA with a 33% duty cycle. We will change the display every 5 ms. In this way, we will scan through the entire display every 15 ms, and each active LED will run at about 66 Hz, faster than the human eye can detect. The LED will “look like” it is continuously on at 10 mA (Figure 8.32).

**Figure 8.32**

Timing used to scan a LED interface.



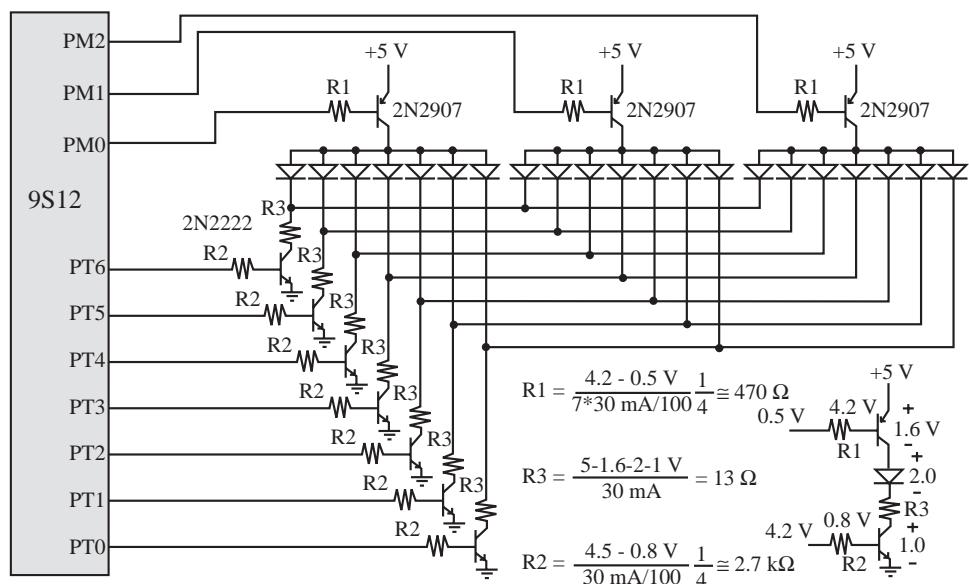
The software will have to convert the decimal digits into the seven-segment codes using Table 8.7. The 2N2222 NPN transistor can sink the required 30 mA. If all seven segments are on, the source current will be  $7 \cdot 30 \text{ mA}$ , or 210 mA. The 2N2907 PNP transistor can supply the necessary current (Figure 8.33). The 9S12 system uses Port M and Port T, but any ports can be used.

It is important to place the resistors on the segment-select side rather than on the digit-select side. In this way the current through each diode is not a function of the number of diodes on. The software that initializes/maintains the display is shown in Program 8.10.

We can also solve this problem using a decoder. The purpose of the previous example was to illustrate the details involved in scanning an LED display. For practical designs we will use special LED display decoder logic. The simplest of the LED drivers is the 7447 seven-segment decoder. Again we will design a three-digit LED display interfaced only to a single output port. The 21 LED segments will again be interfaced in a 3-column by 7-row rectangular matrix. There will be a column for each digit and a row for each of the segments a, b, c, d, e, f, and g. The open-collector outputs of the 7447 will sink current from the seven rows. The ULN2074, in open-emitter mode, will source current for the three columns. The software will utilize a 12-bit packed BCD global variable to contain the current value to be displayed. For example, if the value “456” is to be displayed, the main program will set the 16-bit global to \$0456 (Num). Our interrupt software will read this global and output the appropriate signals to the output port.

**Figure 8.33**

Circuit used to scan an LED interface.



```

        org $3800
Code   rmb 3 ;binary codes to output
Index  rmb 2 ;0,1,2 position
        org $4000
Select fcb 4,2,1 ;LED selection
LED_Init sei
    movw #0,Index
    movb #$FF,DDRT ;segment code
    bset DDRM,#$07 ;LED select
    bset TIE,#$20 ;arm OC5
    bset TIOS,#$20 ;output compare
    movb #$80,TSCR1 ;enable clock
    movb #$02,TSCR2 ;1 us
    ldd TCNT
    addd #50
    std TC5 ; first in 50us
    movb #$20,TFLG1 ;clr C3F
    cli
    rts
OC5han movb #$20,TFLG1 ;ack C5F
    ldd TC5
    addd #5000
    std TC5 ;every 5ms
    ldx Index
    movb Select,x,PTM ;which LED
    movb Code,x,PTT ;enable
    inx
    cpx #3
    blo skip
    ldx #0
skip  stx Index
    rti
    org $FFE4
    fdb OC5han ;vector
// PT7-PT0 output, 7 bit pattern
// PM2-PM0 output, selects LED digit
unsigned char Code[3]; // binary codes
const unsigned char Select[3]={4,2,1};
unsigned short Index; // 0,1,2
void LED_Init(void) {
asm sei // make atomic
Index = 0;
DDRT = 0xFF; // outputs 7 segment code
DDRM |= 0x07; // outputs select LED
TIE |= 0x20; // Arm OC5
TIOS |=0x20; // enable OC5
TSCR1 =0x80; // enable
TSCR2 =0x02; // 1us clock
TC5 = TCNT+10000;
asm cli
}
void interrupt 13 OC5han(void){
TFLG1 = 0x20; // Acknowledge
TC5 = TC5+5000; // every 5 ms
PTM = Select[Index]; // which LED?
PTT = Code[Index]; // enable
if(++Index==3) Index=0;
}

```

### Program 8.10

Software interface of a scanned LED display.

Again, we assume the desired LED operating point is 10 mA, 2 V. Since the display will be scanned, we will operate each active LED at 30 mA with a 33% duty cycle. The 7447A will simplify the generation of the seven-segment codes. This seven-segment LED driver can sink the required 30 mA. Just like the previous example, if all seven segments are on, the source current will be  $7 \cdot 30$  mA, or 210 mA (Figure 8.34). The software that initializes/maintains the display is shown in Program 8.11. The 9S12 system uses Port T.

<pre>         org \$3800 Num    rmb 2 ;12-bit packed BCD Index  rmb 2 ;0,2,4 position         org \$4000 Select fdb 4,2,1 ;LED selection Shift   fdb right,none,left LED_Init sei         movw #0,Index         movb #\$FF,DDRT ;LED outputs         movw #0,Num         bset TIE,#\$20 ;arm OC5         bset TIOS,#\$20 ;output compare         movb #\$80,TSCR1 ;enable clock         movb #\$02,TSCR2 ;1 us         ldd TCNT         addd #50         std TC5 ; first in 50us         movb #\$20,TFLG1 ;clr C3F         cli         rts right lsr d      ;shift right         lsr d         lsr d         lsr d         rts ;into bits 7-4 left   lsl d      ;shift left         lsl d         lsl d         lsl d         rts ;into bits 7-4 none   rts ;already there OC5han movb #\$20,TFLG1 ;ack C5F         ldd TC5         addd #5000         std TC5 ;every 5ms         ldx Index         ldd Num ;12-bit packed BCD         jsr Shift,x ;data in bits 7-4         andb #\$F0         addd Select,x ;bits2-0=4,2,1         stab PTT ;update display         leax 2,x         cpx #6 ;0,2,4         blo skip         ldx #0 skip   stx Index         rti         org \$FFE4         fdb OC5han ;vector </pre>	<pre> unsigned short Num; // 12-bit packed BCD const struct LED{     unsigned char enable; // select     unsigned char shift; // bits to shift     const struct LED *Next; }; // Link typedef const struct LED LEDType; typedef LEDType * LEDPtr; LEDType LEDTab[3]={ { 0x04, 8, &amp;LEDTab[1] }, // Most sig { 0x02, 4, &amp;LEDTab[2] }, { 0x01, 0, &amp;LEDTab[0] } }; // least sig LEDPtr Pt; // Points to current digit  void LED_Init(void) { asm sei           // make atomic         DDRT = 0xFF; // outputs to LED's         Num = 0;         Pt = &amp;LEDTab[0];         TIE  = 0x20; // Arm OC5         TIOS  = 0x20; // enable OC5         TSCR1 = 0x80; // enable         TSCR2 = 0x02; // 1us clock         TC5 = TCNT+50; asm cli }  // update one digit every 5ms // complete display update every 15ms void interrupt 13 TC5handler(void){         TFLG1 = 0x20; // Acknowledge         TC5 = TC5+5000; // every 5 ms         PTT = (Pt-&gt;enable)+(Num&gt;&gt;(Pt-&gt;shift))&lt;&lt;4;         Pt = Pt-&gt;Next; } </pre>
--	---

### Program 8.11

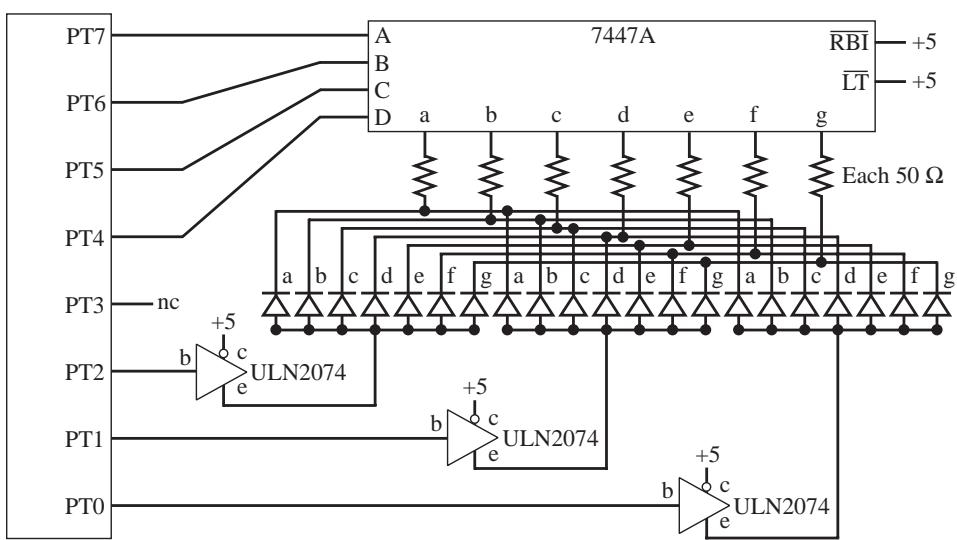
Software interface of a multiplexed LED display.

**Figure 8.34**

An encoder used to interface three seven-segment common-anode LED digits.

## 7447 LED ULN2074

$$R = \frac{5V - V_{OL} - 2V - V_{ce}}{30mA} = \frac{5 - 0.4 - 2 - 1.1V}{30mA} = 50\Omega$$



If more digits are needed, it would be easy to extend this approach by adding more ULN2074 current drivers. There are two issues to consider as the number of digits is added. The first is the scan frequency. For the display to “look” continuous, each digit must be updated faster than 60 Hz. (If you look closely into the specifications of your computer monitor and television sets, you will see this same constraint.) So as you increase the number of digits, the interrupt rate must increase so that each individual digit is scanned faster than 60 Hz. If there are five digits, then the periodic interrupt rate must be increased to at least 300 Hz for each digit to be updated at a rate of 60 Hz. The second issue to consider is the duty cycle for each digit. As the number of digits is increased, the duty cycle for each digit decreases. This makes it necessary to increase the instantaneous current. For example, if there were five digits, the duty cycle would be 20% and the current would have to increase to 50 mA. This design would require an open-collector driver capable of sinking 50 mA and an open-emitter capable of sourcing 350 mA. Another limitation is the maximum instantaneous current allowed by the LED. There is an upper bound to the instantaneous LED current even if the duty cycle decreases, leaving the average power the same. Each LED is different, but this parameter is about 100 mA, as shown in Table 8.6.

**Observation:** The ratio of the maximum instantaneous current divided by the desired LED current determines the maximum number of columns in the LED matrix.

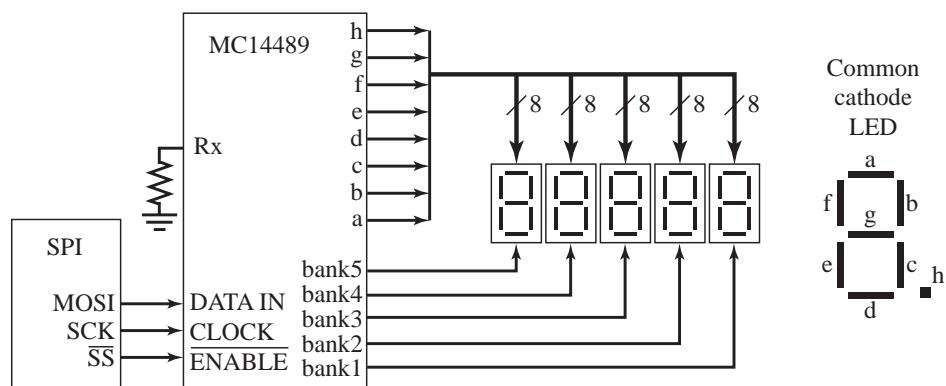
**Example 8.8** Design a five-decimal digit LED display using an MC14489 integrated driver.

**Solution** For LED displays with many segments, one attractive solution is to use an integrated LED display driver. There are three advantages of a chip like the MC14489 over the other LED designs in this section. The first advantage is chip count. A single MC14489 20-pin chip and one resistor are all the components required to interface a five-digit LED display. The second advantage is that the scanning functions occur in hardware, eliminating the need for the software to execute periodic interrupts. Normally, when we think of the

hardware/software tradeoff, we often select the software solution because of its low cost and flexibility. This is a situation where a hardware solution could be cheaper. The third advantage is that it is easy to cascade multiple MC14489 drivers so that larger displays can be interfaced without requiring additional microcomputer output ports. Each of the five eight-segment LED devices has seven segments ("a–g") for creating the decimal digit plus one more segment ("h") for the decimal point. The packed BCD format is shifted serially into the MC14489. The SPI port is a convenient solution for the hardware/software interface between the computer and MC14489(s). Once the BCD pattern is loaded into the MC14489, the hardware will continuously generate the 8 by 5 matrix scanning signals that display the five digits. The Rx resistor controls the LED voltage/current operating point for the display. The software also has the ability to select the LED brightness: "regular" or "dim." The SPI SS pin (PM3) will be configured as a simple output because the ENABLE pin will be low for 8 clock cycles when transmitting a command and for 24 clock cycles when transmitting data (Figure 8.35).

**Figure 8.35**

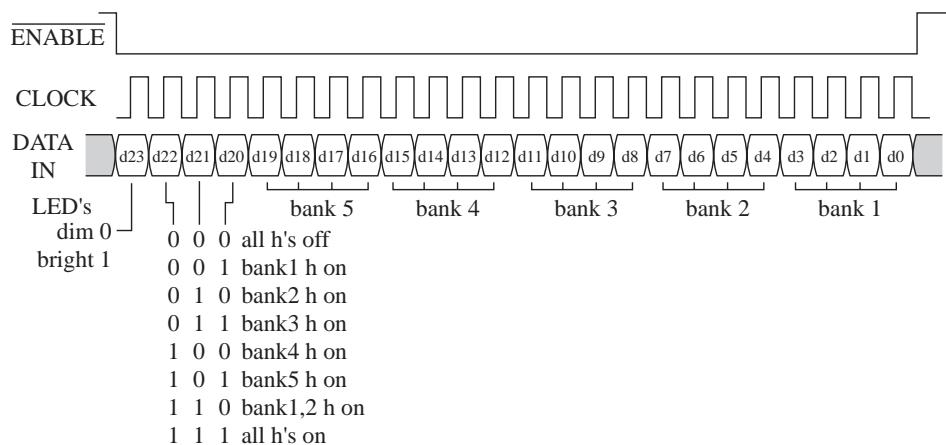
An integrated IC used to interface five seven-segment common-cathode LED digits.



The MC14489 can handle multiple formats. This interface utilizes a packed BCD format, as shown in Figure 8.36. The clock signal (output of computer, input to MC14489) is used to shift 24 bits of data into the LED display interface. For data to be properly transferred, the computer will change the data on the falling edge of the CLOCK and the MC14489 will shift the data on the rising edge. This timing can be achieved with the SPI in master mode and CPHA = 0, CPOL = 0. For ENABLE to be low for 24 bits, we will configure it as a simple output. The 20 bits that control the five banks are encoded as packed BCD. The MC14469

**Figure 8.36**

Data timing of an integrated LED controller.

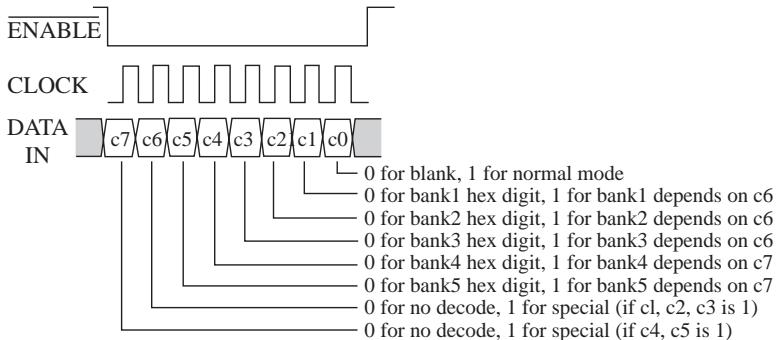


will also display BCD digits  $A_b c_d e_f$ , so the system can be used to show five decimal digits or five hexadecimal digits.

There is also an 8-bit command transmission that is used to configure the display. To implement the standard decimal (or hexadecimal) display, we will send a command of \$01. This enables the device and puts all five banks in hexadecimal format (Figure 8.37). The software to control the interface is simplified when using the SPI. The transmission rate is configured for 1 MHz, but it actually could operate as fast as 2 MHz. Interrupt synchronization is not needed because each 8-bit data frame requires only 8  $\mu s$  to complete (Program 8.12).

**Maintenance tip:** It will be more reliable to design the system using a transmission rate, somewhat slower than the absolute maximum.

**Figure 8.37**  
Configuration timing of an integrated LED controller.



<pre> spi    ;RegA both input and output data brclr SPISR,#\$20,* ;1. wait SPTEF staa  SPIDR        ;2. out data brclr SPISR,#\$80,* ;3. wait SPIF ldaa  SPISR        ;4. in data rts  LED_Init ;initialize display bset DDRM,#\$38      ;outputs movb #\$50,SPICR1   ;enable, master clr  SPICR2        ;regular drive movb #\$01,SPIBR    ;1MHz SCLK bset PTM,#\$08       ;ENABLE=1 bclr PTM,#\$08       ;ENABLE=0 ldaa #\$01          ;hex format bsr   spi bset PTM,#\$08       ;ENABLE=1 rts  LED_out ;Reg X points to data[3] bclr PTM,#\$08       ;ENABLE=0 ldaa 2,x bsr   spi           ;send MSbyte ldaa 1,x bsr   spi           ;send middle byte ldaa 0,x bsr   spi           ;send LSbyte bset PTM,#\$08       ;ENABLE=1 rts </pre>	<pre> // PM4/MOSI = MC14489 DATA IN // PM5/SCLK = MC14489 CLOCK IN // PM3 (simple output) = MC14489 ENABLE unsigned char spi(unsigned char code){     while((SPISR&amp;0x20)==0){}// wait SPTEF     SPIDR = code;           // data out     while((SPISR&amp;0x80)==0){}// wait SPIF     return SPIDR;           // clear SPIF } void LED_Init(void) {     DDRM  = 0x38; // outputs to MC14489     SPICR1 = 0x50; // enable, master     // CPOL=CPHA=0, no interrupts, msb first     SPICR2 = 0x00; // PM3 regular drive     SPIBR = 0x01; // 1MHz SCLK     PTM  = 0x08; // ENABLE=1     PTM &amp;=~0x08; // ENABLE=0     spi(0x01); // hex format     PTM  =0x08; // ENABLE=1 } void LED_out(unsigned char data[3]){     PTM &amp;=~0x08; // ENABLE=0     spi(data[2]); // send MSbyte     spi(data[1]); // send middle byte     spi(data[0]); // send LSbyte     PTM  =0x08; // ENABLE=1 } </pre>
---	---

### Program 8.12

Software interface of an integrated LED display.

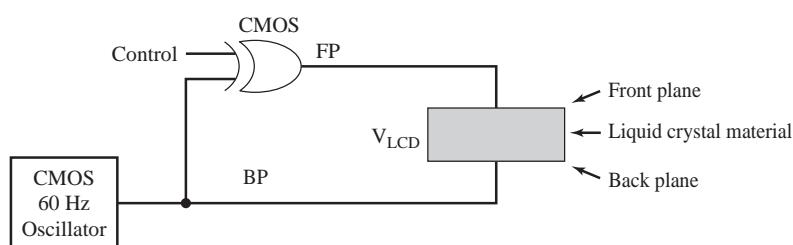
## 8.3 Liquid Crystal Displays

### 8.3.1 LCD Fundamentals

Liquid crystal displays are widely used in microcomputer systems (Figure 8.38). One advantage of LCDs over LEDs is their low power consumption. This allows the display, and perhaps the entire computer system, to be battery-operated. In addition, LCDs are more flexible in their sizes and shapes, permitting the combination of numbers, letters, words, and graphics to be driven with relatively simple interfaces. An LCD consists of a liquid crystal material that behaves electrically as a capacitor. Whereas an LED converts electric power into emitted optical power, an LCD uses an alternating current (AC) voltage to change the light reflectivity (or sometimes transmittivity). The light energy is supplied by the room or a separate back light, and not by the electric power within the LCD (as with LEDs). The computer controls the display by altering the reflectivity of each segment. The disadvantage of LCDs is their slow response time. Fortunately, the bandwidth of most displays (both LEDs and LCDs) is limited by the human visual processing system (about 30 Hz).

**Figure 8.38**

The basic idea of a liquid crystal interface.

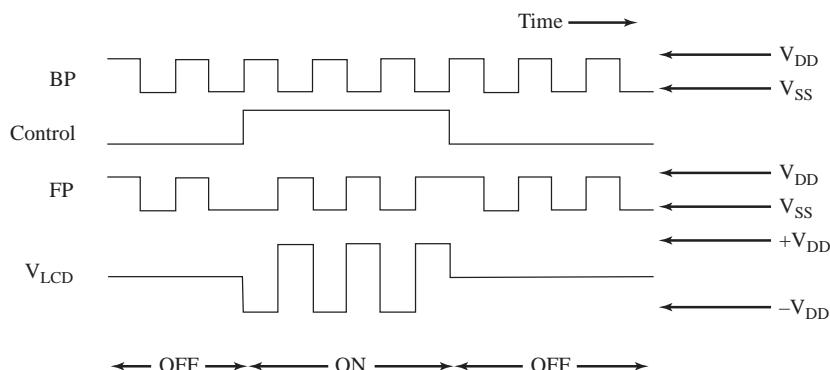


It is important to provide the LCD with only AC and no DC signals. A DC signal above 50 mV will cause permanent damage to the LCD. Let  $V_{DD}$  be the supply voltage of the CMOS logic, and let  $V_{SS}$  be the ground voltage of the CMOS logic. The use of CMOS logic has two advantages. First, CMOS requires very little supply current. Second, CMOS logic has fairly stable high and low logic voltage levels. In other words, the  $V_{OH}$  of the CMOS EOR gate and CMOS 60 Hz oscillator will both be close to  $V_{DD}$  (hence, close to each other). Similarly, the  $V_{OL}$  of the CMOS EOR gate and CMOS 60-Hz oscillator will also both be close to  $V_{SS}$  (hence, close to each other). Therefore, for both **Control** high and low,  $V_{LCD} = FP - BP$  will contain no DC component. Obviously, it will be important for the oscillator to have a 50% duty cycle.

The oscillator output **BP** is a square wave ( $V_{SS}$  to  $V_{DD}$ ) with a frequency of 60 Hz. When the **Control** signal is low, the front-plane voltage, **FP**, is in phase with the back-plane voltage, **BP**. Hence,  $V_{LCD}$  will be zero. In this state, the display does not reflect light, and the display is blank. When the **Control** signal is high, the front-plane voltage, **FP**, is out of phase with the back-plane voltage, **BP**. Hence,  $V_{LCD}$  will be an AC square wave ( $-V_{DD}$  to  $+V_{DD}$ ). In this state, the display reflects light, and the display is visible (Figure 8.39).

**Figure 8.39**

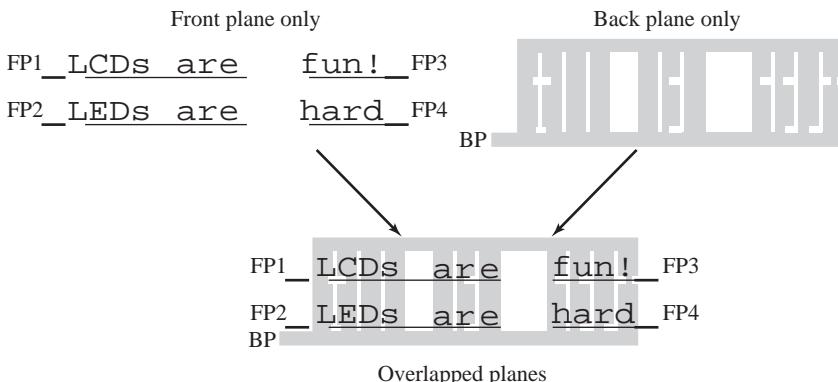
The basic timing of a liquid crystal interface.



**Observation:** LCDs are controlled with waveforms in the 40 to 60 Hz range.

One of the most important advantages of the LCD technology over lamps and LEDs is the flexibility in configuring the shapes and sizes of the segments. With LEDs the shapes of the segments are limited to simple regular shapes like circles and rectangles. Liquid crystal displays are created by sandwiching the liquid crystal material between a front and a back plane. The LCD segment is created in the overlap area of the front and back planes. Since the front and back planes are manufactured using techniques similar to PC board layout, there is a great flexibility in the sizes and shapes. In Figure 8.40 the entire phrase “LCDs are” represents a single segment, the overlap of FP1 and BP.

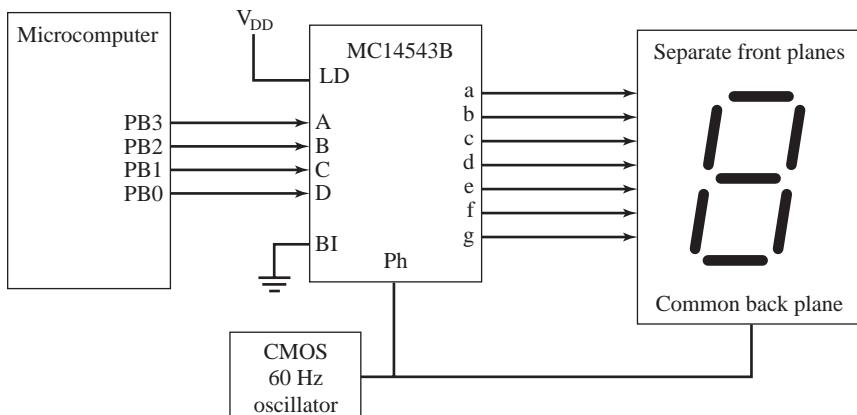
**Figure 8.40**  
Example artwork for an LCD.



### 8.3.2 Simple LCD Interface with the MC14543

The LCD interface (hardware and software) is similar to that for LEDs. The direct-driven interface is simple but requires a large number of output bits and cable wires. The MC14543 is a CMOS BCD to seven-segment LCD driver. When **LD** = 1, **BI** = 0, and **Ph** = 60 Hz square wave, one seven-segment LCD can be driven (Figure 8.41).

**Figure 8.41**  
Direct interface of an LCD.



The ability to latch data into the driver allows multiple LCDs to be interfaced with a single microcomputer output port. To latch one digit, the software outputs the 4-bit BCD on PB3-0, then toggles one of PB7-4 high, then low, as, for example, shown in Program 8.13.

**Program 8.13**  
Helper function for a simple LCD display.

```
void LCDOutDigit(unsigned char position, unsigned char data) {
    // position is 0x80, 0x40, 0x20, or 0x10 and data is the BCD digit
    PORTB = 0x0F&data; // set BCD digit on the A-D inputs of the MC14543B
    PORTB |= position; // toggle one of the LD inputs high
    PORTB = 0x0F&data; // LD=0, latch digit into MC14543B
}
```

To set all four digits, the software calculates the BCD from an unsigned input, as, for example, in Program 8.14.

#### Program 8.14

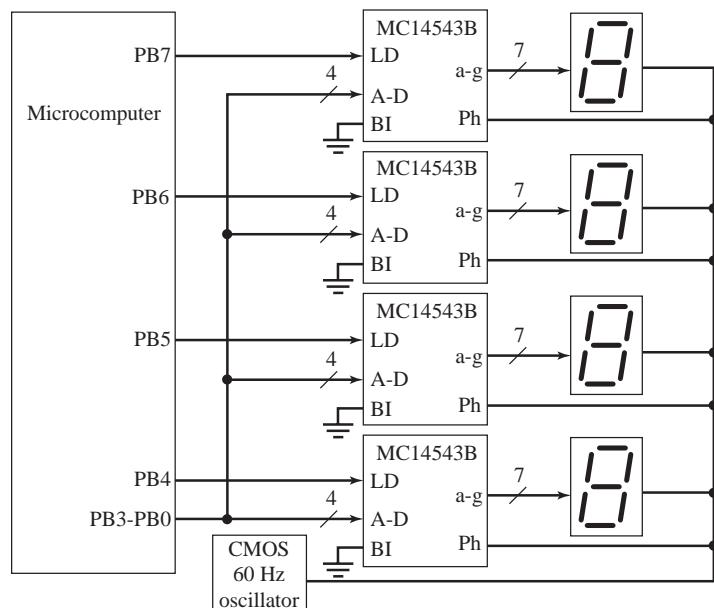
C software interface of a simple LCD display.

```
void LCD_OutNum(unsigned short data){ unsigned short digit,num,i;
unsigned char pos;
num = min(data,9999); // data should be unsigned from 0 to 9999
pos = 0x10; // position of first digit (ones)
for(i=0;i<4;i++){
    digit = num%10; num = num/10; // next BCD digit 0 to 9
    LCDOutDigit(pos,digit); pos = pos<<1;
}
}
```

The 9S12 requires a simple ritual that sets the direction register to outputs (e.g., DDRB=0xFF;). (See Figure 8.42.)

**Figure 8.42**

Latched interface of an LCD.



#### 8.3.3

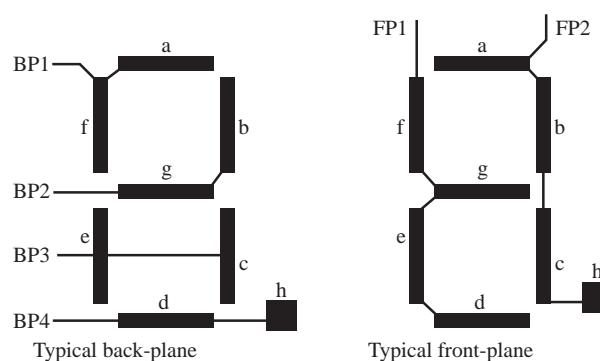
### Scanned LCD Interface with the MC145000, MC145001

Figure 8.43 shows a typical front-plane/back-plane configuration for an eight-segment LCD display. Different from LED displays, these LCD components do not have a common connection but rather utilize a 2 by 4 matrix. This configuration is compatible with LCD drivers like the MC145000 and MC145001.

Complex LCD artwork combines numbers, letters, words, and graphics on a single LCD. To simplify both the artwork and the interface, the front-plane and back-plane signals

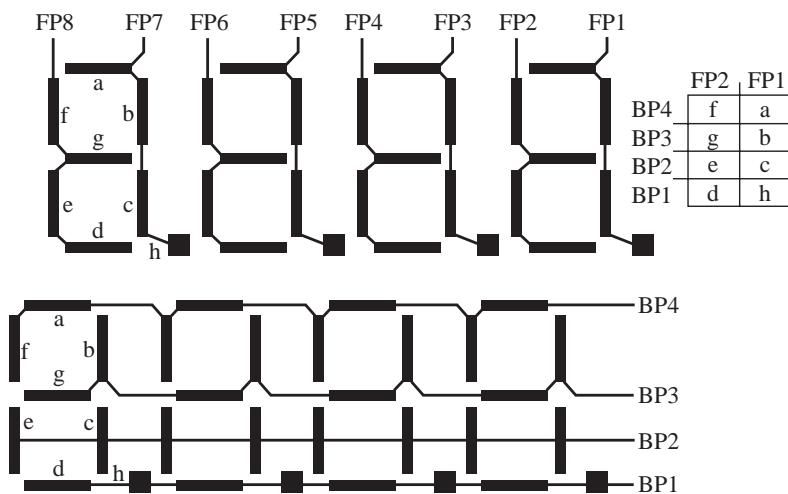
**Figure 8.43**

Artwork for an eight-segment liquid crystal digit.



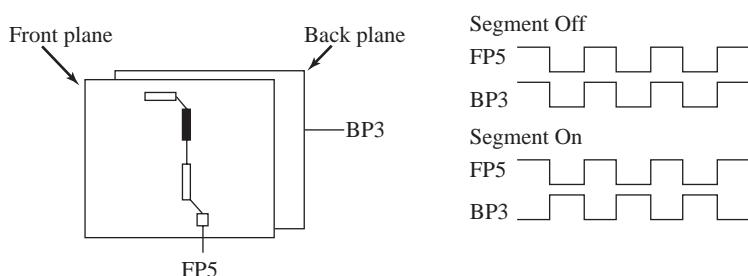
can be multiplexed. The concept is similar to the scanned LED displays described earlier. A four-digit display containing 32 segments can be multiplexed into four back-plane rows and eight front-plane columns (Figure 8.44). This reduces the number of cable wires from 32 (Figure 8.42) to 13 (Figure 8.44).

**Figure 8.44**  
Artwork for four eight-segment liquid crystal digits.



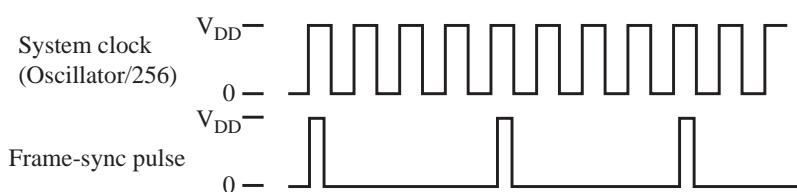
Consider the highlighted segment in Figure 8.45. To display this segment, an AC voltage should be applied across BP3, FP5. To hide this segment, no AC voltage should be applied across BP3, FP5. It is impossible to control all 32 segments in this simple manner. Fortunately, Freescale has developed chips to simplify the interfacing of multiple LCDs. One MC145000 master and three MC145001 slaves accept serial input from the microcomputer and will directly drive 20 LCD digits or 180 segments. These same chips (MC145000 and MC145001) could be used to drive a complex LCD display with numbers, letters, words, and graphics. The MC14500 master can control up to 48 segments organized in 4 back planes and 12 front planes. Each additional MC145001 slave can control up to 44 segments organized in 4 back planes and 11 front planes.

**Figure 8.45**  
Each segment has a unique front-plane/back-plane combination.



The MC145000 master creates the frame-sync pulses in Figure 8.46 each time the display is updated. The time-multiplexed back-plane voltages used to create the scanned

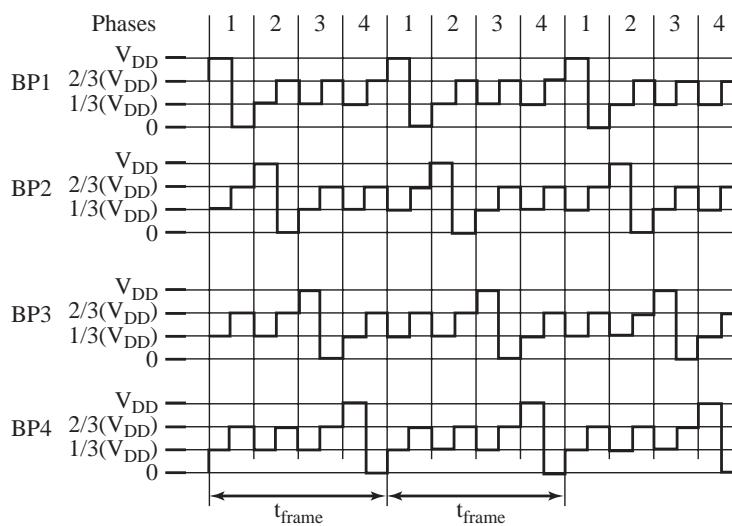
**Figure 8.46**  
Synchronization pulses for the LCD display.



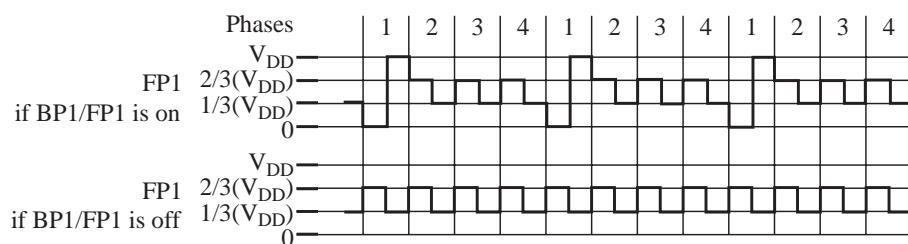
display are divided into four phases. During the first phase, the BP1 signal goes +5 V, then 0. If you wish any of the front-plane/BP1 segments to be active, the corresponding FP signal will go 0, then +5 V, during the first phase. This will create an AC signal large enough to activate the segment. Each of the BP signals has 25% of the time (when BP goes +5 V, then 0) to produce either an activation with the BP/FP pair (by making FP go 0, then +5 V) or a deactivation (by making FP go 3.33 V, then 1.67 V). In both the on and off cases, the DC component is zero. In the on situation, the BP to FP differential voltage is 5 to 0 V, then 0 to 5 V—that is, 5 V then -5 V. In the off situation, the BP to FP differential voltage is 5 to 3.33 V, then 0 to 1.67 V—that is, 1.67 V then -1.67 V. The  $\pm 5$  V AC signal is large enough to active the LCD, but the  $\pm 1.67$  V AC signal is not (Figures 8.47, 8.48).

**Figure 8.47**

Fixed waveforms for the four back-plane signals for the LCD display.

**Figure 8.48**

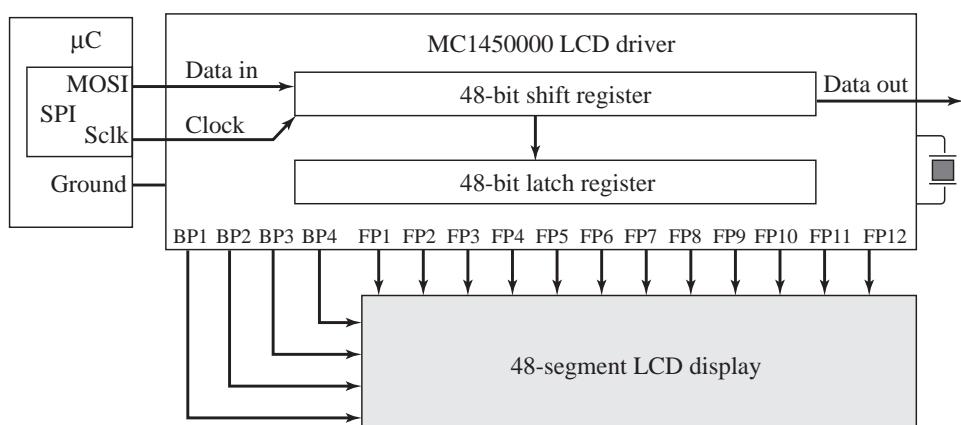
Variable waveforms for the front-plane signal for the LCD display.



The interface between the microcomputer and the LCD driver(s) utilizes a synchronized serial format (both serial data and serial clock). Most 9S12s have an SPI that automatically creates these signals. The computer interface is responsible for sending 48 bits in serial (one bit at a time). The SPI software simply outputs six bytes (most significant first), and the SPI hardware creates the MOSI data output and SCK clock. Once the information is loaded, the latch is activated, and the MC145000 LCD driver will maintain the display without software overhead (until the software wishes to change the display) (Figure 8.49).

The details of the SPI port were discussed in Chapter 7. Even without an SPI port, the LCD interface would be possible with two ordinary output bits. The SPI software to control this LCD interface is very similar to the LED interface using the MC14489 (Program 8.15).

**Figure 8.49**  
Interface of a  
48-segment  
LCD display.



<pre> LED_Init :initialize display     bset DDRM,#\$30      ;outputs     movb #\$50,SPICR1   ;enable, master     clr SPICR2         ;regular drive     movb #\$01,SPIBR    ;1MHz SCLK     rts ;Reg X points to data[6] LED_out     ldab #5 loop ldaa B,x     brclr SPISR,#\$20,* ;1. wait SPTEF     staa SPIDR        ;2. out data     brclr SPISR,#\$80,* ;3. wait SPIF     ldaa SPISR        ;4. clear SPIF     decb     bpl loop     ;5,4,3,2,1,0     rts </pre>	<pre> // PM4/MOSI = MC145000 DATA IN // PM5/SCLK = MC145000 CLOCK IN void LCD_Init(void) {     DDRM  = 0x30; // outputs     SPICR1 = 0x50; // enable,master     SPICR2 = 0x00; // PM3 not used     SPIBR = 0x01; // 1MHz SCLK } void LCD_out(unsigned char data[6]){     unsigned char j,dummy;     for(j=5; j&gt;=0; j--){         while((SPISR&amp;0x20)==0){}         SPIDR = data[j]; // Msbyte first         while((SPISR&amp;0x80)==0){}         dummy = SPIDR; // clear SPIF     } } </pre>
---	--

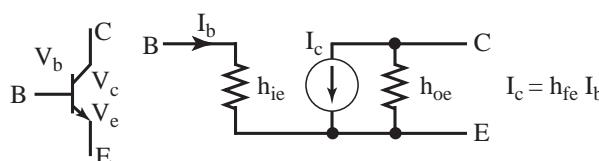
### Program 8.15

SPI interface to a scanned LCD display using an MC145000.

## 8.4 Transistors Used for Computer-Controlled Current Switches

We can use individual transistors to source or sink current. In this chapter the transistors are used in saturated mode. This means that when the NPN transistor is on, current flows from the collector to the emitter. When the NPN transistor is off, no current flows from the collector to the emitter. Each transistor has an input and output impedance,  $h_{ie}$  and  $h_{oe}$ , respectively. The current gain is  $h_{fe}$  or  $\beta$ , the model for the bipolar NPN transistor is shown in Figure 8.50.

**Figure 8.50**  
NPN transistor model.

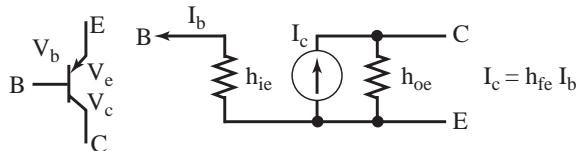


There are five basic design rules when using individual bipolar NPN transistors in saturated mode:

1. Normally  $V_c > V_e$ .
2. Current can flow only in the following directions: from base to emitter (input current), from collector to emitter (output current), and from base to collector (doesn't usually happen but could if  $V_b > V_c$ ).
3. Each transistor has maximum values for the following terms that should not be exceeded:  $I_b$ ,  $I_c$ ,  $V_{ce}$ , and  $I_c \cdot V_{ce}$ .
4. The transistor acts like a current amplifier:  $I_c = h_{fe} \cdot I_b$ .
5. The transistor will activate if  $V_b > V_e + V_{be(SAT)}$ , where  $V_{be(SAT)}$  is typically above 0.6 V.

The model for the bipolar PNP transistor is shown in Figure 8.51.

**Figure 8.51**  
PNP transistor model.



There are five basic design rules when using individual bipolar PNP transistors in saturated mode:

1. Normally  $V_e > V_c$ .
2. Current can flow only in the following directions: from emitter to base (input current), from emitter to collector (output current), and from collector to base (doesn't usually happen but could if  $V_c > V_b$ ).
3. Each transistor has maximum values for the following terms that should not be exceeded:  $I_b$ ,  $I_c$ ,  $V_{ce}$ , and  $I_c \cdot V_{ce}$ .
4. The transistor acts like a current amplifier:  $I_c = h_{fe} \cdot I_b$ .
5. The transistor will activate if  $V_b < V_e - V_{be(SAT)}$ , where  $V_{be(SAT)}$  is typically above 0.6 V.

**Performance tip:** A good transistor design is one in which the I/O response is independent of  $h_{fe}$ . We can design the interface so that  $I_b$  can be twice as large as needed to supply the necessary  $I_c$ .

Table 8.8 illustrates the wide range of bipolar transistors that we can use.

Type	NPN	PNP	Package	$V_{be(SAT)}$	$V_{ce(SAT)}$	$h_{fe}$ min/max	$I_c$
General purpose	2N3904	2N3906	TO-92	0.85 V	0.2 V	100	10 mA
General purpose	PN2222	PN2907	TO-92	1.2 V	0.3 V	100	150 mA
General purpose	2N2222	2N2907	TO-18	1.2 V	0.3 V	100/300	500 mA
Power transistor	TIP29A	TIP30A	TO-220	1.3 V	0.7 V	15/75	1 A
Power transistor	TIP31A	TIP32A	TO-220	1.8 V	1.2 V	25/50	3 A
Power transistor	TIP41A	TIP42A	TO-220	2.0 V	1.5 V	15/75	3 A
Power Darlington	TIP120	TIP125	TO-220	2.5 V	2.0 V	1000 min	3 A

**Table 8.8**

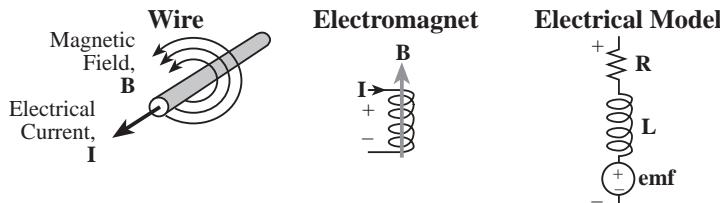
Parameters of typical transistors used by microcomputer to source or sink current.

## 8.5 Computer-Controlled Relays, Solenoids, and DC Motors

Relays, solenoids, and pulse-width modulated DC motors are grouped together because their electric interfaces are similar. In each case, there is a coil, and the computer must drive (or not drive) current through the coil. We can add speakers to this group if the sound is generated with a square wave. An *electromagnet* is created by winding a thin wire around a ferromagnetic core (Figure 8.52). An electromagnet converts electrical power into mechanical power. When current flows through the wire, a magnetic field is created, inducing an EM force on the system. We can model the coil as a series combination of a resistor (**R**), an inductor (**L**), and a battery (**emf**). The resistance in the coil (**R**) comes from the long wire that goes from the + terminal to the – terminal of the electromagnet. The inductance in the coil (**L**) arises from the fact that the wire is wound in a cylindrical pattern to create the electromagnet. The coil itself can generate its own voltage (**emf**) because of the interaction between the electric and magnetic fields. If the coil is a DC motor, then the **emf** is a function of both the speed of the motor and the developed torque (which in turn is a function of the applied load on the motor). Because of the internal **emf** of the coil, the current will depend on the mechanical load. Under a friction load, the **emf** will become negative and the current will increase. If external forces are applied, the DC motor can become a generator, converting mechanical power to electrical power.

**Figure 8.52**

The current through a coiled wire induces a magnetic field.



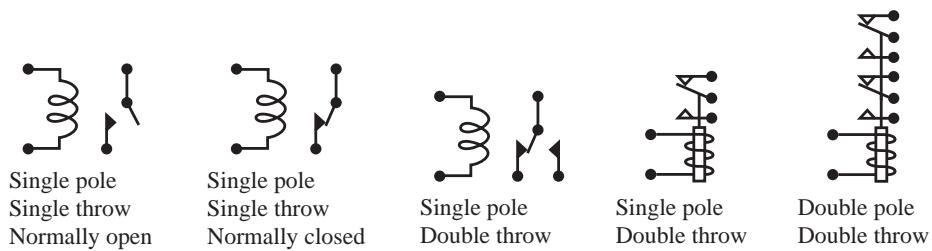
**Checkpoint 8.10:** A certain DC motor has a coil resistance of  $100\ \Omega$ . If 5 V is applied across this motor under no load ( $\text{emf} = 0$ ), a DC current of 50 mA will flow. Under a friction load, the **emf** jumps to  $-20\text{ V}$ . How much DC current will now flow?

### 8.5.1 Introduction to Relays

A relay is a device that responds to a small current or voltage change by activating switches or other devices in an electric circuit. It is used to remotely switch signals or power. The input control is usually electrically isolated from the output switch. The input signal determines whether the output switch is open or closed. Figure 8.53 shows typical circuit drawings of classic general-purpose electromagnetic relays. In each, the input current affects the position of the output switch.

**Figure 8.53**

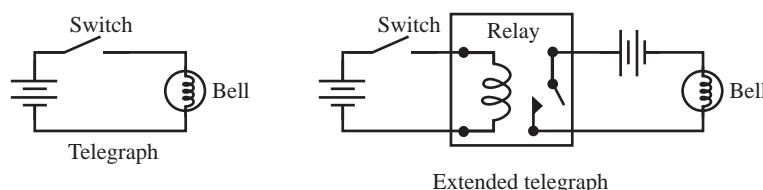
Various types of relays.



The *American Heritage Dictionary* defines *relay* as “an act of passing something along from one person, group, or station to another.” Electronic relays were originally used during the 19th century for extending the range of remote telegraph systems, *relying* the signal from one station to another (Figure 8.54).

**Figure 8.54**

Original application of relays.

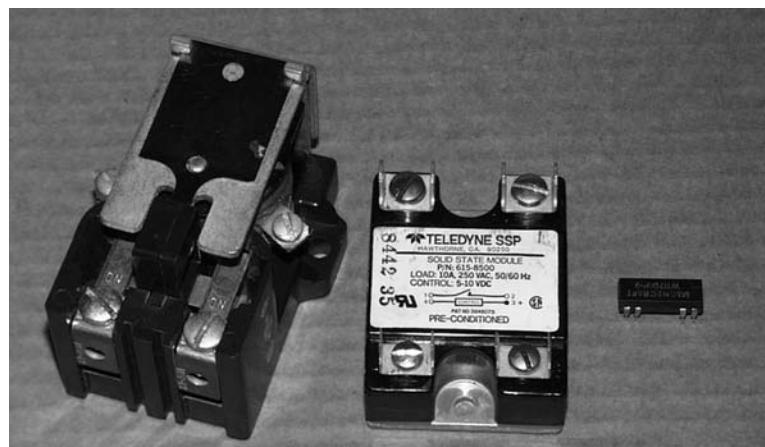


Relays are classified into four categories depending upon whether the output switches power (i.e., high currents through the switch) or electronic signals (i.e., low currents through the switch). Another difference is how the relay implements the switch. An electromagnetic (EM) relay uses a coil to apply EM force to a contact switch that physically opens and closes. The solid-state relay uses transistor switches made from solid-state components to electronically allow or prevent current flow across the switch. The four types (three shown in Figure 8.55) are the following.

- The classic general-purpose relay has an EM coil and can switch power.
- The reed relay has an EM coil and can switch low-level DC electronic signals.
- The solid-state relay (SSR) has an input-triggered semiconductor power switch.
- The bilateral switch uses CMOS, FET, or biFET transistors.

**Figure 8.55**

Photo of an EM, solid-state, and reed relay.



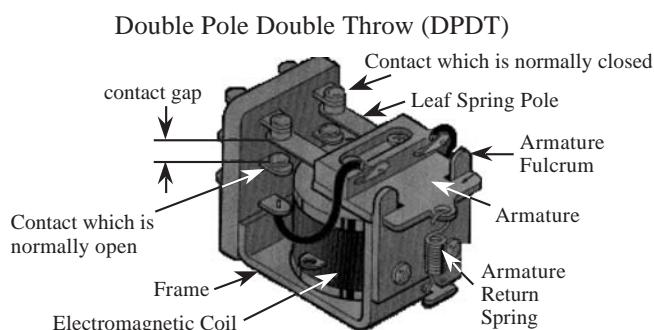
Actually, the bilateral switch is not a relay, but it is included in this discussion because it is a solid-state device that can be used to switch low-level signals.

## 8.5.2 Electromagnetic Relay Basics

Figure 8.56 illustrates a classical general-purpose EM relay. The input circuit is an EM coil with an iron core. The output switch includes two sets of silver or silver-alloy contacts (called *poles*). One set is fixed to the relay *frame*, and the other set is located at the end of

**Figure 8.56**

Drawing of an EM relay.



leaf spring poles connected to the *armature*. The contacts are held in the “normally closed” position by the armature return spring. When the input circuit energizes the EM coil, a “pull-in” force is applied to the armature and the “normally closed” contacts are released (called *break*) and the “normally open” contacts are connected (called *make*). The armature pull-in can either energize or deenergize the output circuit, depending on how it is wired. The illustrated relay has its transparent protective polycarbonate cover removed. Relays are mounted in special sockets, or directly soldered onto a printed circuit board.

The number of poles (e.g., single-pole, double-pole, 3P, 4P) refers to the number of switches that are controlled by the input. *Single throw* means each switch has two contacts that can be open or closed. *Double throw* means each switch has three contacts. The common contact will be connected to one of the other two contacts (but not both at the same time). Figure 8.53 shows relays of different types.

The parameters of a relay are specified in terms of the input coil and output switch (Table 8.9). The input parameters include DC or AC excitation, coil resistance, pickup voltage, dropout voltage, and maximum coil power. The *pickup voltage* is the coil potential above which activation is guaranteed. The *dropout voltage* is the coil potential below which deactivation is guaranteed. Typically the coil can handle sustained voltages a few volts above its nominal value before damage occurs. In microcomputer-based interfaces, DC coils are more convenient than AC coils. Unless there is an internal snubber diode, the polarity of the coil current does not matter.

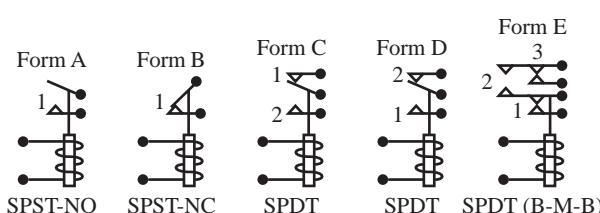
**Table 8.9**  
Parameters of four  
different types of  
computer-controlled  
switches.

Manufacturer Model number	Teledyne 712-5	Magnecraft W107DIP-2	Teledyne 611	PMI SW-01
Type	TO-5 relay	Reed	SSR	JFET
Coil resistance	50 Ω	500 Ω	500 Ω	3 μA input
Pickup voltage	3.6 V	3.8 V	3.8 V	2 V
Dropout voltage		0.5 V	0.8 V	0.8 V
Contact load	AC/DC	DC	AC/DC	DC
Max contact power		10 W DC	2500 W AC	
Max contact voltage	250 V AC	100 V DC	250 V AC	+11 to -10 V
Max contact current	600 mA AC	0.5 A DC	10 A AC	5 mA DC
On resistance	0.2 Ω	0.1 Ω	0.15 Ω	100 Ω
Off resistance	Infinite	Infinite	28 kΩ	58 dB
Turn-on time	4 ms	600 μs	8.3 ms	400 ns
Turn-off time	3 ms	75 μs	16.6 ms	300 ns
Life expectancy	$10^7$	$25 \cdot 10^6$	Infinite	Infinite
Approximate cost	\$2 to \$4	\$1 to \$2	\$10 to \$20	\$0.50-\$2

The parameters of the output switch include maximum AC (or DC) power, maximum current, maximum voltage, on resistance, and off resistance. A DC signal welds the contacts together at a lower current value than an AC signal; therefore the maximum ratings for DC are considerably smaller than for AC. Other relay parameters include turn-on time, turn-off time, life expectancy, and I/O isolation. *Life expectancy* is measured in number of operations. Storage life for relays is usually quite long. Table 8.5 lists specifications for four devices.

Figure 8.57 illustrates the various configurations available. The sequence of operation is listed in Table 8.10.

**Figure 8.57**  
Various relay  
configurations.



**Table 8.10**

Five relay configurations.

Form	Activation sequence	Deactivation sequence
A	Make 1	Break 1
B	Break 1	Make 1
C	Break 1, Make 2	Break 2, Make 1
D	Make 1, Break 2	Make 2, Break 1
E	Break 1, Make 2, Break 3	Make 3, Break 2, Make 1

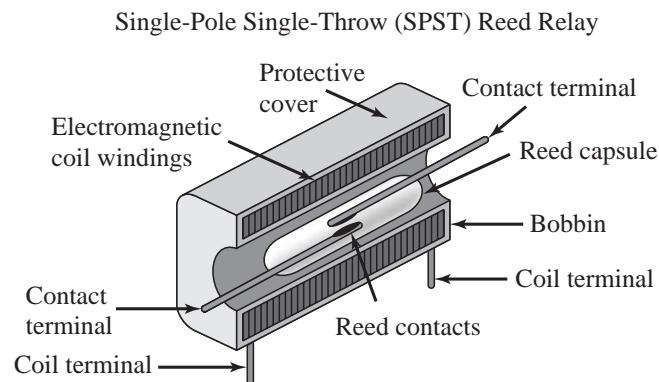
Latching relays do not require continuous input current to remain in their present state. A short pulse on the two-input coil (1 to 50 ms) causes the switch to change state, after which it will remain in the new state (open or closed) indefinitely. These devices are suitable for low-power operation when the contact is infrequently switched. They are also convenient for low-noise applications, because they do not require continuous coil currents. Mercury-wetted relays provide for faster switching and eliminate contact bounce. Some devices have an internal snubber diode and/or electrostatic shielding. The shielding reduces the noise crosstalk from the input coil to the external circuits. A shielded relay should be used if the relay is switching simultaneously with critical low-noise functions.

### 8.5.3 Reed Relays

Reed relays (Figure 8.58) are often used in medical electronics, telecommunications, and automated test equipment (ATE) for signal-level switching (e.g., mode/gain/offset selection). The single-pole-single-throw (SPST) reed capsule has two contacts that are normally open. When the coil is activated, an EM force causes the contacts to close.

**Figure 8.58**

Drawing of a reed relay.



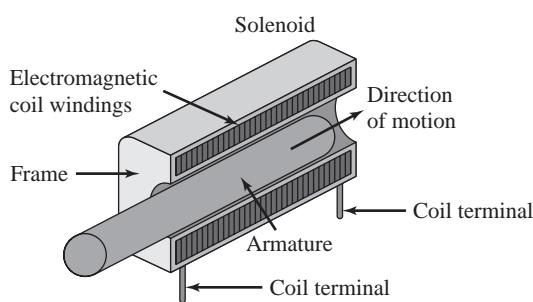
The coil is wound on a bobbin that surrounds the glass reed capsule. The reed capsule is hermetically sealed with inert gas. The contacts are typically made from precious metal like rhodium. Gold-cobalt contacts provide (at additional cost) reduced contact resistance (less than  $0.05\ \Omega$ ) for applications that require higher accuracy. Reed relays are available as DIPs or single inline packages (SIPs). The relay can be purchased as SPST-NO (Form A), SPST-NC (Form B), DPST-NO (Form 2A), and SPDT (Form C).

### 8.5.4 Solenoids

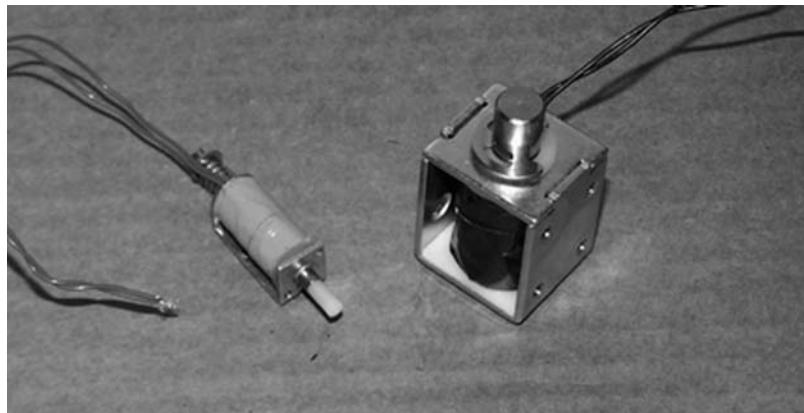
Solenoids are used in door locks, automatic disk/tape ejectors, and liquid/gas flow control valves (on/off type). Much like an EM relay, there is a frame that remains motionless and an armature that moves in a discrete fashion (on/off). A solenoid has an electromagnet (Figure 8.59 and Figure 8.60). When current flows through the coil, a magnetic force is created, causing a discrete motion of the armature. When the current is removed, the magnetic force stops, and the armature is free to move. The motion in the opposite direction can be produced by a spring, by gravity, or by a second solenoid.

**Figure 8.59**

Mechanical drawing of a solenoid showing that the EM coil causes the armature to move.

**Figure 8.60**

Photograph of two solenoids.



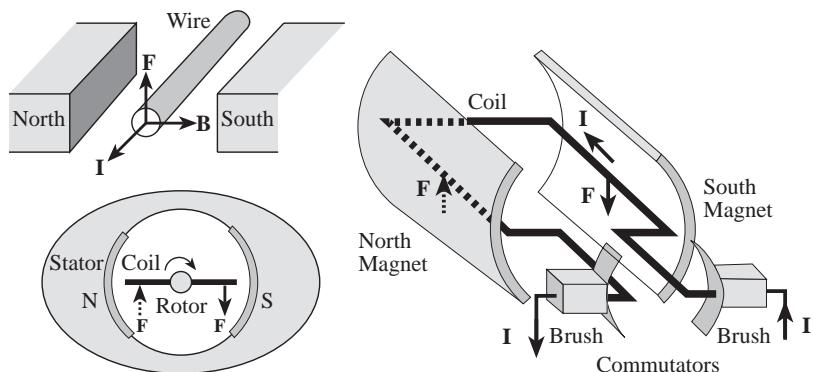
Courtesy of Jonathan Valvano.

### 8.5.5 Pulse-Width Modulated DC Motors

Similar to the solenoid and EM relay, the DC motor has a frame that remains motionless (called the *stator*), and an armature that moves (called the *rotor*). A *brushed DC motor* has an electromagnetic coil as well, located on the rotor, and the rotor is positioned inside the stator. In Figure 8.61, **North** and **South** refer to a permanent magnet, generating a constant **B** field from left to right. In this case, the rotor moves in a circular manner. When current flows through the coil, a magnetic force is created causing a rotation of the shaft. A brushed DC motor uses commutators to flip the direction of the current in the coil. In this way, the coil on the right always has an up force, and the one on the left always has a down force. Hence, a constant current generates a continuous rotation of the shaft. When the current is removed, the magnetic force stops, and the shaft is free to rotate. In a pulse-width-modulated DC motor, the computer activates the coil with a current of fixed magnitude but varies the duty cycle in order to adjust the power delivered to the motor. Software examples that generate variable duty-cycle waves were presented in Section 6.7.

**Figure 8.61**

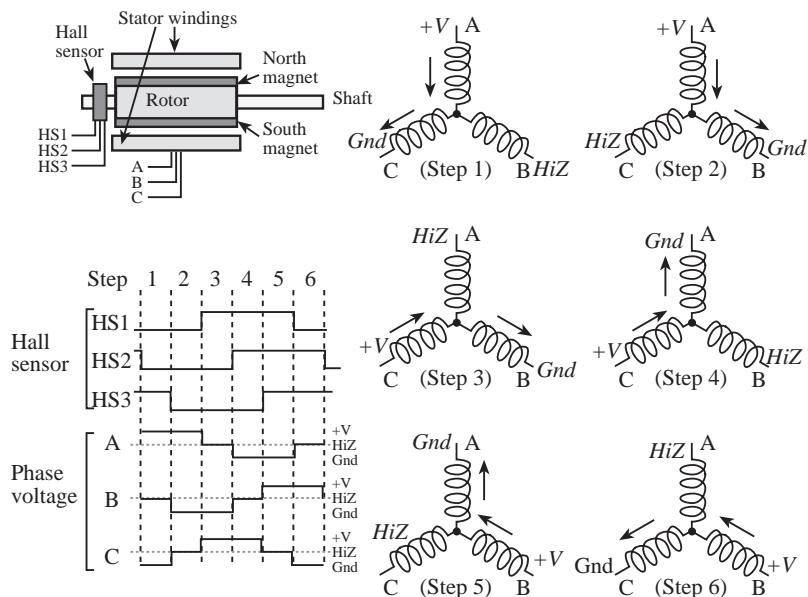
A brushed DC motor uses a commutator to flip the coil current.



A brushless DC *motor* (BLDC), as the name implies, does not have mechanical commutators or brushes to flip the currents. It is a synchronous electric motor powered by direct current and has an electronic commutation system—rather than a mechanical commutator and brushes. In BLDC motors, current-to-torque and voltage-to-rpm are linear relationships. The controller uses either the back emf of the motor itself or Hall-effect sensors to know the rotational angle of the shaft. The controller uses this angle to set the direction of the currents in the electromagnets, shown as the six-step sequence in Figure 8.62. Other differences from a brushed DC motor are that the BLDC permanent magnets are in the rotor and the electromagnets are in the stator. Typically, there are three electromagnetic coils, labeled Phase A, Phase B, and Phase C, which are arranged in a Wye formation. Each coil can be modeled as a resistance, inductance, and emf, as previously shown in Figure 8.52. The Hall sensor goes through the sequence 001, 000, 100, 110, 111, 011 each time the shaft rotates once. It is a synchronous motor because the controller adjusts the phase current according to the six-step sequence. For example, if the Hall sensor reads 001, then the controller places +V on Phase A and ground on Phase C (Step 1). In other words, the phase currents are synchronized to the shaft position. To rotate the motor in the other direction, we reverse the currents in each step. We will see later for stepper motors that the process is reversed. For stepper motors, the controller sets the phase currents, and the motor moves to that position. To adjust the power to a BLDC motor, we change the voltage, V, or use PWM on the control signals themselves. The PWM period should be at least 10 times shorter than the time for each of the six steps. In other words, the PWM frequency should be 60 times faster than the shaft rotational frequency.

**Figure 8.62**

A brushless DC motor uses an electronic commutator.

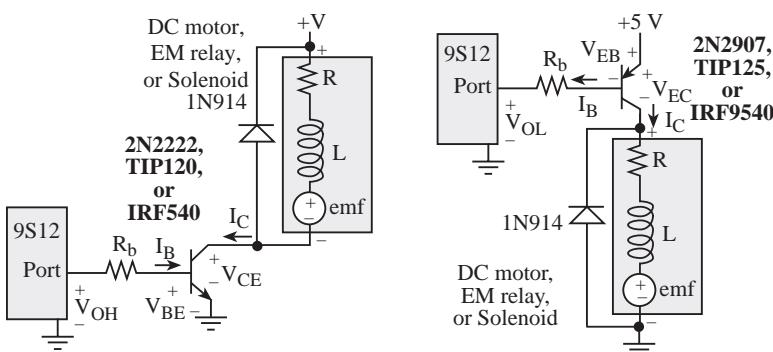


BLDC motors have many advantages and few disadvantages when compared to brushed DC motors. Because there are no brushes, they require less maintenance and hence have a longer life. Therefore, they are appropriate for applications where servicing is inconvenient or expensive. BLDC motors produce more output torque per weight than brushed DC motors and hence are used for pilotless airplanes and helicopters. Because the rotor is made of permanent magnets, the rotor inertia is less, allowing it to spin faster and to change quicker. In other words, it has faster acceleration and deceleration. Removing the brushes reduces friction, which also contributes to the improved speed and acceleration. It has a linear speed/torque relationship. Because there is no brush contact, BLDC motors operate more quietly and have less *Electromagnetic Interference* (EMI). The only disadvantages are the complex controller and increased cost.

### 8.5.6 Interfacing EM Relays, Solenoids, and DC Motors

The circuits in this section can be used to interface solenoids, EM relays, and DC motors because they all have electromagnets that behave like Figure 8.52. To interface an electromagnet, we consider **voltage**, **current**, and **inductance**. First, we need a power supply at the desired voltage requirement of the coil. If the only available power supply is larger than the desired coil voltage, we use a voltage regulator (rather than a resistor divider) to create the desired voltage. We connect the power supply to the positive terminal of the coil, shown as +V in Figure 8.63. We will use an NPN transistor to drive the negative side of the coil to ground. The other way to interface the coil is to use a PNP transistor to drive the positive side of the coil to +V. The computer can turn the current on and off by controlling the base of the transistor.

**Figure 8.63**  
Binary interface of an electromagnet.



The second consideration is current. In particular, we must select a power supply and an interface device that can support the desired coil current. The 2N2222 is an NPN bipolar junction transistor (BJT) with moderate current gain. The TIP120 is an NPN Darlington transistor that can handle up to 3 A. The IRF540 is a MOSFET transistor that can handle even more current (up to 28 A). BJT and Darlington transistors are current-controlled (meaning the output current is a function of the input current), while the MOSFET is voltage-controlled (output conductance is a function of input voltage). When interfacing a coil to the microcontroller, we use information as in Tables 8.5, 8.6, and 8.8 to select an interface device capable of conducting the current necessary to activate the coil ( $I_C$ ). It is a good design practice to select a driver with a maximum  $I_C$  of at least twice the required coil current.

**Observation:** It is important to realize that many devices cannot be connected directly up to the microcontroller. In the specific case of motors, we need an interface that can handle the voltage and current required by the motor.

The third consideration is inductance in the coil. The 1N914 diode in Figure 8.63 provides protection from the back emf generated when the switch is turned off, and the large  $dI/dt$  across the inductor induces a large voltage (on the negative terminal of the coil), according to  $V = L \cdot dI/dt$ . For example, if you are driving 0.1 A through a 0.1 mH coil (port output is high) using a 2N2222, then disable the driver (port output becomes low), and the 2N2222 will turn off in about 20 ns. This creates a  $dI/dt$  of at least  $5 \cdot 10^6$  A/s, producing a back emf of 500 V! The 1N914 diode shorts out this voltage, protecting the electronic from potential damage. The 1N914 is called a *snubber diode*. It is important that this *snubber diode* be fast.

Another option is to use an open-collector driver in place of the  $R_b$  resistor and NPN transistor in Figure 8.63. Table 8.4 lists some open-collector drivers. The output low current ( $I_{OL}$ ) of the driver should be sufficient to activate the electromagnet.

There are lots of motor driver chips, but they are fundamentally similar to the circuits shown in Figure 8.63. For the NPN BJT and Darlington transistors (left side of Figure 8.63), if the port output is low,  $V_{BE}$  will be zero, no current can flow into the base (so the transistor is off), and the collector current,  $I_C$ , will be zero. If the port output is high, current does flow into the base, and  $V_{BE}$  goes above  $V_{BEsat}$ , turning on the NPN transistor. The

transistor is in the linear range if  $V_{BE} > V_{BESat}$  and  $I_c = h_{fe} \cdot I_b$ . When controlling an electromagnet, we will have the transistor completely off or completely saturated and will not use the linear mode at all. The transistor is in the saturated mode if  $V_{BE} < V_{BESat}$ ,  $V_{CE} = 0.3V$ , and  $I_c < h_{fe} \cdot I_b$ . We select the base resistor ( $R_b$ ) for the NPN transistor interfaces to operate right at the transition between linear and saturated mode. We start with the desired coil current,  $I_{coil}$  (the voltage across the coil will be  $+V - V_{CE}$  which will be about  $+V - 0.3V$ ). Next, given the current gain of the NPN ( $h_{fe}$ ), we calculate the needed base current ( $I_b$ ) as

$$I_b = I_{coil} / h_{fe}$$

See Tables 8.8 and 8.11. Finally, given the output high voltage of the microcontroller ( $V_{OH}$  is about 5 V) and base-emitter voltage of the NPN ( $V_{BESat}$ ) needed to activate the transistor, we can calculate the desired interface resistor.

$$R_b = (V_{OH} - V_{BESat}) / I_b = h_{fe} * (V_{OH} - V_{BESat}) / I_{coil}$$

The inequality means we can choose a smaller resistor, creating a larger  $I_b$ . Because the  $h_{fe}$  of the transistors can vary a lot, it is a good design practice to make the  $R_b$  resistor about two or five times smaller than the value calculated in the above equation. This rule of thumb will guarantee the transistor goes into saturation when the port output is high. Since the transistor is saturated, the increased base current produces the same  $V_{CE}$  and thus the same coil current. Note only parameters change from sample to sample, but parameters are also dependent with the collector current (see Table 8.11). The coils and transistors can vary a lot, so it is appropriate to experimentally verify the design by measuring the voltages and currents.

**Checkpoint 8.11:** A DC motor is interfaced with the 2N2222 circuit in Figure 8.63.

The positive terminal of the motor is connected to +5 V and the motor requires a maximum of 150 mA. What  $R_b$  resistor would you use? What will be the voltage across the motor when active?

Parameter	2N2222 ( $I_C = 150$ mA)	2N2222 ( $I_C = 500$ mA)	TIP120 ( $I_C = 3$ A)
	2N2907 ( $I_C = 150$ mA)	2N2907 ( $I_C = 500$ mA)	TIP125 ( $I_C = 3$ A)
$h_{fe}$	100	40	1000
$V_{BESat}$	0.6	2	2.5 V
$V_{CE}$ at saturation	0.3	1	2 V

**Table 8.11**

Design parameters for the 2N2222 and TIP120.

The simple one PNP transistor circuit on the right side of Figure 8.63 works only if the motor voltage is +5 V. This is because we need zero volts across the emitter-base to shut the transistor off. In other words, if the port output is high,  $V_{EB}$  will be 0, and no collector current flows. If the port output is low, current does flow out of the base, and  $V_{EB}$  goes above  $V_{EBsat}$ , turning on the PNP transistor. We select the base resistor ( $R_b$ ) for the PNP transistor interfaces to supply enough base current to saturate the transistor. We start with the desired coil current,  $I_{coil}$  (the voltage across the coil will be  $+V - V_{CE}$  which will be about  $+V - 0.3V$ ). Next, given the current gain of the PNP ( $h_{fe}$ ), we calculate the needed base current ( $I_b$ ):

$$I_b = I_{coil} / h_{fe}$$

Finally, given the output low voltage of the microcontroller ( $V_{OL}$  is about 0 V) and emitter-based voltage of the PNP ( $V_{EBsat}$ ) needed to activate the transistor, we can calculate the desired interface resistor:

$$R_b = (5 - V_{EBsat} - V_{OL}) / I_b = h_{fe} * (5 - V_{EBsat} - V_{OL}) / I_{coil}$$

The inequality means we can choose a smaller resistor, creating a larger  $\mathbf{I}_b$ . Again, the rule of thumb is to make the  $\mathbf{R}_b$  resistor about two or five times smaller than the value calculated in the above equation.

Because of the resistance of the coil, there will not be significant  $dI/dt$  when the device is turned on. Consider a DC motor (as shown in Figure 8.63) with  $+V = 12\text{ V}$ ,  $\mathbf{R} = 50\ \Omega$ , and  $\mathbf{L} = 100\ \mu\text{H}$ . Assume we are using a 2N2222 with a  $V_{CE}$  of 1 V at saturation. Initially, the motor is off (no current to the motor). At time  $t = 0$ , the digital port goes from 0 to +5, and the transistor turns on. Assume for this section, the **emf** is zero (motor has no external torque applied to the shaft) and the transistor turns on instantaneously, we can derive an equation for the motor current ( $\mathbf{I}_c$ ) as a function of time. The voltage across both  $\mathbf{L}$  and  $\mathbf{R}$  together is  $12 - V_{CE} = 11\text{ V}$  at time  $t = 0^+$ . At time  $t = 0^+$ , the inductor is an open circuit. Conversely, at time  $t = -$ , the inductor is a short circuit. The  $\mathbf{I}_c$  at time  $0^-$  is 0, and the current will not change instantaneously because of the inductor. Thus, the  $\mathbf{I}_c$  is 0 at time  $= 0^+$ . The  $\mathbf{I}_c$  is  $11\text{ V}/50\ \Omega = 220\text{ mA}$  at  $t = -$ .

$$11\text{ V} = \mathbf{I}_c * \mathbf{R} + \mathbf{L} * d\mathbf{I}_c/dt$$

General solution to this differential equation is

$$\mathbf{I}_c = \mathbf{I}_0 + \mathbf{I}_1 e^{-t/\tau} \quad d\mathbf{I}_c/dt = -(\mathbf{I}_1/\tau) e^{-t/\tau}$$

We plug the general solution into the differential equation and boundary conditions:

$$11\text{ V} = (\mathbf{I}_0 + \mathbf{I}_1 e^{-t/\tau}) * \mathbf{R} - \mathbf{L} * (\mathbf{I}_1/\tau) e^{-t/\tau}$$

To solve the differential equation, the time constant will be  $\tau = \mathbf{L}/\mathbf{R} = 2\ \mu\text{sec}$ . Using initial conditions, we get

$$\mathbf{I}_c = 220\text{ mA} * (1 - e^{-t/2\mu\text{s}})$$

**Example 8.9** Design an interface for a +12-V, 1-A geared DC motor. The time constant of the motor is 100 ms.

**Solution** We will use the TIP120 circuit in Figure 8.63, because the TIP120 can sink at least three times the current needed for this motor. We select a +12 V supply and connect it to the +V in the circuit. The needed base current is

$$\mathbf{I}_b = \mathbf{I}_{coil}/h_{fe} = 1\text{ A}/1000 = 1\text{ mA}$$

The desired interface resistor is

$$\mathbf{R}_b = (V_{OH} - V_{be})/\mathbf{I}_b = (5 - 2.5)/1\text{ mA} = 2.5\text{ k}\Omega$$

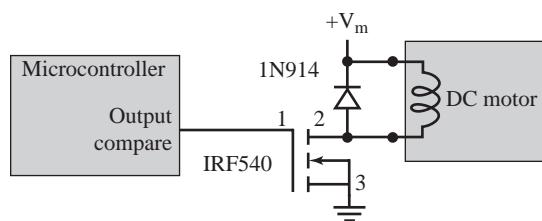
To cover the variability in  $h_{fe}$ , we will use a 1 k $\Omega$  resistor instead of the 2.5 k $\Omega$ . The actual voltage across the motor when active will be  $+12 - 2 = 10\text{ V}$ . We adjust the power to the motor using the PWM software shown as Program 6.15. The period of the PWM signal is selected as 10 times faster than the motor time constant. In particular, we choose a PWM period of 10 ms.

**Checkpoint 8.12:** Can the 9S12 output port supply the required 1 mA base current for the solution to Example 8.9?

For coils that require currents above 500 mA, a MOSFET can be used. The microcomputer output sources the current to the gate (pin 1) to turn on the MOSFET. Although the MOSFET gate does not need a lot of current to maintain the on state, large currents are required to switch it from off to on and from on to off (Figure 8.64). The voltage  $V_m$  is selected to match the specification of the motor. No resistor is needed between the port output and the gate of the MOSFET, but often we add a resistor (i.e.,  $\mathbf{R}_b = 1\text{ k}\Omega$ ) to limit

**Figure 8.64**

Motor interface using a high-current MOSFET.

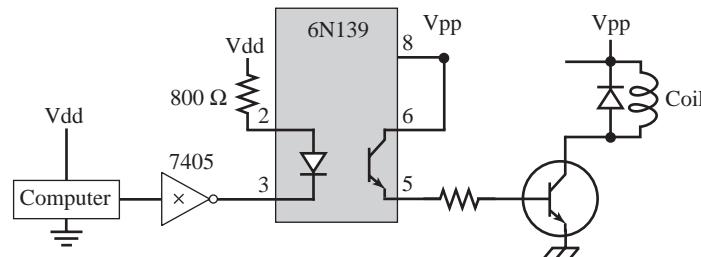


current into and out of the 9S12 during the turn on/off transients. At a gate-source voltage of 5 V, the drain current of the MOSFET will be only 5 to 10 A. To get the full 28 A output drive, the gate-source voltage needs to be above 10 V.

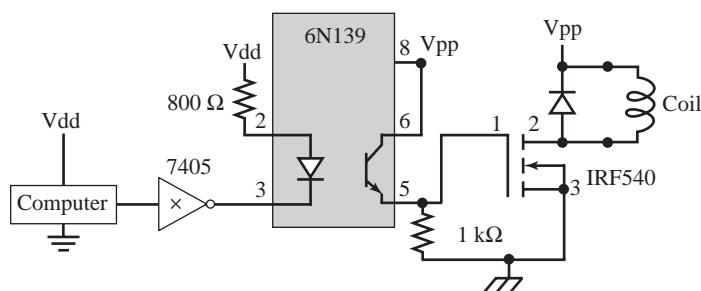
The relay by its nature isolates the computer and its driver electronics from the currents in the switch contacts. On the other hand, currents in the solenoid and DC motor must pass through the same ground as the computer. When these currents are large and noisy, we can decouple the coil currents from the computer ground using an optoisolator like the 6N139. Using electrical isolation can protect the computer electronics from surges in and around the motor. In the circuit of Figure 8.65, the computer ( $V_{dd}$ ) and motor ( $V_{pp}$ ) power are separate and their grounds are usually not connected. High-speed optoisolators, similar to these, can be used in computer networks to protect one computer from another. For more current, we can use the IRF540 MOSFET (Figures 8.66 and 8.67).

**Figure 8.65**

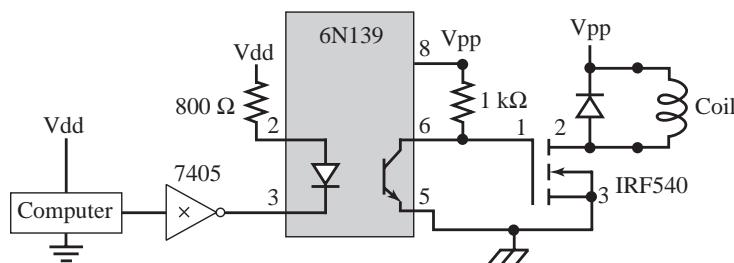
An isolated motor interface using a 6N139.

**Figure 8.66**

A high-current isolated motor interface using a 6N139 and a MOSFET.

**Figure 8.67**

Another high-current isolated motor interface using a 6N139 and a MOSFET.

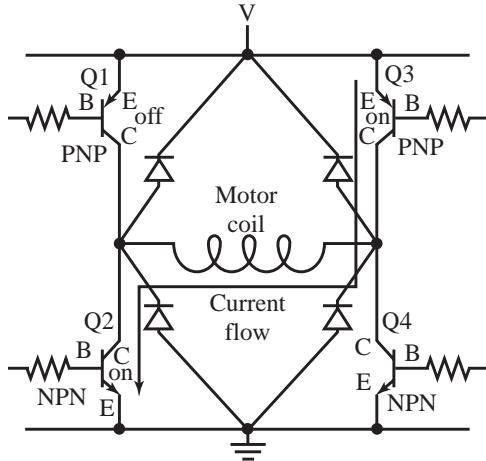


In some applications we wish to drive the motor forward and backward. To achieve this result we must be able to drive current both forward and backward through the motor coil. One approach to this interface is called the H-bridge. It is important not to

simultaneously drive both Q1, Q2 or both Q3, Q4. The basic approach to the H-bridge is illustrated in Figure 8.68. If Q2 and Q3 are on, then current flows right to left across the coil. If Q1 and Q4 are on, then current flows in the opposite direction. PNP transistors are used to source current into the coil, and NPN transistors are used to sink current out of the coil. Four diodes are required to prevent back EMF that occurs when the current is applied and removed.

**Figure 8.68**

An H-bridge is used to drive current in both directions.

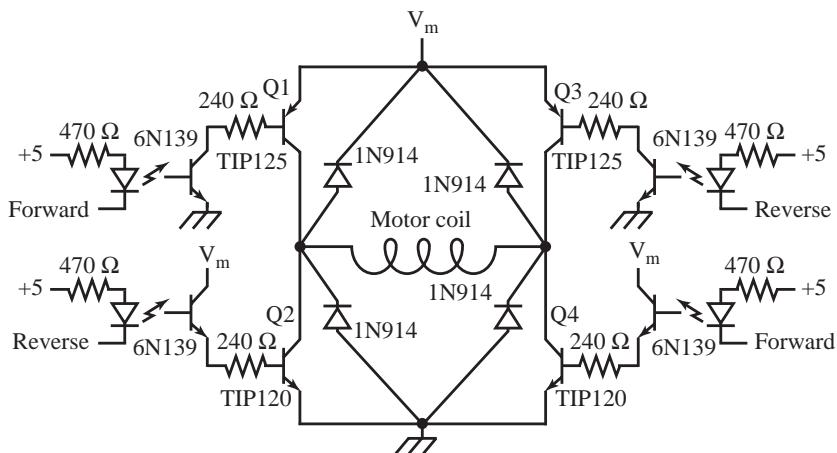


The TIP125 PNP Darlington transistors can source up to 3 A. To activate the PNP transistor, the base voltage must be 0.6 V less than the emitter voltage ( $V_m$ ). When light flows through its 6N139 isolation barrier, the 6N139 output goes to about 1 V, which drops the TIP125 base voltage low enough and sinks enough base current to activate the PNP source transistor. A low on the **Forward** signal will activate Q1 and Q4.

The TIP120 NPN Darlington transistors can sink up to 3 A. To activate the NPN transistor, the base voltage must be 0.6 V more than the emitter voltage (ground). When light flows through its 6N139 isolation barrier, the 6N139 output goes to about  $V_m - 1$  V, which raises the TIP120 base voltage high enough and sources enough base current to activate the NPN sink transistor. A low on the **Reverse** signal will activate Q2 and Q3 (Figure 8.69).

**Figure 8.69**

An isolated H-bridge can drive current in both directions.

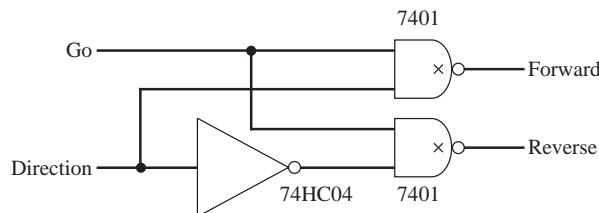


As mentioned earlier it is important not to activate both **Forward** and **Reverse** at the same time. To prevent this situation even when the software crashes, the digital interface

shown in Figure 8.70 could be used. To prevent temporary current paths through Q1 + Q2 or Q3 + Q4, the software should change the **Direction** only while the motor has been stopped ( $G_o = 0$ ) for enough time to allow all transistors to turn off.

**Figure 8.70**

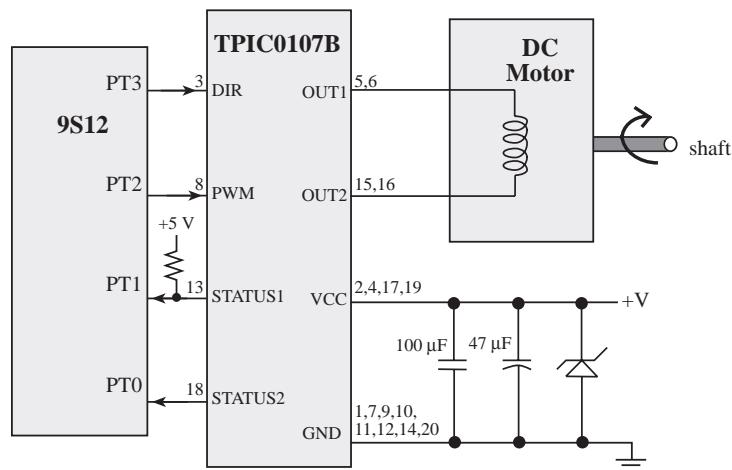
A digital circuit used with an H-bridge.



The classic approach to control a bidirectional DC motor is to use an H-bridge driver. We have the choice of creating an H-bridge with individual transistors, as shown in Figure 8.68, or using an integrated circuit. The advantage of using an integrated circuit is reduced cost, smaller size, and faster design. The TPIC0107B is an integrated H-bridge that is optimized for PWM input and reversible DC motor control. It can source/sink up to 3 A. The TPIC0107B is made with a CMOS digital logic and Double-Diffused Metal-Oxide Semiconductor (DMOS) logic for switching power to the motor. It has an extremely low on resistance, 280 mΩ typical, to minimize system power dissipation. Power is applied to the motor when the PWM input is high. Figure 8.71 shows the interface such that the PWM pin is connected to a pulse-width-modulated output, DIR pin is connected to a regular output, and STATUS1, STATUS2 are connected to regular inputs. STATUS1 requires a pull-up resistor because it is an open-collector. The DIR input controls the CW/CCW direction. Program 6.15 implements a PWM output that can be used to control the speed of this DC motor. The TPIC0107B provides protection against over-voltage, over-current, over-temperature, and cross-conduction faults. Fault diagnostics can be obtained by monitoring the STATUS1 and STATUS2 signals. Allegro also produces a complete line of DC motor controllers (e.g., A3936, A3949).

**Figure 8.71**

Bidirectional DC motor interface.



**Checkpoint 8.13:** What changes do you make to use the circuit in Figure 8.71 to control a 0.5 A, 12 V DC motor?

**Example 8.10** Interface a 24 V, 2 A brushless DC motor.

**Solution** A brushless DC motor has three coils connected in a Wye pattern. Each of the phases can be driven into one of three states: 24 V, ground, or floating. We will use MOSFETs to source and sink the current required by the motor (Figure 8.72). Remember, when the motor

is under load, the current will increase. The P-channel MOSFET will connect the 24 V to the phase when its gate voltage is below 24 V. The N-channel MOSFET will drive the phase to ground when its gate is above zero. It will be important to prevent turning on both MOSFETs at the same time. For safety reasons, we will use digital logic in the interface so the driver can be in the only three valid states. Table 8.12 shows the design specification for Phase A. When **EnA** is low, both MOSFETs are off, and the phase will float (HiZ). When **EnA** is high, the **InA** determines whether the phase is high or low. The six gate voltages are labeled in Figure 8.72 as **G1** to **G6**. These gate voltages are 24 V, produced by the 10 kΩ pull-up, when the corresponding 7406 driver output is floating. Alternatively, these gate voltages are 0.5 V when the 7406 driver output is low. It is good design to use integrated drivers like the ULN2074, L293, TPIC0107, and MC3479 rather than individual transistors. In particular, the entire interface circuit in Figure 8.72 could be replaced with three L6203 full-bridge drivers.

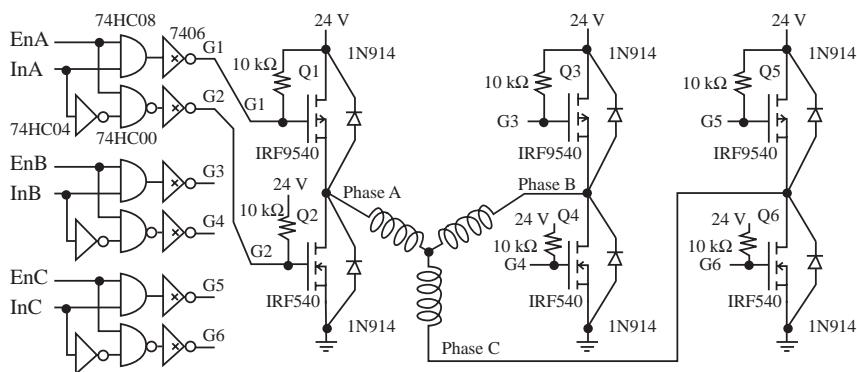
**Table 8.12**

Control signals for one phase of the brushless DC motor.

<b>EnA</b>	<b>InA</b>	<b>G1</b>	<b>G2</b>	<b>P-channel</b>	<b>N-channel</b>	<b>Phase A</b>
1	1	Low	Low	On	Off	+24 V
1	0	High	High	Off	On	Ground
0	X	High	Low	Off	Off	HiZ

**Figure 8.72**

Brushless DC motor interface.



The **InA**, **InB**, and **InC** signals in Figure 8.72 are connected to any output ports of the 9S12, whereas the **EnA**, **EnB**, and **EnC** signals will be attached to PWM outputs. The PWM period will be selected 60 times faster than the motor speed in rps. The three Hall-effect sensor signals will be attached to input capture pins (See Figure 8.62). Interrupts will be armed for both the rise and the fall of these three sensors. In this way, an ISR will be run at the beginning of each of the six steps. The BLDC motor is a synchronous motor, so the six control signals are a function of the shaft position. In particular, the ISR will lookup the Hall sensors and output the pattern, as shown in Table 8.13. Furthermore, when any of the enable

<b>Step</b>	<b>HS1</b>	<b>HS2</b>	<b>HS3</b>	<b>EnA</b>	<b>InA</b>	<b>EnB</b>	<b>InB</b>	<b>EnC</b>	<b>InC</b>	<b>A</b>	<b>B</b>	<b>C</b>
1	0	0	1	PWM	1	0	X	PWM	0	24 V	HiZ	0 V
2	0	0	0	PWM	1	PWM	0	0	X	24 V	0 V	HiZ
3	1	0	0	0	X	PWM	0	PWM	1	HiZ	0 V	24 V
4	1	1	0	PWM	0	0	X	PWM	1	0 V	HiZ	24 V
5	1	1	1	PWM	0	PWM	1	0	X	0 V	24 V	HiZ
6	0	1	1	0	X	PWM	1	PWM	0	HiZ	24 V	0 V

**Table 8.13**

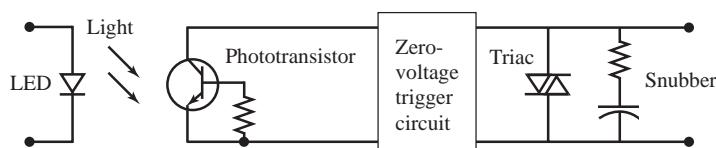
Input-output relationships for the synchronous controller.

signals are scheduled to be high, they will be pulsed using positive logic PWM. The software can adjust the delivered power to the BLDC motor by setting the duty cycle of the PWM. The software implementation has been left as Lab 8.4.

### 8.5.7 Solid-State Relays

To solve the limited life expectancy and contact bounce problems, SSRs were developed. Figure 8.73 illustrates the major components of an SSR. Figure 8.74 is a photograph of two SSRs. The SSR has no moving parts. The optocoupler provides isolation between the input circuit (pseudocoil) and the triac (pseudocontact). The switch function is constructed from either two inverse-parallel SCRs or an electrically equivalent triac. In the next section we will use silicon devices for switching DC signals that use power bipolar transistors or MOSFETs.

**Figure 8.73**  
Internal components of an SSR.



**Figure 8.74**  
Photograph of two SSRs.



Courtesy of Jonathan Valvano.

The signal from the phototransistor triggers the output triac so that it switches the load current. The zero-voltage detector triggers the triac only when the AC voltage is zero, reducing the surge currents when the triac is switched. Surge currents can occur when controlling capacitive loads like lamps. Once triggered, the triac conducts until the next zero crossing. If the input signal continues, then the triac will continuously conduct. The RC circuit is called a snubber and is used to reduce the voltage transients that occur when switching inductive loads like motors and solenoids. SSRs cost five to ten times more than general-purpose relays but have the following advantages because there are no moving parts:

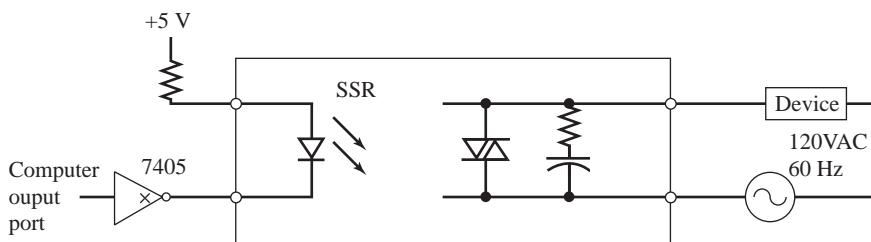
- Longer life
- Higher reliability
- Faster switching (although it occurs on the AC load zero crossing)
- Better mechanical stability
- Insensitive to vibrations and shock
- No contact bounce
- Reduces electromagnetic interference
- Quieter
- Eliminates contact arcing (can be used in the presence of explosive gases)

Interfacing the SSR to the microcomputer is identical to interfacing the LED (Figure 8.75).

The value of the resistor, R, is chosen to select the  $V_d$ ,  $I_d$  operating point of the LED.

$$R = \frac{+5 - V_d - V_{OL}}{I_d}$$

**Figure 8.75**  
Interface of an SSR.

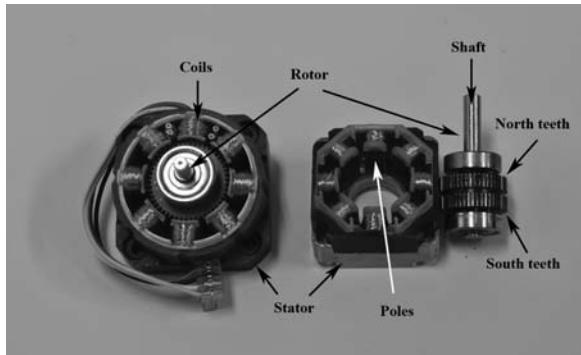


where  $V_{OL}$  is the output low voltage of the open-collector driver (7405). Again, the output low current ( $I_{OL}$ ) of the driver should be sufficient to activate the LED (e.g., 7406, 75492, 75451, ULN2074).

## 8.6 Stepper Motors

Stepper motors are very popular for microcomputer-controlled machines because of their inherent digital interface. It is easy for a microcomputer to control both the position and the velocity of a stepper motor in an open-loop fashion. Although the cost of a stepper motor is typically higher than that of an equivalent DC permanent magnetic field motor, the overall system cost is reduced because stepper motors may not require feedback sensors. For these reasons, they are used in many computer peripherals such as hard disk drives, floppy disk drives, and printers. Stepper motors can also be used as shaft encoders, measuring both position and speed (Figure 8.76).

**Figure 8.76**  
Stepper motors have permanent magnets on the rotor and electromagnetics around the stator.



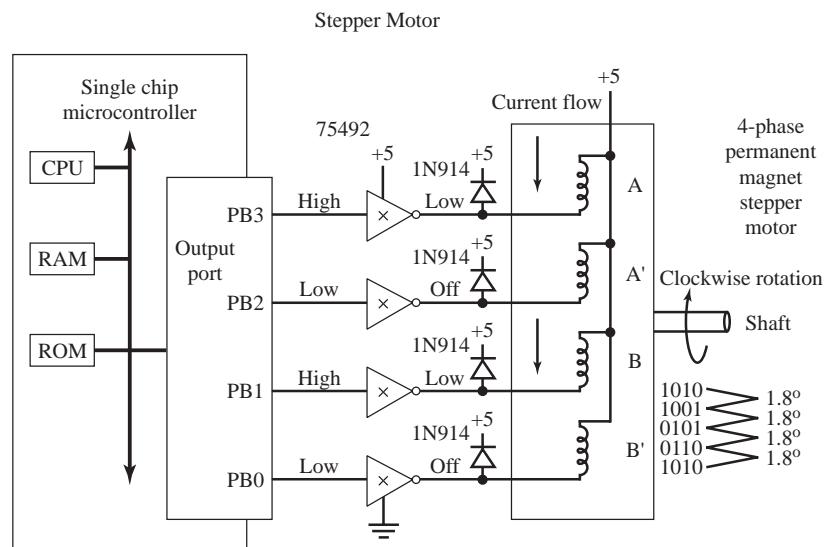
Courtesy of Jonathan Valvano.

### Example 8.11 Interface a 5 V, 70 mA unipolar stepper motor.

**Solution** In this example, we will introduce stepper motors by showing a simple example. The computer can make the motor spin by outputting the sequence . . . 10, 9, 5, 6, 10, 9, 5, 6 . . . over and over. For a motor with 200 steps per revolution, each new output will cause the motor to rotate  $1.8^\circ$ . If the time between outputs is fixed at  $\Delta t$  s, then the shaft rotation speed will be  $0.005/\Delta t$  in revolutions per second (rps). In each system, we will connect the stepper motor to the least significant bits of output PORTB. In this first hardware interface, we will connect a unipolar stepper with four 5 V coils, A, A', B, and B'. Because the coil resistance is about  $70\ \Omega$ , it will require about  $5\text{ V}/70\ \Omega$ , or 70 mA, to activate. The output low current of the 75492 is sufficient to sink the 70 mA. The 1N914 diodes will protect the 75492 from the back EMF that will develop across the coil when the current is shut off. Figure 8.77 illustrates the state when 10 (binary 1010) is output to PORTB. Bits PB3 and PB1 are high, making their 75492 outputs low and driving current through coils A and B. Bits PB2 and PB0 are low, making their 75492 outputs off (HiZ), and driving no current through coils A' and B'.

**Figure 8.77**

Simple stepper interface.



We define the active state of the coil when current is flowing. The basic operation is summarized in Table 8.14.

**Table 8.14**

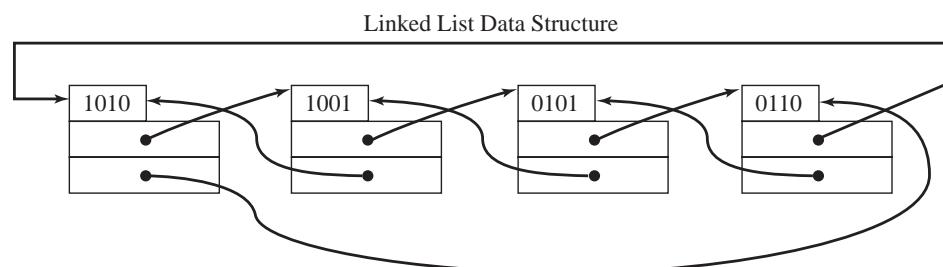
Stepper motor sequence.

PORTE output	A	A'	B	B'
10	Activate	Deactivate	Activate	Deactivate
9	Activate	Deactivate	Deactivate	Activate
5	Deactivate	Activate	Deactivate	Activate
6	Deactivate	Activate	Activate	Deactivate

We will implement a linked list approach to the stepper motor control software. This approach yields a solution that is easy to understand and change. If the computer outputs the sequence backward, then the motor will spin in the other direction. To ensure proper operation, this . . . 10, 9, 5, 6, 10, 9, 5, 6 . . . sequence must be followed. For example, assume the computer outputs . . . 9, 5, 6, 10, and 9. Now it wishes to reverse direction, since the output is already at 9; then it should begin at 10, and continue with 6, 5, 9. . . . In other words, if the current output is “9,” then the only two valid next outputs would be “5” if it wanted to spin clockwise or “10” if it wanted to spin counterclockwise. Maintaining this proper sequence will be simplified by implementing a double circular linked list. For each node in the linked list there are two valid next states, depending upon whether the computer wishes to spin clockwise or counterclockwise (Figure 8.78).

**Figure 8.78**

A double circular linked list used to control the stepper motor.



The sequence 10, 9, 5, 6 is called *full stepping* and each output causes the motor to move  $\Delta\theta$ . The sequence 10, 8, 9, 1, 5, 4, 6, 2 is called *half stepping* and each output causes the motor to move  $\Delta\theta/2$ .

A slip is when the computer issues a sequence change, but the motor does not move. A slip can occur if the mechanical load on the shaft exceeds the available torque of the motor. A slip can also occur if the computer tries to change the outputs too fast. If the system knows the initial shaft angle, and the motor never slips, then the computer can control both the shaft speed and angle without a position sensor. The routines CW and CCW will step the motor once in the clockwise and counterclockwise directions, respectively. If every time the computer calls CW or CCW it were to wait for 5 ms, then the motor would spin at 1 rps. The linked list data structure will be stored in EEPROM, and the variables are allocated into RAM and initialized at run time in the ritual (Program 8.16).

<pre>;Linked list stored in EEPROM S10  fcb 10      ;Output pattern       fdb S9      ;Next if CW       fdb S6      ;Next if CCW S9   fcb 9       fdb S10       fdb S5 S5   fcb 5       fdb S9       fdb S6 S6   fcb 6       fdb S5       fdb S10 ;Global variables stored in RAM Pos  ds  1       ;0&lt;= Pos &lt;=199 Pt   ds  2       ;to current state</pre>	<pre>const struct State{     unsigned char Out;           // Output     const struct State *Next[2]; // CW/CCW };  typedef struct State StateType; typedef StateType *StatePtr; #define clockwise 0             // Next index #define counterclockwise 1      // Next index StateType fsm[4]={     {10,{&amp;fsm[1],&amp;fsm[3]}},     { 9,{&amp;fsm[2],&amp;fsm[0]}},     { 5,{&amp;fsm[3],&amp;fsm[1]}},     { 6,{&amp;fsm[0],&amp;fsm[2]}} }; unsigned char Pos; // between 0 and 199 StatePtr Pt;      // Current State</pre>
--	---

### Program 8.16

A double circular linked list used to control the stepper motor.

The programs that step the motor also maintain the position in the global, Pos. If the motor slips, then the software variable will be in error. Also it is assumed the motor is initially in position 0 at the time of the ritual (Program 8.17).

<pre>;Move 1.8 degrees clockwise CW: ldx Pt      ;current state     ldx 1,x    ;Next clockwise     stx Pt      ;Update pointer     ldaa ,x    ;Output pattern     staa PORTB ;Set phase control     ldaa Pos    ;Update position     inca        ;clockwise     cmpa #200     blo OK1    ;0&lt;= Pos &lt;=199     clra OK1: staa Pos     rts ;Move 1.8 degrees counterclockwise CCW: ldx Pt      ;current state     ldx 3,x    ;Next CCW     stx Pt      ;Update pointer     ldaa ,x    ;Output pattern     staa PORTB ;Set phase control     ldaa POS    ;Update position     deca        ;CCW direction</pre>	<pre>// Move 1.8 degrees clockwise void CW(void){     Pt = Pt-&gt;Next[clockwise]; // circular     PORTB = Pt-&gt;Out;          // step motor     if(Pos==199){              // shaft angle         Pos = 0;               // reset     }     else{         Pos++;                // CW     } }  // Move 1.8 degrees counterclockwise; void CCW(void){     Pt = Pt-&gt;Next[counterclockwise];     PORTB = Pt-&gt;Out;          // step motor     if(Pos==0){                // shaft angle         Pos = 199;              // reset     }     else{</pre>
---	--

*continued on p. 446*

*continued from p. 445*

```

        cmpa #255
        bne OK2    ;0<= Pos <=199
        ldaa #199
OK2: staa Pos
        rts
Init: clr Pos
        ldx #S10
        stx Pt
        movb #$FF,DDRB ;6812 only
        rts
    }

        Pos--;                      // CCW
    }

// Initialize Stepper interface
void Init(void){
    Pos = 0;
    Pt = &fsm[0];
    DDRB = 0xFF; // 6812 only
}

```

### Program 8.17

Helper functions used to control the stepper motor.

The function in Program 8.18 will step the motor moving to the desired position. It will choose to go clockwise or counterclockwise, depending on which way is closer.

```

;Reg B=desired 0<=RegB<=199
desired equ 0    ;desired state
Seek  pshb      ;Save as local
      tsy
      subb Pos    ;Go CW or CCW?
      beq done    ;Skip if equal
      bhi high    ;Desired>Pos?
;Desired<Pos
      negb      ;(POS-Desired)
      cmpb #100
      blo goCCW ;Go CCW if
;Desired<Pos and Pos-Desired<100
goCW  bsr CW    ;RegA current
      cmpa desired,Y
      bne goCW   ;Pos=Desired?
      bra done
high  cmpb #100  ;(Desired-Pos)
      blo goCW   ;Go CW if
;Desired>Pos and Desired-Pos<100
goCCW bsr CCW   ;RegA current
      cmpa desired,Y
      bne goCCW  ;Pos=Desired?
done  pulb
      rts

```

```

void Seek(unsigned char desired){
short CWsteps;
    if((CWsteps=desired-Pos)<0){
        CWsteps+=200;
    } // CW steps is 0 to 199
    if(CWsteps>100){
        while(desired!=Pos){
            CCW();
        }
    }
    else{
        while(desired!=Pos){
            CW();
        }
    }
}

```

### Program 8.18

High-level function to control the stepper motor.

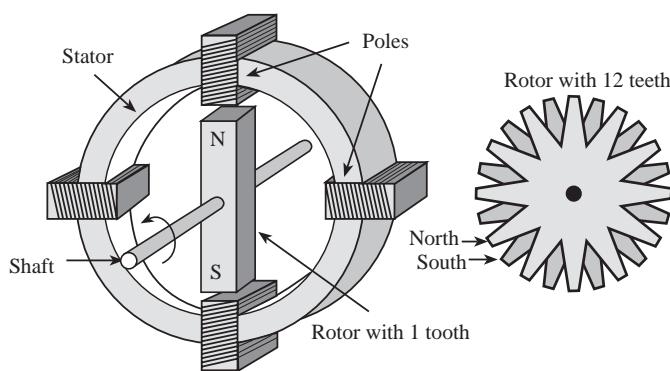
In the routine of Program 8.18 the computer will step the motor to the desired New position. We can use the current position, Pos, to determine if it would be faster to go clockwise or counterclockwise. CWsteps is calculated as the number of steps from Pos to New if the motor were to spin clockwise. If it is greater than 100, then it would be faster to get there by going counterclockwise.

### 8.6.1 Basic Operation

Figure 8.79 shows a simplified stepper motor. The permanent magnet stepper has a rotor and a stator. The rotor is manufactured from a gear-shaped permanent magnet. This simple rotor has one North tooth and one South tooth. North and South teeth are equally spaced and offset from each other by half the tooth pitch, as illustrated by the rotor with 12 teeth.

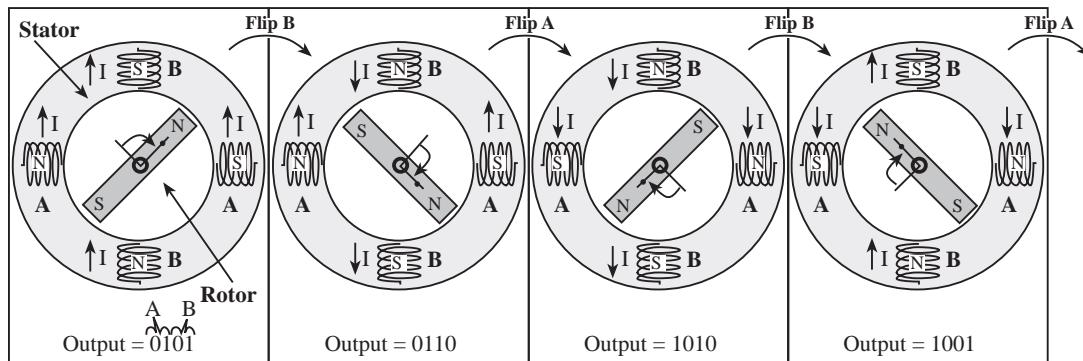
**Figure 8.79**

Simple stepper motor with four steps/revolution and a rotor with 12 teeth.



The stator consists of multiple iron-core electromagnets whose poles are also equally spaced. The stator of this simple stepper motor has four electromagnets and four poles. The stepper motors in Figure 8.76 have 50 North teeth and 50 South teeth, resulting in motors with 200 steps per revolution. The stator of a stepper motor with 200 steps per revolution has eight electromagnets each with five poles, making a total of 40 poles.

The operation of this simple stepper motor is illustrated in Figure 8.80. The four-number sequence 0101, 0110, 1010, 1001 is called *full stepping*. In general, if there are  $n$  North teeth and  $n$  South teeth, the shaft will rotate  $360^\circ/(4 \cdot n)$  per step. For this simple motor, each step causes  $90^\circ$  rotation. Assume the initial position is the one on the left with the output equal to 0101. There are strong attractive forces between North and South magnets. This is a stable state because the North tooth is equally positioned between the two South electromagnets, and the South tooth is equally positioned between the two North electromagnets. There is no net torque on the shaft, so the motor will stay fixed at this angle. In fact, if there is an attempt to rotate the shaft with an external torque, the stepper motor will oppose that rotation and try to maintain the shaft angle fixed at this position. In fact, stepper motors are rated according to their holding torque. Typical holding torques range from 10 to 300 oz·in.

**Figure 8.80**

The full-step sequence to rotate a stepper motor.

When the software changes the output to 0110, the polarity of Phase B is reversed. The rotor is in an unstable state, because the North tooth is near the North electromagnet on the top and the South tooth is near the South electromagnet on the bottom. The rotor will move because there are strong repulsive forces from the top and bottom poles. By observing the left and right poles, the closest stable state occurs if the rotor rotates clockwise, resulting in the stable state illustrated as the picture in Figure 8.80 labeled "Output = 0110". The "Output = 0110" state is exactly  $90^\circ$  clockwise from the "Output = 0101" state. Now, once again the

North tooth is near the South poles and the South tooth is near the North poles. This new position has strong attractive forces from all four poles, holding the rotor at this new position.

Next, the software outputs 1010, causing the polarity of Phase A to be reversed. This time, the rotor is in an unstable state because there are strong repulsive forces on the left and right poles. The closest stable state occurs if the rotor rotates clockwise, resulting in the stable state illustrated in the picture labeled in Figure 8.80 as “Output = 1010”. This new state is exactly 90° clockwise from the last state, moving to position the North tooth near the South poles and the South tooth near the North poles.

When the software outputs 1001, the polarity of Phase B is reversed. This causes a repulsive force on the top and bottom poles and the rotor rotates clockwise again by 90°, resulting in the stable state shown in the picture in Figure 8.80 labeled “Output = 1001”. After each change in software, there are two poles that repel and two poles that attract, causing the shaft to rotate. The rotor moves until it reaches a new stable state with the North tooth close to South poles and the South tooth close to North poles. When the software outputs a 0101, it will rotate 90°, resulting in a position similar to the original “Output = 0101” state. If the software outputs a new value from the 5, 6, 10, 9 sequence every 250 ms, the motor will spin clockwise at 1 rps. The rotor will spin in a counterclockwise direction if the sequence is reversed.

There is an eight-number sequence called *half stepping*. In full stepping, the direction of current in one of the coils is reversed in each step. In half stepping, the coil goes through a no-current state between reversals. The half-stepping sequence is 0101, 0100, 0110, 0010, 1010, 1000, 1001, and 0001. If a coil is driven with the 00 command, it is not energized, and the electromagnet applies no force to the rotor. A motor that requires 200 full steps to rotate once will require 400 half-steps to rotate once. In other words, the half-step angle is  $\frac{1}{4}$  of a full-step angle.

In a four-wire (or bipolar) stepper motor, the electromagnets are wired together, creating two phases. The five- and six-wire (or unipolar) stepper motors also have two phases, but each is center-tapped to simplify the drive circuitry. In a bipolar stepper, all copper in the windings carries current at all times, whereas in a unipolar stepper, only half the copper in the windings is used at any one time.

The time between states determines the rotational speed of the motor. Let  $\Delta T$  be the time between steps and  $\theta$  the step angle; then the rotational velocity  $v$  is  $\theta / \Delta T$ . As long as the load on the shaft is below the holding torque of the motor, the position and speed can be reliably maintained with an open-loop software control algorithm. To prevent skips (digital commands that produce no rotor motion) it is important to limit the change in acceleration, or *jerk*. Let  $\Delta T(n - 2)$ ,  $\Delta T(n - 1)$ ,  $\Delta T(n)$  be the discrete sequence of times between steps.

$$v(n) = \theta / \Delta T$$

The acceleration is given by

$$a(n) = (v(n) - v(n - 1)) / \Delta T(n) = (\theta / \Delta T(n) - \theta / \Delta T(n - 1)) / \Delta T(n) = \theta / \Delta T(n)^2 - \theta / (\Delta T(n - 1) \Delta T(n))$$

The change in acceleration, or jerk, is given by

$$b(n) = (a(n) - a(n - 1)) / \Delta T(n)$$

For example, if the time between steps is to be increased from 1000 to 2000  $\mu$ s, an ineffective approach (as shown in Table 8.15) would simply be to go directly from 1000 to 2000. This produces a very large jerk that may cause the motor to skip.

Table 8.16 shows that a more gradual change from 1000 to 2000 produces a ten times smaller jerk, reducing the possibility of skips. The optimal solution (the one with the smallest jerk) occurs when  $v(t)$  has a quadratic shape. This will make  $a(t)$  linear, and  $b(t)$  a constant. Limiting the jerk is particularly important when starting to move a stopped motor.

**Table 8.15**

An ineffective approach to changing motor speed.

<i>n</i>	$\Delta T$ ( $\mu s$ )	$v(n)$ ( $^{\circ}/s$ )	$a(n)$ ( $^{\circ}/s^2$ )	$b(n)$ ( $^{\circ}/s^3$ )
1		1000	1800	
2	1000	1800	0.00E+00	
3	2000	900	-2.50E+05	0.00E+00
4	2000	900	0.00E+00	-2.25E+08
5	2000	900	0.00E+00	2.25E+08
6	2000	900	0.00E+00	0.00E+00

**Table 8.16**

An effective approach to changing motor speed.

<i>n</i>	$\Delta T$ ( $\mu s$ )	$v(n)$ ( $^{\circ}/s$ )	$a(n)$ ( $^{\circ}/s^2$ )	$b(n)$ ( $^{\circ}/s^3$ )
1		1000	1800	
2	1000	1800	0.00E+00	
3	1000	1800	0.00E+00	0.00E+00
4	1008	1786	-1.39E+04	-1.38E+07
5	1032	1744	-4.11E+04	-2.64E+07
6	1077	1671	-6.77E+04	-2.47E+07
7	1152	1563	-9.37E+04	-2.26E+07
8	1275	1411	-1.19E+05	-1.96E+07
9	1500	1200	-1.41E+05	-1.48E+07
10	1725	1044	-9.06E+04	2.91E+07
11	1848	974	-3.78E+04	2.86E+07
12	1923	936	-1.96E+04	9.44E+06
13	1968	915	-1.09E+04	4.43E+06
14	1992	904	-5.65E+03	2.63E+06
15	2000	900	-1.77E+03	1.94E+06
16	2000	900	0.00E+00	8.85E+05
17	2000	900	0.00E+00	0.00E+00

## 8.6.2 Stepper Motor Hardware Interfaces

The bipolar stepper motor can be controlled with two H-bridge drivers. We could design two H-bridge drivers using individual transistors, as previously shown in Figures 8.68 and 8.69. Unipolar stepper motors can be controlled with four current switches, as previously shown in Figure 8.77. During the design phase of a project it is appropriate to evaluate alternative solutions. In this section, we present stepper motor interfaces that employ integrated circuits to control the stepper motor. In addition to the devices presented in this section, Allegro, Texas Instruments, and ST Microelectronics offer a variety of stepper motor controllers (e.g., A3972, A3980, and UCN5804B). Table 8.17 lists some low cost, low torque stepper motors.

**Table 8.17**

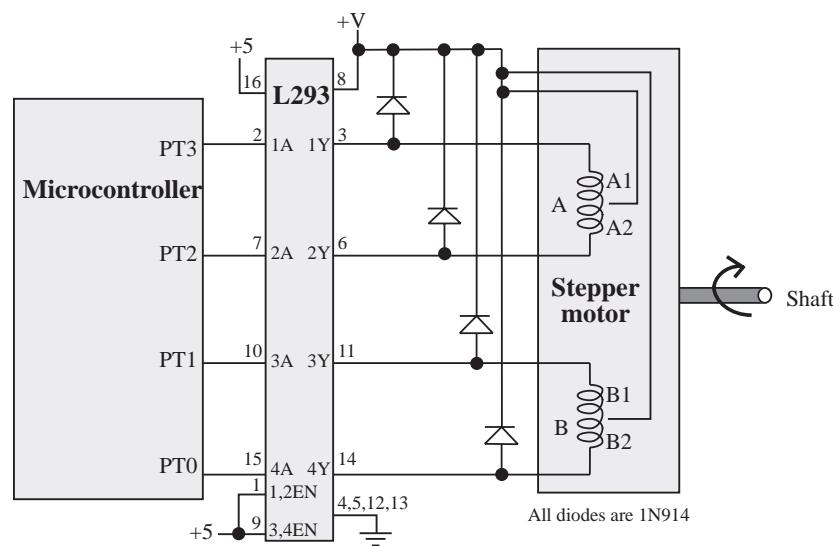
Specifications of typical stepper motors.

Manufacturer	Phase Voltage	Phase Current	Step Angle	Holding Torque	Type	No. of Leads	Used Cost
Eastern Air	2.7 V	1.9 A	1.8°	53 oz-in	unipolar	8	\$8.95
Astro-Syn	5 V	0.45 A	1.8°	10 oz-in	bipolar	4	\$4.95
Shinano Kenshi	6 V	1.2 A	1.8°	80 oz-in	unipolar	5	\$12.95
AirPax	12 V	0.33 A	7.5°	10.5 oz-in	unipolar	6	\$3.95
Oriental Vexta	12 V	0.68 A	1.8°	125 oz-in	unipolar	6	\$24.50
Sigma	24 V	0.66 A	1.8°	325 oz-in	unipolar	5	\$34.50

The L293 is a popular IC for interfacing stepper motors. It uses Darlington transistors in a double H-bridge configuration (as shown in Figure 8.81), which can handle up to 1 A per channel and voltages from 4 to 36 V. The 1N914 snubber diodes protect the electronics from the back EMF generated when currents are switched on and off. The L293D has internal snubber diodes, but can handle only 600 mA. Figure 8.81 shows four

**Figure 8.81**

## Bipolar stepper motor interface using an L293 driver.

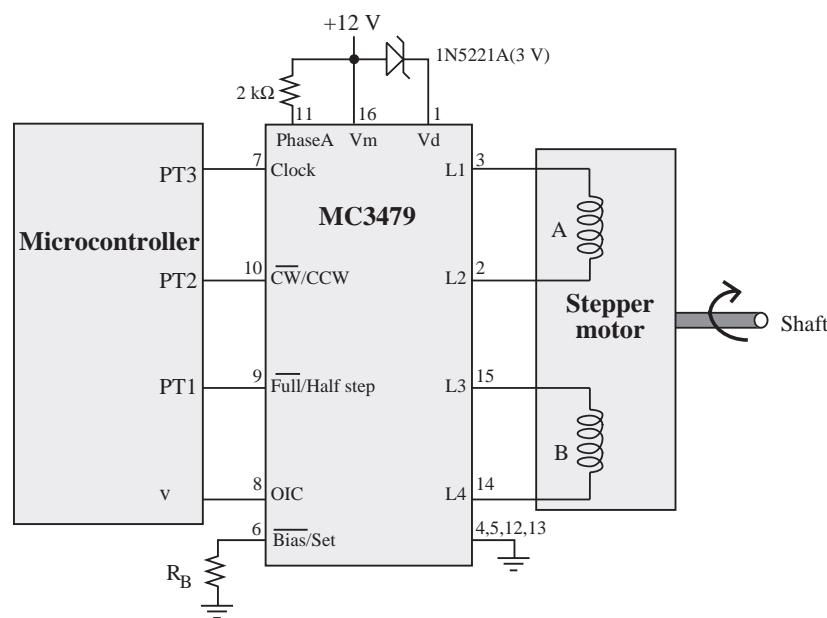


digital outputs from the microcontroller connected to the **1A**, **2A**, **3A**, **4A** inputs. The software rotates the stepper motor using either the standard full step (5-6-10-9...) or half step (5-4-6-2-10-8-9-1...) sequence. For example, Programs 8.16 and 8.17 can be used to control this motor.

There are integrated circuits, such as the MC3479 in Figure 8.82, that perform even more of the stepper motor logic in hardware. **V<sub>m</sub>** is the motor voltage, and the IC derives an internal +5 V to power its digital logic. The circuit is protected from back EMF with internal snubber diodes, and the external zener diode placed between **V<sub>m</sub>** and **V<sub>d</sub>**. The high current outputs on **L<sub>1</sub>, L<sub>2</sub>, L<sub>3</sub>, and L<sub>4</sub>** can source or sink 350 mA. The maximum sink current can be limited by the **R<sub>B</sub>** from pin 6 to ground. The stepper motor is moved on each low-to-high transition of the **Clock** input; therefore the speed of the stepper is controlled by the frequency of the **Clock**. This interface can be controlled using the squarewave generation software in Program 6.8 or 6.15. Either full or half stepping can be selected (pin 9), and the direction is controlled by the **CW/CCW** signal. The **OIC**

**Figure 8.82**

Another bipolar stepper motor interface.



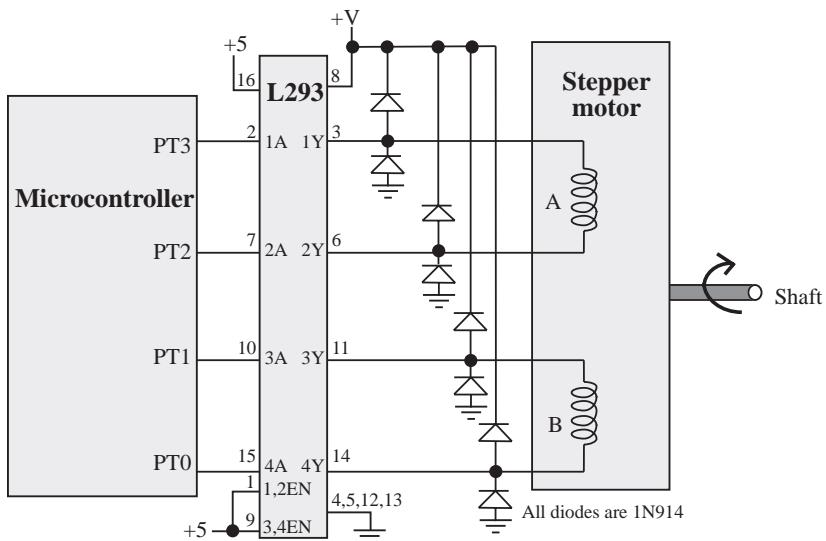
signal controls the voltage on the coils during the half-stepping off states. For example, the half-step voltages on (**L1,L2**) on a 12 V interface can be either

$$\begin{aligned} \text{OIC}=0 \text{ (12,0)} &\rightarrow (\text{hiZ},\text{hiZ}) \rightarrow (0,12) \rightarrow (\text{hiZ},\text{hiZ}) \rightarrow \dots \text{ or} \\ \text{OIC}=1 \text{ (12,0)} &\rightarrow (12,12) \rightarrow (0,12) \rightarrow (12,12) \rightarrow \dots \end{aligned}$$

The unipolar stepper architecture provides for bidirectional currents by using a center tap on each phase. The center tap is connected to the +V power source, and the four ends of the phases are controlled with open-collector drivers, as shown in Figure 8.83. Only half of the electromagnets are energized at one time. The L293 provides up to 1A current.

**Figure 8.83**

Unipolar stepper motor interface.



**Checkpoint 8.14:** What changes could you make to a stepper motor system to increase torque, increasing the probability that a step command actually rotates the shaft?

**Checkpoint 8.15:** Do you need a sensor feedback to measure the shaft position when using a stepper motor?

### 8.6.3 Stepper Motor Shaft Encoder

Stepper motors can also be used in a passive mode as shaft encoders, giving both speed and position. When the shaft of a stepper motor is rotated, a series of electric potentials is generated across its coils. The circuit shown in Figure 8.84 converts the AC voltage signals into two digital square waves. The frequency of the waves gives the rotational speed of the shaft, and the phase between the two waves gives the direction (clockwise or counterclockwise).

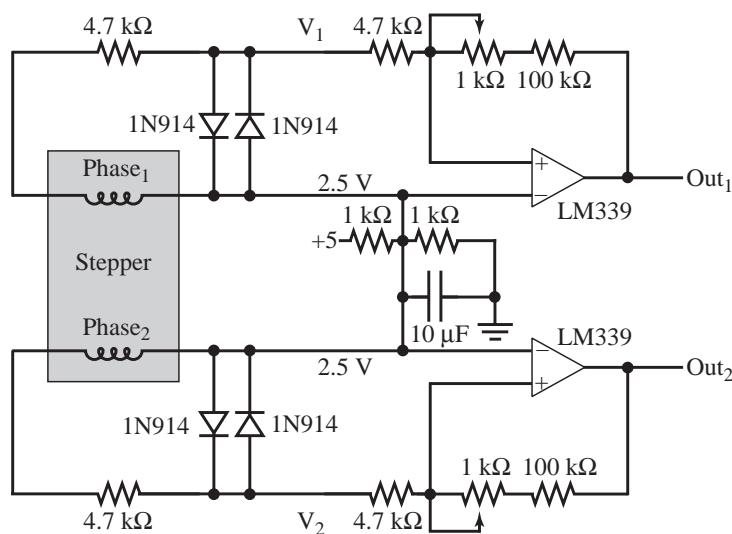
The 1N914 diodes produce signals that are  $\pm 1$  V from the 2.5 V center. Four steps of the motor will produce one large cycle and one small cycle on  $V_1$  and  $V_2$ . The distance traveled to produce these two cycles is the distance between two magnetic detents. The positive feedback hysteresis is adjusted so that the output signals ( $\text{Out}_1$ ,  $\text{Out}_2$ ) are sensitive to the large cycle but miss the small cycle. The shaft rotation speed,  $r$  in rps, is given by

$$r = \frac{4f}{N}$$

where  $f$  is the frequency of  $\text{Out}_1$  in hertz and  $N$  is the number of steps in the motor. For example, a 200-step motor will yield 50 pulses per rotation. The microcomputer can measure the frequency of  $\text{Out}_1$  to obtain shaft speed, or count pulses modulo 50 to establish position.

**Figure 8.84**

Stepper motors can also be used as shaft position sensors.

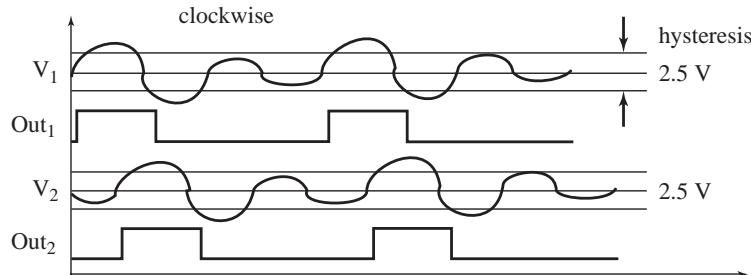


The direction of the rotation can be determined by the phase between Out<sub>1</sub> and Out<sub>2</sub>. For a clockwise rotation, the value of Out<sub>2</sub> at the time of the fall of Out<sub>1</sub> is high (Figure 8.85).

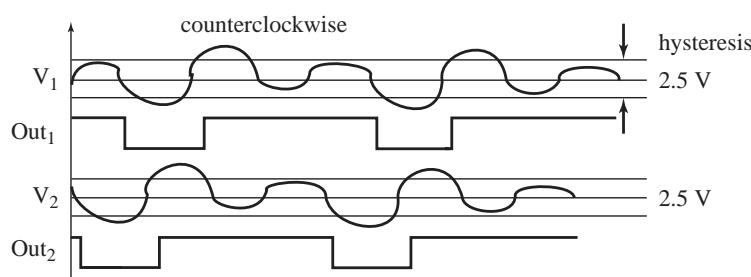
For a counterclockwise rotation, the value of Out<sub>2</sub> at the time of the fall of Out<sub>1</sub> is low (Figure 8.86).

**Figure 8.85**

Clockwise timing of a stepper motor used as shaft position sensor.

**Figure 8.86**

Counterclockwise timing of a stepper motor used as shaft position sensor.



A unipolar stepper with six wires can be used exactly like a bipolar stepper by not connecting the common wires. A unipolar stepper with five wires is connected with the 2.5 V center at the common signal; then only half of each phase is used. This configuration will yield signals with half the amplitude.

## 8.7 Servo Motors

*Servo motors* have a rotating shaft (called a horn), a geared DC motor, a shaft sensor, and a built-in controller. Servos are a popular motor to implement steering in robotics (see Figure 8.87). Ranging from micro servos with 15 oz-in torque to powerful heavy-duty

**Figure 8.87**

This servo motor has a horn that can rotate 60 degrees.



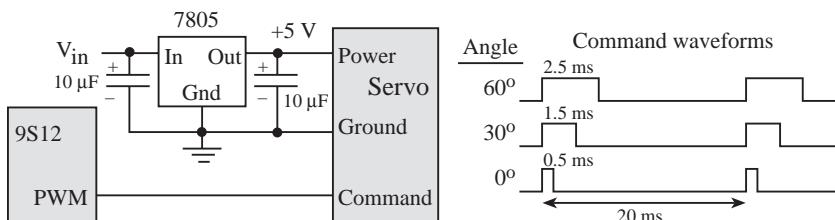
Courtesy of Jonathan Vavano.

sailboat servos, they have similar characteristics. A servo is essentially a motor for which you set the desired position or angle. The servo knows where it is (the actual angle) and where it wants to be (the desired angle). The controller built into the servo attempts to move the servo horn to the desired angle. In this section, we will focus on interfacing a servo to the microcontroller.

The servo is controlled by three wires: ground, power (+5 V), and command (PWM wave), as shown in Figure 8.88. Power is usually between 4 V and 6 V and should be separate from system power (as servos are electrically noisy). Servos are rated according to their speed and torque. Even small servos can draw over an amp under heavy load, so the power supply should be appropriately rated. Though not recommended, servos may be driven to higher voltages to improve torque and speed characteristics. Servos are commanded using a PWM signal sent through the command wire. Essentially, the width of a pulse defines the desired angle. The full range of motion for a typical servo is 60°. For example, sending a 1.5 ms pulse to the servo tells the servo that the desired angle is 30°. In order for the servo to hold this angle, the command must be sent at about 50 Hz, or every 20 ms.

**Figure 8.88**

A servo motor interfaced to a microcontroller, controlled by PWM.



If you were to send a pulse longer than 2.5 ms or shorter than 0.5 ms, the servo would attempt to overdrive and damage itself. Once the servo has received the desired angle (via the PWM signal) the servo must attempt to match the desired and actual angles. It does this by turning a small, geared motor clockwise or counterclockwise. If, for example, the desired angle is less than the actual angle, the servo will turn counterclockwise. On the other hand, if the desired angle is greater than the actual angle, the servo will turn clockwise. In this manner, the servo zeros in on the correct angle. Should a load rotate the servo horn, the servo will attempt to compensate. Note that there is no control mechanism for the speed of movement, and for most servos, the speed is specified in degrees/second. If you connect the servo to the +5 V used by the microcontroller, then the servo will cause transient power losses to the microcontroller, causing 9S12 to reset. The 7805 linear regulator shown in Figure 8.88 is used to separate servo power from microcontroller power.

## 8.8 Exercises

**8.1** For each term give a definition in 32 words or less.

- |                |                      |                        |
|----------------|----------------------|------------------------|
| a) Pull-up     | h) Periodic polling  | o) Double-throw switch |
| b) Overdamped  | i) Frame             | p) 3-pole switch       |
| c) Underdamped | j) Armature          | q) Dropout voltage     |
| d) Hysteresis  | k) Back EMF          | r) Commutator          |
| e) Bounce      | l) Rotor             | s) Jerk                |
| f) Latency     | m) Stator            | t) Snubber diode       |
| g) Overhead    | n) Break before make | u) Duty cycle          |

**8.2** Define each term.

- |             |             |
|-------------|-------------|
| a) $V_{OH}$ | e) $I_{OH}$ |
| b) $V_{OL}$ | f) $I_{OL}$ |
| c) $V_{IH}$ | g) $I_{IH}$ |
| d) $V_{IL}$ | h) $I_{IL}$ |

**8.3** In 32 words or less, describe the similarities and differences between the following pairs of terms

- a) Baud rate versus bandwidth
- b) Encoder versus decoder
- c) Direct versus scanned
- d) LED versus LCD
- e) EM relay versus solid state relay
- f) Brushed DC motor versus brushless DC motor
- g) 2-pole versus 3-pole switch
- h) Single-throw versus double-throw switch
- i) BJT versus MOSFET
- j) Unipolar versus bipolar stepper motor
- k) Front plane versus backplane of an LCD
- l) 7405 and 7406

**8.4** What happens to an LED display if the scan rate is 10 Hz?

**8.5** Assume the  $V_{OL}$  of the 7405 driver can range from 0 to 0.5 V. Using the open-collector approach, what range of resistor values would you need for a 1.5 V, 15 mA LED? Pick a resistance value in the middle; then calculate the range of currents that will occur as  $V_{OL}$  ranges from 0 to 0.5 V.

**8.6**  $V_{OH}$  can range from 4.2 to 5 V. Using the direct approach, what range of resistor values would you need for a 2.1 V, 1 mA LED? Pick a resistance value in the middle; then calculate the range of currents that will occur as  $V_{OH}$  ranges from 4.2 to 5 V.

**8.7** What happens if you remove the seven  $R_3$  resistors in Figure 8.33 and replace them with three resistors between the emitter terminals of the PN2907 and the common anode side of the LEDs? Can you choose a resistor value for this new display?

**8.8** Look up the data sheet of the 1N914. Give a brief definition for the following parameters: forward current  $I_F$ , breakdown voltage  $B_V$ , reverse voltage  $V_R$ , reverse current  $I_R$ , and reverse recovery time  $trr$ . What diode parameter is most important when choosing a snubber diode?

**8.9** A stepper motor has 24 North teeth and 24 South teeth. What angle change occurs on each step? If a full step is output every 1 ms (and assuming it doesn't slip), at what speed does the motor spin?

**8.10** Draw a figure similar to Figure 8.80 showing how half stepping works.

**D8.11** Redesign Example 8.5, assuming a 3 by 3 matrix keyboard with the numbers 0 to 8.

**D8.12** Redesign Figure 8.33, assuming a common cathode 7-segment LED (using 2N2907 and 2N2222 transistors).

**D8.13** Design an interface for a +24 V, 500 mA geared DC motor. The time constant of the motor is 10 ms. Include software to adjust the delivered power from 0 to 100%.

**D8.14** Design an interface for a +12 V, 500 mA bipolar stepper motor. There are 200 steps/revolution. Write software to spin the motor at 1 rps.

**D8.15** An alarm is powered by 120 VAC, and the on/off state of the alarm is controlled by an EM relay. Design the interface between a computer output port and the EM relay. The dropout voltage is 3.5 V, and the coil current needs at least 100 mA. Limit the coil voltage to 6 V. Write three software functions: **Alarm\_Init**, **Alarm\_On**, and **Alarm\_Off**.

**D8.16** Design a security code system. Assume there are four keys that do have 10 ms of bounce. Assume there is a shared global structure stored in EEPROM that contains the valid secret code

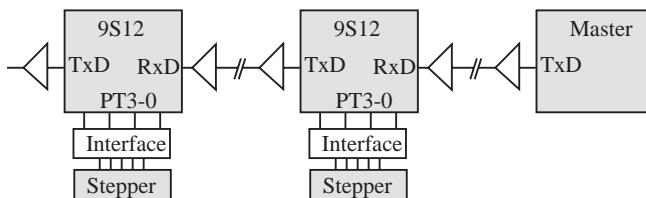
```
unsigned char const secret[3];
```

Don't worry about how this structure is initialized.

- Show interface between the four keys and the microcomputer. Any interrupting mechanism can be used (input capture, key wakeup, output compare, RTI, etc.)
- Show the initialization software. Assume there is a main program that you do not write that will call this initialization; then perform other unrelated tasks.
- Show the interrupt handler(s) that performs the security functions in the background. Call the function **Alarm\_Off** if the proper 3-code sequence is typed. You will not write this **Alarm\_Off** function.

**D8.17** The objective of this problem is to create a synchronized stepper motor network using multiple 9S12s, as shown in Figure 8.89. Each 9S12 is battery operated and is separated by 100 meters. Each system has a serial input, a serial output, and a 4-bit parallel output. The TxD output of one 9S12 is connected to the RxD input of the next 9S12. You will implement 8-bit, no parity, 1-stop, 2400-bps baud serial communication. The first 9S12 in the chain is the master, which you will not write. However, the master will send 5, 6, 10, 9,... stepper motor commands to the first slave in the chain. Upon receipt of a serial input, the 8-bit data are written to the stepper motor interfaced on Port T. After a 10-ms delay, the 8-bit data are then transmitted in serial fashion to the next computer along the chain. At 2400 baud, the smallest interval between receive frames is about 4 ms (10 bits/2400 bit/sec.) This means it is possible for a second (and a third) input frame to arrive while one is waiting the 10 ms required before transmitting the first data. You will not design the master controller. Your software must be interrupt driven. You will specify all the software contained in each slave system. The +5 V stepper motor has a coil resistance of 500  $\Omega$ .

**Figure 8.89**  
Stepper motor distributed network.



- Show the hardware interface for one 9S12 slave system. Please label chip numbers but not pin numbers. Choose the appropriate interface mechanism and be careful how the systems are grounded to each other. Clearly label all wires in the two 100 meter cables (from the previous and to the next.) The Stepper motor is unipolar, has a +5 V power, and the standard A A' B B' control.

- Show all software for one slave. The system should be interrupt driven. After initialization, the main program performs a do-nothing loop. Other than this loop in the main program, there should be no other backward jumps in the software. In the following table, notice that the parallel output occurs right away but the serial output is delayed by 10 ms. Also notice that the serial input may not (but could) occur at the full 2400 bits/sec.

Time	Serial Input	Parallel Output	Serial Output
0	6	6	none
4	9	9	none
9	11	11	none
10	none	none	6
13	2	2	none
14	none	none	9
19	none	none	11
23	none	none	2

**D8.18** Design the hardware interface that allows the computer to control an electromagnetic solenoid. The computer controls the solenoid by driving current (activating the solenoid) or no current (deactivating it) through the coil, which has a resistance of  $200\ \Omega$ . The solenoid activates when the coil voltage is above 4 V (i.e., a coil current above 20 mA). Limit the voltage across the coil to less than 6 V. There will be back EMF voltages, so protect the electronics.

**D8.19** Interface a double-throw switch to the microcontroller. Use digital logic to debounce the switch in hardware using no capacitors.

## 8.9 Lab Assignments

**Lab 8.1** The overall objective of this lab is to design a four-function fixed-point calculator. The first task is to interface a 4 by 4 matrix keyboard using interrupt synchronization. You must debounce the keyboard and handle two-key rollover. Rollover is defined as touching the next key before releasing the last key. For example, when some people type “1,2,3,” they push “1,” push “2,” release “1,” push “3,” release “2,” and then release “3.” Design a low-level device driver for the keyboard interface. Second, you need to interface an LCD display, such as one controlled by the HD44780. The third task is to design a four-function 16-bit signed fixed-point calculator. All numbers will be stored in signed 16-bit fixed-point format with a constant of 0.001. The full-scale range is from  $-32.767$  to  $+32.767$ . You should include routines to facilitate fixed-point input from the keyboard and output to the LCD. The matrix keyboard will include the numbers •0’—•9’, and the letters •+’, •-’, •\*’, •/’, •=’, and •.’. If you want to extend the number of keys, you can define one of the 16 keys as the •shift’ key, creating 30 possibilities. The HD44780 LCD display will show both a 16-bit global accumulator, and a 16-bit temporary register. You are free to design the calculator functionality in any way you wish, but you must be able to (1) clear the accumulator and temporary; (2) type numbers in using the matrix keyboard; (3) add, subtract, multiply, and divide; (4) display the results on the HD44780 LCD display. No SCI input/output is allowed in the calculator program.

**Lab 8.2** In this lab, you will control a stepper motor using a finite state machine. The finite state machine must be implemented as a linked data structure. Two switches will allow the operator to control the motor. A background periodic interrupt (either OC or RTI) thread will perform inputs from the switches and outputs to the stepper motor coils. You will use a motor that has fixed-number full-steps to move the shaft one rotation. Two switches determine the motor operation. If both buttons are released, the motor should stop. If switch 1 is pressed, the motor should spin slowly clockwise. If switch 2 is pressed, the motor should spin quickly counterclockwise. The motor should stop when it reaches either the beginning or the end of one rotation even if the operator continues to press the switch. If both switches are pressed, the motor should continuously spin as fast as possible back and forth across the full range of one rotation. If the system is performing one operation and another command is issued, the first operation is terminated and the second command is performed. With an ohmmeter, measure the resistance of one coil. Apply the power across the coil and simultaneously measure, using both a voltmeter and a current-meter, the voltage and current required to activate one coil of your stepper motor. Be sure the interface circuit you select can supply enough current to activate the coil. Measure the inductance of one coil, and determine the turnoff time of your interface. Use these two measurements to estimate the back EMF.

**Lab 8.3** The objective of this lab is to design and test a digital alarm clock. You can connect individual push-button switches to input pins of the microcontroller, which the user can use to set the current time and the alarm time. The LCD display will be used to display the current time. You are free to implement whatever features you wish, but there must be a way to set the time. The system maintains three global variables—`hour`, `minute`, `second`. No SCI input is allowed, and the time parameters must be maintained using interrupts. In particular, the interrupt service routine should increment `second` once a second, increment `minute` once a minute, and increment `hour` once an hour. The foreground (main) will output to the LCD display, and interact with the operator via the switch inputs. If the correct sequence of switches is pushed, the main program can initialize the values of `hour`, `minute`, `second`. Design an interface between a speaker and an output port, such that when you toggle the output, an alarm sound is made. The interrupt service routine maintains time, and the main program outputs to the LCD. You can input from the switches in either the foreground or the background. You must be careful not to let the LCD show an intermediate time of 1:00:00 as the time rolls over from 1:59:59 to 2:00:00. You must also be careful not to disable interrupts too long (more than one interrupt period), because a time error will result if any interrupts are skipped. If you use RTI interrupts, you will have to do some 32-bit math to maintain the exact time. For example, if the RTI interrupts at 30.517Hz, then interrupts are requested at exactly 32768  $\mu$ s. One method is to add 32768 to a 32-bit counter. When the counter exceeds 1 million, increment `second` and subtract 1 million from the counter.

**Lab 8.4** The objective of this lab is to interface a brushless DC motor. Rather than use discrete transistors like Figure 8.72, use an integrated motor interface chip like the L293 or L6103. Using three power resistors in a Wye shape, test the motor drive circuit. Write two software functions: Initialization and PowerSet. The second function can be used to adjust the power to the motor from 0 to 100%. Add period measurements to the input capture ISRs, and use them to measure shaft speed in Rpm. Use a logic analyzer similar to Figure 8.62 to collect data. Take measurements of shaft speed versus duty cycle. Take transient measurements of speed versus time, as the duty cycle changes from 25 to 75%.

# 9 Memory Interfacing

## Chapter 9 objectives are to:

- ❖ Present the basic engineering design steps for interfacing memory to the computer
- ❖ Review the basic building blocks of computer architecture
- ❖ Compare and contrast synchronous, partially asynchronous, and fully asynchronous buses
- ❖ Discuss address translation as it occurs in a paged-memory system
- ❖ Develop software to program internal flash EEPROM.

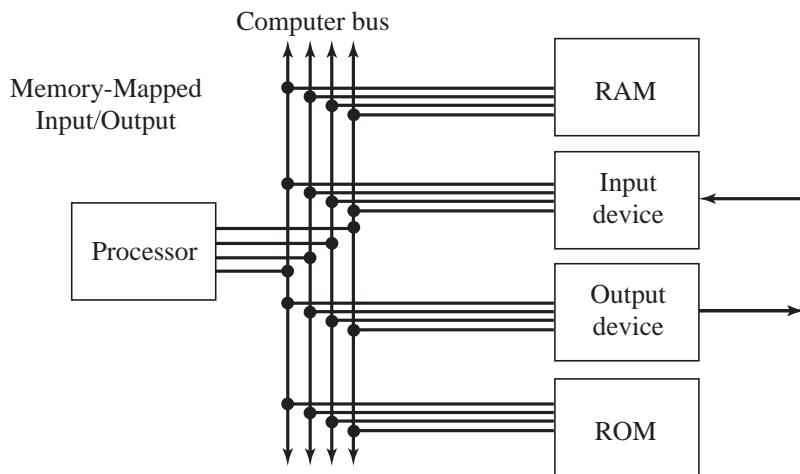
For most applications, the embedded system uses a single-chip microcomputer with built-in memory devices. Therefore there is no need to attach additional memory. In other words, we select an available microcontroller with adequate RAM and ROM space to satisfy our constraints. Consequently, most embedded system designers can skip this chapter. On the other hand, there are a few situations that might lead one to need to attach external devices to the address/data bus of the microcontroller. Memory interfacing and bus timing are general topics of computer architecture, and you may wish to study this chapter, not because you need to know how to attach memory for the embedded system but because you plan to apply this knowledge toward the design of general-purpose computers. In a similar fashion, understanding how to attach interface memory is an important step in the design of the single-chip microcontroller itself. The other situation that requires memory interfacing occurs when you need more memory than is available on the single-chip microcomputer. In this situation you can attach external memory to satisfy the constraint of the problem. For example, if the embedded system needed 1 mebibyte of RAM, then the best solution would be to interface an external memory. Some microcontrollers have large flash EEPROMs, and it will be convenient to use some of this space to log data at run time.

## 9.1 Introduction

In this chapter, we will approach memory interfacing using both general concepts and specific examples. The processor is connected to the memory and I/O devices via the bus. The bus contains timing, command, address, and data signals. The timing signals, like the 9S12 E clock, determine when strategic events will occur during the transfer. The command signals, like the 9S12 R/W line, specify the bus cycle type. During a *read cycle*, data flow from the memory or input device to the processor, where the address bus specifies the memory or input device location and the data bus contains the information. During a *write cycle*, data are sent from the processor to the memory or output device, where the address bus specifies the memory or output device location and the data bus contains the information. During each particular cycle on most computers there is exactly one device sending the data and exactly one device receiving the data. Some buses like the IEEE488

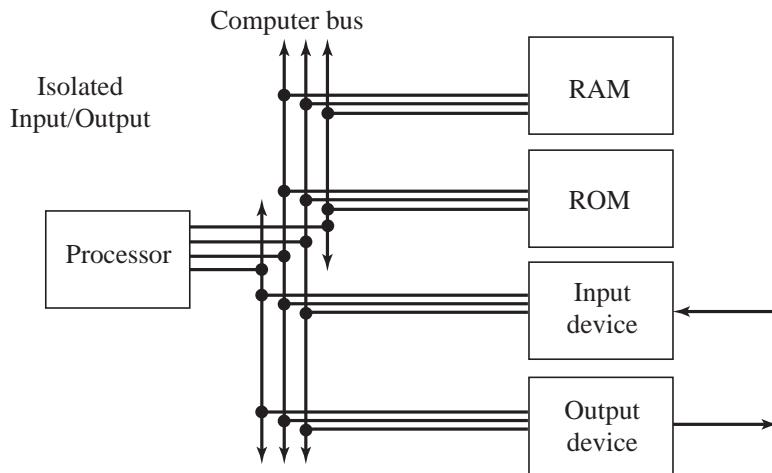
and SCSI allow for broadcasting, where data is sent from one source to multiple destinations. In a computer system with memory-mapped I/O, all slaves share the same bus (Figure 9.1).

**Figure 9.1**  
Architecture of a computer with memory-mapped I/O.



In an Intel x86 computer system with isolated I/O, the slaves share the address and data buses but have separate control signals (Figure 9.2). From a programming perspective, processors with isolated I/O access their I/O devices and memory with separate instructions. On the Intel x86, there are four basic bus cycles: memory read, memory write, I/O read, and I/O write. The memory read/write cycles are similar to the memory-mapped cycles described above. During an *I/O read cycle*, data flow from the input device to the processor, where the address bus specifies the input device location and the data bus contains the information. During an *I/O write cycle*, data are sent from the processor to the output device, where the address bus specifies the output device location and the data bus contains the information.

**Figure 9.2**  
Architecture of a computer with isolated I/O.

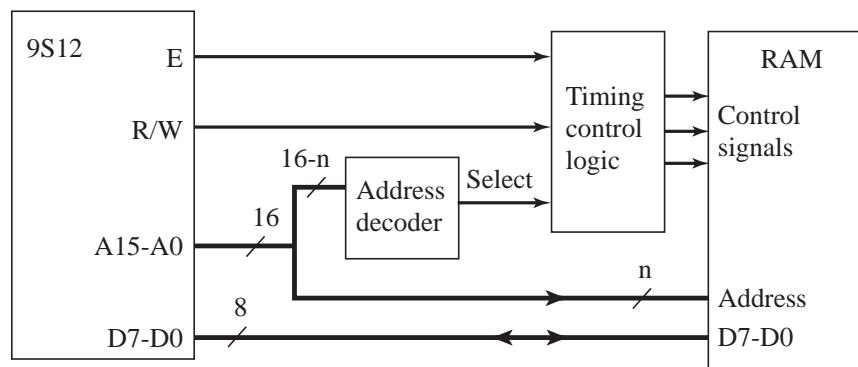


The interface to the slave devices, such as the RAM, ROM, and I/O devices, will consist of two parts. For each cycle, one packet of data is transmitted on the bus. We use the *command* part of the design to specify which slave will be active. The command portion of

our slave design will specify whether or not the current cycle is meant for our slave. If the slave can participate in both the read and write cycles, then the command part will also distinguish between the read and write functions. In a system without DMA, the processor will always be the bus master. The bus master will determine the address and bus cycle type for each cycle. The 16 address lines and the R/W signal of the 9S12 specify the type of the current cycle (Figure 9.3). *There is no timing information in the address and R/W lines.* On computer systems that contain coprocessors or DMA controllers, there can be more than one bus master. For any given cycle there is only one bus master that will specify the slave address and bus cycle type. The processor, coprocessors, and DMA controllers vie for the use of the shared memory bus. More about DMA can be found in Chapter 10.

**Figure 9.3**

The 9S12 supports external memory interfaces (expanded mode).



The first step in interfacing a device to the microcomputer is to construct an address decoder. Let SELECT be the positive logic output of the address decoder circuit. The command portion of our interface can be summarized by Table 9.1.

**Table 9.1**

The address decoder and R/W determine the type of bus activity for each cycle.

SELECT	R/W	Function	Rationale
0	0	Off	Because the address is incorrect
0	1	Off	Because the address is incorrect
1	0	Write	Data flow from 9S12 to our device
1	1	Read	Data flow from our device to the 9S12

The second part of the interface will be the *timing*. The 9S12 uses a *synchronous* bus; thus all timing signals will be synchronized to the E clock. It is important to differentiate timing signals from command signals within the interface. A timing signal is one that has rising and falling edges at guaranteed times during the memory access cycle. In contrast, command signals are generally valid during most of the cycle but do not have guaranteed times for their rising and falling edges. Table 9.2 divides the bus signals into the command or timing category. When interfacing memory to a 9S12 without paging, it can access up to 64 kibibytes (KiB) using a 16-bit address. When interfacing memory to a 9S12 with paging, it can access up to 1 mebibyte (MiB) using a 20-bit address.

**Table 9.2**

Available timing signals and command signals from the 9S12.

Configuration	Timing Signals	Command Signals
64 kibibyte	E	R/W AD15–AD0 LSTRB
1 mebibyte	E	R/W XCS X19–X14 AD15–AD0 LSTRB

Timing equations were presented in Section 3.1.4, and timing diagrams were presented in Section 3.1.5. Bus timing will be presented. Various RAM memories will be interfaced to the MC9S12C32, but a similar approach can be used to interface memory to other 9S12s.

**Common error:** If control signals in our interface are derived only from command (type and address) signals, then we will have no guarantee when the rising and falling edges will occur.

**Observation:** We use the command signals to specify which cycles activate our slave, and we use the timing (clock) signals to specify when during the cycle to activate.

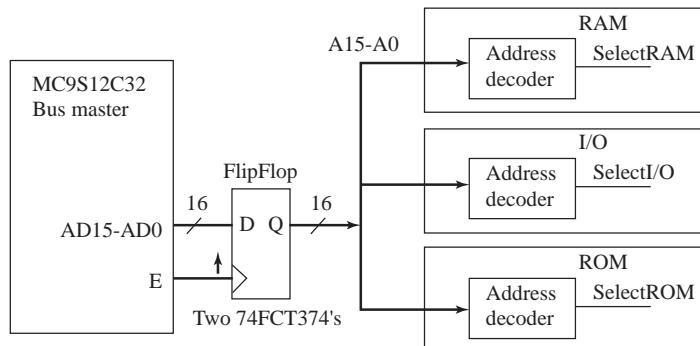
## 9.2 Address Decoding

The address decoder is usually located in each slave interface. The purpose of the address decoder is to monitor the 16 address lines from the bus master (the processor) and determine whether or not the slave has been selected to communicate in the current cycle. Each slave has its own address decoder, uniquely designed to select the addresses intended for that device. Care must be taken to avoid having two devices driving the data bus at the same time. We wish to select exactly one slave device during every cycle. The MC9S12C32 uses a multiplexed address/data bus.

In the MC9S12C32, the same 16 wires, AD15–AD0, are used for the address (A15–A0) in the first half of each cycle and for the data (D15–D0) in the second half of each cycle. The rising edge of the E clock is used to capture the address into an external latch (two 74FCT374s), as shown in Figure 9.4. In this way, the entire 16-bit address is available during the second half of the cycle.

**Figure 9.4**

The MC9S12C32 multiplexes the 16-bit address on the same pins as the 16-bit data.



The address decoders should not select two devices simultaneously. In other words, no two select lines should be active at the same time. In this case,

$$\text{SelectRAM} \cdot \text{SelectI/O} = 0$$

$$\text{SelectRAM} \cdot \text{SelectROM} = 0$$

$$\text{SelectI/O} \cdot \text{SelectROM} = 0$$

**Common error:** If two devices have address decoders with overlapping addresses, then a write cycle will store data at both devices, and during a read cycle the data from the two devices will collide, possibly causing damage to one or both devices.

### 9.2.1 Full-Address Decoding

*Full-address decoding* is where the slave is selected if and only if the slave's address appears on the bus. In positive logic,

$$\begin{aligned} \text{Select} &= 1 && \text{if the slave address appears on the address bus} \\ &= 0 && \text{if the slave address does not appear on the address bus} \end{aligned}$$

**Example 9.1** Design a fully decoded positive logic select signal for a 1 KiB RAM at \$4000–\$43FF.

### Solution

**Step 1.** Write specified address using **0,1,X** and the following rules:

- There are 16 symbols, one for each of the address bits A15, A14 . . . , A0.
- 0** means the address bit must be 0 for this device.
- 1** means the address bit must be 1 for this device.
- X** means the address bit can be 0 or 1 for this device.
- All the **Xs** (if any) are located on the right-hand side of the expression.
- Let **n** be the number of **Xs**. The size of the memory in bytes is  $2^n$ .
- Let **I** be the unsigned binary integer formed from the  $16-n$  **0s** and **1s**.
- We get the beginning address if all the **Xs** are set to 0.
- We get the ending address if all the **Xs** are set to 1.

$$I = \frac{\text{beginning address}}{\text{memory size}}$$

In this example:

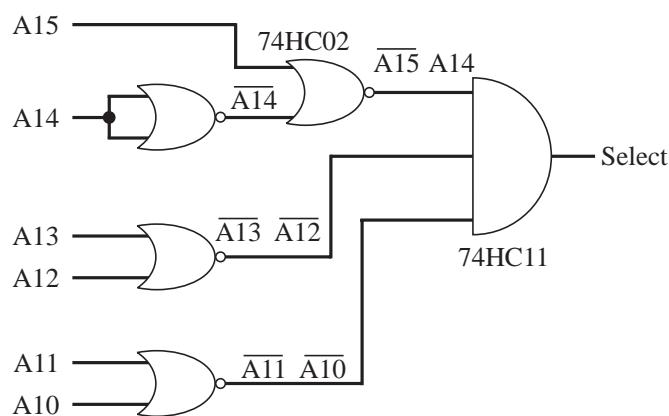
$$\begin{aligned} \text{address} &= \mathbf{0100,00XX,XXXX,XXXX} \\ \text{beginning address} &= \$4000 \\ n &= 10 \quad \text{size} = 1024 \text{ bytes} = \$0400 \\ I &= 010000_2 = 16_{10} = \frac{\$4000}{\$0400} \end{aligned}$$

**Step 2.** Write the equation using all **0s** and **1s**. A **0** translates into the complement of the address bit, and a **1** translates directly into the address bit. For example,

$$\text{Select} = \overline{A15} \cdot A14 \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10}$$

**Step 3.** Build circuit using real TTL gates (Figure 9.5).

**Figure 9.5**  
An address decoder identifies which cycles to activate.



**Common error:** If all the **Xs** are not in a group on the right-hand side, then the address range will become discontinuous.

**Example 9.2** Design a fully decoded select signal for an I/O device at \$5500 in negative logic.

### Solution

**Step 1.** Address = **0101,0101,0000,0000**

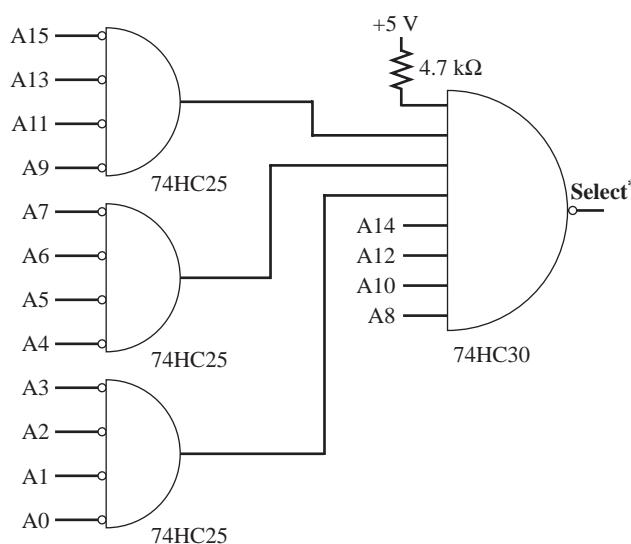
**Step 2.** The negative logic output can be created simply by inverting the output of a positive logic design. Recall that the address bus of the 6811/6812 is always in positive logic.

$$\text{Select}^* = \overline{A_{15}} \cdot A_{14} \cdot \overline{A_{13}} \cdot A_{12} \cdot \overline{A_{11}} \cdot A_{10} \cdot \overline{A_9} \cdot A_8 \cdot \overline{A_7} \cdot \overline{A_6} \cdot \overline{A_5} \cdot \overline{A_4} \cdot \overline{A_3} \cdot \overline{A_2} \cdot \overline{A_1} \cdot \overline{A_0}$$

**Step 3.** Build the circuit, as shown in Figure 9.6.

**Figure 9.6**

An address decoder like this could be implemented with programmable logic (PAL).



**Maintenance tip:** Use full-address decoding on systems where future expansion is likely.

**Observation:** We will use symbols ending in \* to signify negative logic. We will also use a line over the symbol to signify negative logic.

**Checkpoint 9.1:** Write the 0,1,X pattern for the address range \$2000 to \$3FFF.

**Checkpoint 9.2:** What address range corresponds to 1101,1XXX,XXXX,XXXX?

### 9.2.2 Minimal-Cost Address Decoding

*Minimal-cost address decoding* is a mechanism to simplify the decoding logic by introducing don't care states for unspecified addresses. It does not guarantee lowest cost but often produces simpler solutions than fully decoded. Most embedded microcomputer systems do not use the entire 65,536 available addresses. An unspecified address is one at which there is no device. A reduction of logic may be realized if we introduce *don't care* outputs in the Karnaugh maps when the address equals one of the unspecified addresses. This shortcut should work because the software should never access an unspecified address. If a valid address is accessed, then one and only one device will be selected as expected. But, the address decoders may select zero, one, or more devices if an unspecified address were to be accessed. On most computers, the address bus is in positive logic, but we may construct the **Select** signals in either positive or negative logic, depending on which is more convenient. The formal definition of minimal cost is shown in Table 9.3.

**Table 9.3**

Minimal-cost decoding optimizes by taking advantage of the don't care states.

Address	Select
Matches our device	True
Matches other specified device	False
Unspecified address	Don't care

**Example 9.3** Design four minimal-cost select signals in positive logic. The 4 KiB RAM is at \$0000 to \$0FFF, the input device is at \$5000, the output device is at \$5001, and the 16 KiB ROM is at \$C000 to \$FFFF.

### Solution

**Step 1.** Write out the addresses in binary for all devices. Include all specified addresses in the computer, not just the devices that we are designing.

RAM	0000,xxxx,xxxx,xxxx
Input	0101,0000,0000,0000
Output	0101,0000,0000,0001
ROM	11xx,xxxx,xxxx,xxxx

**Step 2.** Choose as many address lines that are required to differentiate between the devices. Consider the different devices in a pairwise fashion. If the decoder for only one device is being built, then only the addresses that differentiate that device from the others are required. In this example we choose A15, A14, A0. The address bits A15, A12, and A0 also could have been used to differentiate the devices.

**Step 3.** Draw a Karnaugh map for each device.

- a. Put a true for addresses specified by that device
- b. Put a false for other devices
- c. Put an "X" for unspecified addresses

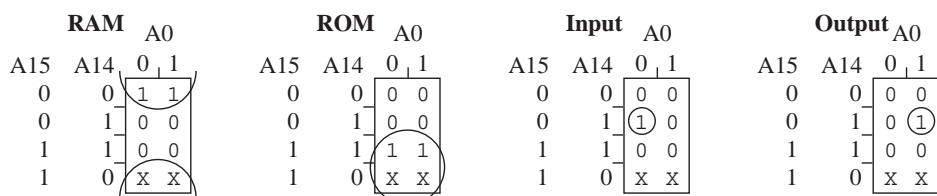
Positive logic	true = 1 and false = 0
Negative logic	true = 0 and false = 1

**Step 4.** Minimize using Karnaugh maps and determine equations (Figure 9.7).

$$\begin{array}{ll} \text{RAMSelect} = \overline{\text{A14}} & \text{ROMSelect} = \text{A15} \\ \text{InputSelect} = \overline{\text{A15}} \cdot \text{A14} \cdot \overline{\text{A0}} & \text{OutputSelect} = \overline{\text{A15}} \cdot \text{A14} \cdot \text{A0} \end{array}$$

**Figure 9.7**

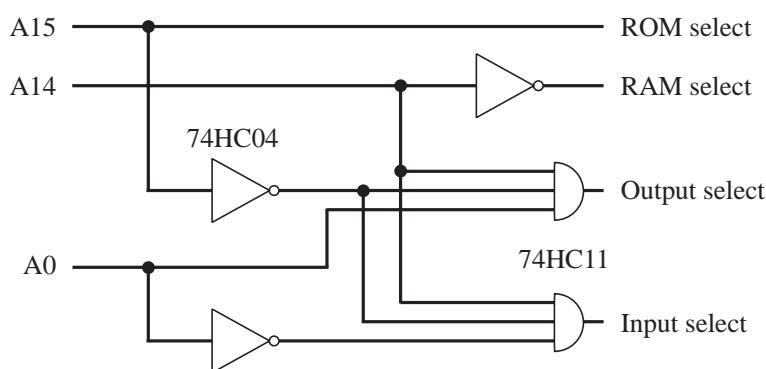
Four Karnaugh maps.



**Step 5.** Build circuit using real TTL gates (Figure 9.8).

**Figure 9.8**

Four address decoders like these could also be implemented with PAL.



To implement **negative logic** either we can put true = 0, false = 1 into the Karnaugh map or we can build the decoders in positive logic and invert the outputs.

**Performance tip:** Use minimal-cost decoding on systems where cost and speed are more important than future expansion.

**Observation:** If there are no unspecified addresses, then minimal-cost decoders will be the same as the full-address decoders.

**Performance tip:** Address decoders for the entire computer system can sometimes be implemented with a single demultiplexer like the 74HC138 or 74HC139.

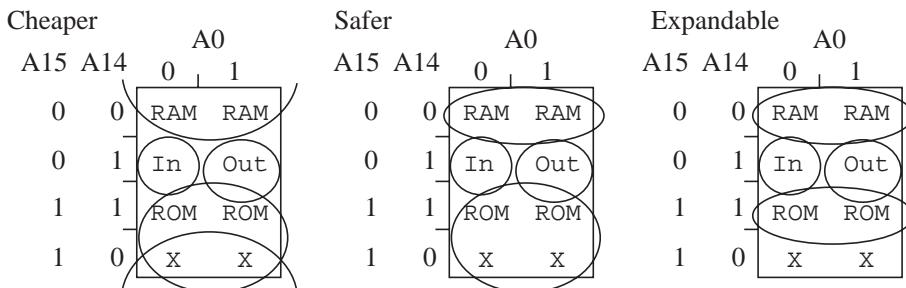
**Common error:** If one does not select enough address lines to differentiate between the devices, then some addresses may incorrectly select more than one device.

**Observation:** If one selects too many address lines, then the Karnaugh map will be harder to draw, but the resulting solution should be the same.

If all the slave select signals require the same address lines, then all the Karnaugh maps can be combined into a single map using slave names instead of 1s. We can use different colors for each slave. We then make the biggest circles that include each slave one by one. Added protection can be achieved by making none of the slave circles overlap. In this way, if the software inadvertently accesses the unspecified address, at most one slave will be activated. The “Cheaper” example in Figure 9.9 is the same solution as the one above. In this example, if the software reads location is \$8000, both the RAM and ROM would drive the data bus. If added protection is desired, we would eliminate the overlapping ROM and RAM circles. The “Safer” solution would activate only the ROM, if the unspecified address is accessed. If the ability to expand in the future is desired, we could leave holes, as shown in the “Expandable” solution.

**Figure 9.9**

A shorthand method for drawing multiple Karnaugh maps.



### 9.2.3 Special Cases When Address Decoding

If the size of the memory is not a power of 2 or if the memory size does not evenly divide the starting address, then the **0,1,X** address specification will require more than one line. To generate the multiline address specification, begin with the starting address and add **Xs** to the right side as long as the address range does not exceed the ending address. Start over at the next address until the entire range is covered. The address decoder is the **OR** of the separate lines. Do not add **Xs** to the left of a **0** or **1**, because doing so will make a discontinuous address range.

We define a *regular address range* as one with a size that is a power of 2, and one where the beginning address divided by the memory size is an integer. A regular address range can be expressed as a single line of 0s, 1s, and Xs. Two special cases with irregular address ranges are illustrated by the following examples. In the first case, the size of the memory is not a power of 2. In the second case, the beginning address divided by the memory size is not an integer. It requires more than one line of 0s, 1s, and Xs to specify the irregular address range. For irregular address ranges, we break the address range into multiple regular address ranges, solve each part separately, and then combine the parts to form the decoder for the whole.

**Example 9.4** Build a fully decoded positive-logic address decoder for a 20 KiB RAM with an address range from \$0000 to \$4FFF.

**Solution** Start with

**0000,0000,0000,0000**      Range \$0000 to \$0000

add Xs while the range is still within \$0000 to \$4FFF

<b>0000,0000,0000,000x</b>	Range \$0000 to \$0001
<b>0000,0000,0000,00xx</b>	Range \$0000 to \$0003
<b>0000,0000,0000,0xxx</b>	Range \$0000 to \$0007
<b>0000,0000,0000,xxxx</b>	Range \$0000 to \$000F

stop at

**00xx,xxxx,xxxx,xxxx**      Range \$0000 to \$3FFF

Start over

**0100,0000,0000,0000**      Range \$4000 to \$4000

add Xs while the range is still within \$4000 to \$4FFF

<b>0100,0000,0000,000x</b>	Range \$4000 to \$4001
<b>0100,0000,0000,00xx</b>	Range \$4000 to \$4003
<b>0100,0000,0000,0xxx</b>	Range \$4000 to \$4007
<b>0100,0000,0000,xxxx</b>	Range \$4000 to \$400F

stop at

**0100,xxxx,xxxx,xxxx**      Range \$4000 to \$4FFF

Combine the two parts to get:

$$\text{RAM SELECT} = \overline{\text{A15}} \cdot \overline{\text{A14}} + \overline{\text{A15}} \cdot \text{A14} \cdot \overline{\text{A13}} \cdot \overline{\text{A12}}$$

**Example 9.5** Build a fully decoded negative-logic address decoder for a 32 KiB RAM with an address range of \$2000 to \$9FFF.

**Solution** Even though the memory size is a power of 2, the size 32,768 does not evenly divide the starting address 8192. We break the \$2000 to \$9FFF irregular address range into three regular address ranges.

001x,xxxx,xxxx,xxxx	Range \$2000 to \$3FFF
01xx,xxxx,xxxx,xxxx	Range \$4000 to \$7FFF
100x,xxxx,xxxx,xxxx	Range \$8000 to \$9FFF

Combine the two parts to get:

$$\text{SELECT}^* = \overline{\text{A15}} \cdot \overline{\text{A14}} \cdot \text{A13} + \overline{\text{A15}} \cdot \text{A14} + \text{A15} \cdot \overline{\text{A14}} \cdot \overline{\text{A13}}$$

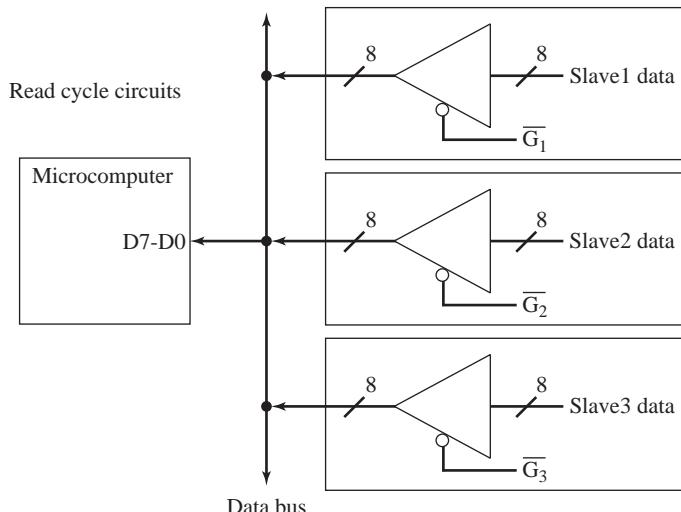
**Observation:** If the beginning address divided by the memory size is not an integer, then the address range cannot be specified by a single row of 0s, 1s, and Xs.

**Observation:** If the memory size is not a power of 2, then the address range cannot be specified by a single row of 0s, 1s, and Xs.

## 9.3 General Memory Bus Timing

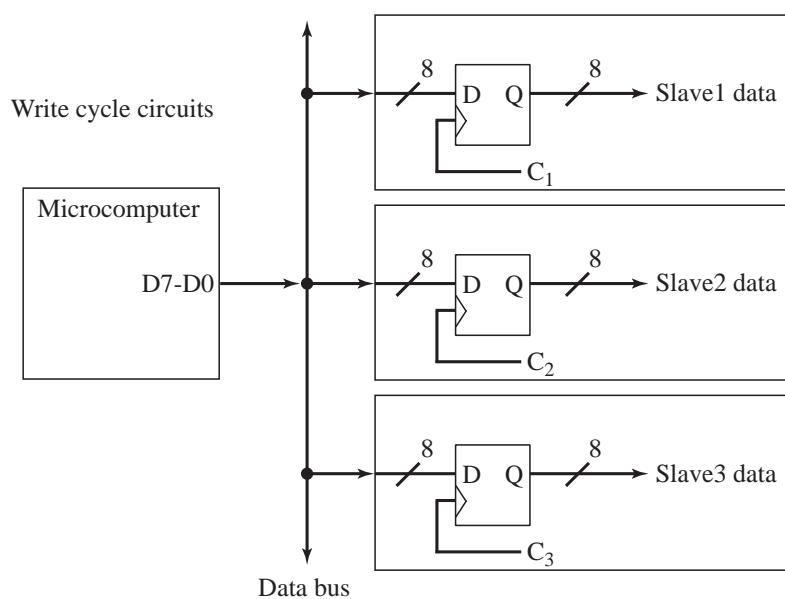
There are two types of memory access cycles. During a *read cycle*, data are passed from a slave device (memory or I/O) into the processor. During a *write cycle*, data are passed from the processor to a slave device (memory or I/O). To participate in a read cycle, the slave must have tristate logic so that it can drive data onto the data bus during read cycles involving that slave. In the first simplified diagram (Figure 9.10), we show the circuits to drive data on the bus during a read cycle for three slaves. In the second simplified diagram (Figure 9.11), we show the circuits to accept data from the bus during a write cycle for three slaves. Recall that the **command** portion of our slave design specified yes or no as to whether the current cycle is meant for our slave. In this general memory bus timing section, we will discuss three design approaches for generating the bus control signals (like  $\overline{G_1}$ ,  $\overline{G_2}$ ,  $\overline{G_3}$ , C1, C2, C3). For the TIMING portion of our slave design we will specify when during a read cycle the data are driven onto the bus (e.g.,  $\overline{G_1}$ ,  $\overline{G_2}$ ,  $\overline{G_3}$ ) and when during a write cycle data are clocked off the bus and into the slave (e.g., C1, C2, C3).

**Figure 9.10**  
Simplified diagram showing circuits used during a read cycle.



**Figure 9.11**

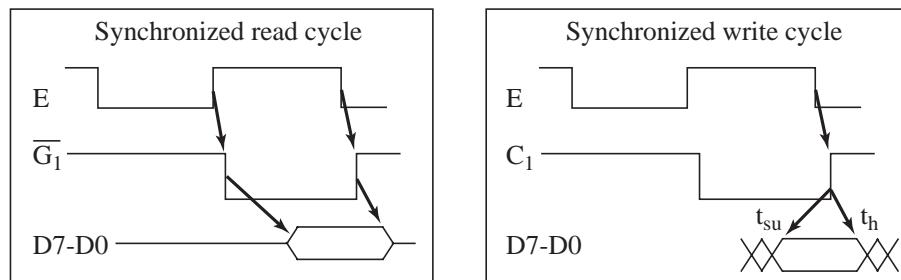
Simplified diagram showing circuits used during a write cycle.



### 9.3.1 Synchronous Bus Timing

The 9S12 implements synchronous bus transfers. In a *synchronous bus* scheme, the bus master (usually the processor) generates a clock (e.g., E clock), and all data transfer timings occur relative, or synchronous, to this clock. In our simplified circuit example above, a synchronous system has the control signals synchronized to the bus clock. In particular, the read and write cycle timing will be as shown in Figure 9.12.

**Figure 9.12**  
Synchronized bus timing.



**Observation:** In a synchronized bus interface the rising and falling edges of the control signals are synchronized (occur immediately) after an edge of the bus clock.

**Observation:** In a synchronized bus interface there is no feedback from the slave about whether or not the transfer was performed.

**Observation:** In a synchronized bus interface each bus cycle is exactly the same length, regardless of how fast the slave is.

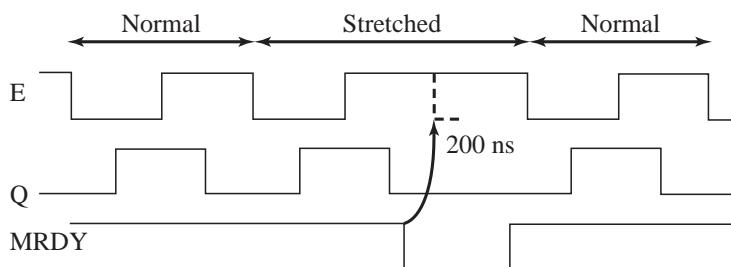
**Common error:** If the slave cannot respond fast enough or if no slave exists at the address, data are not properly read or written, but no bus error signal is generated.

**Common error:** If one slave is replaced with an equivalent but faster device in a synchronized bus interface, the computer will not execute any faster.

### 9.3.2 Partially Asynchronous Bus Timing

As mentioned above, one of the limitations of a synchronous bus is that there is no feedback from the slave to the master. The two limiting consequences of having no feedback are inefficiency (requires all devices to operate at the speed of the slowest pair) and a lack of flexibility (all devices must be considered together when designed or redesigned.) We define a *partially asynchronous bus* as one that allows slaves to affect bus timing. For example, some computers (like the Freescale 6809 and 680x0 and the Intel x86) allow an external device to slow down the system clock, thus extending the data access time. This is called *cycle stretching*. In a 6809, the E and Q clocks operate normally when MRDY is high. The 6809 E, Q clocks will stop (or stretch) for an integer multiple of quarter bus cycles whenever MRDY is low. The 6809 MRDY must fall within 200 ns of when the end of the cycle would have been for the cycle to be stretched. In Figure 9.13, the cycle is stretched by half a bus cycle. If a slave can respond in the normal time for the cycle, it will not pull MRDY low. In this way, a simple interface (one without the ability to drive MRDY low) can be designed using the regular rules of a synchronous interface.

**Figure 9.13**  
Partially asynchronous bus timing of a 6809.



Typically a slow memory or I/O device pulls MRDY low until its read or write function is complete. It is much more efficient for the system to slow down only when accessing the slow devices, rather than changing the crystal frequency (which would slow down the system for every cycle). We will use this mechanism to interface dynamic RAMs and slow I/O chips.

**Observation:** In a partially asynchronous bus, the bus cycle length can vary.

**Common error:** If the software uses a cycle-counting scheme to implement timing delays, then errors will occur if the programmer does not consider that memory accessed to slow devices will be stretched, thereby causing those cycles to be longer.

**Observation:** If a slow slave is replaced with an equivalent but faster device in a partially asynchronous bus interface, cycle stretching may be reduced and the computer will execute faster.

**Common error:** If the slave cannot respond fast enough or if no slave exists at the address, data are not properly read or written, but no bus error signal is generated.

### 9.3.3 Fully Asynchronous Bus Timing

The above two approaches cannot generate a hardware bus error signal when the slave does not or cannot respond. Another limitation of the previous methods is the difficulty in upgrading. To make a synchronous system run faster, each module (master and all slaves) must be redesigned. To upgrade a partially asynchronous system, the new

cycle stretching amount must be determined and programmed into the software. In a *fully asynchronous bus* interface, there are control and acknowledge handshake signals that are generated for each bus cycle. Each phase in the following example is signified by the rise or fall of a control signal. The particular example is similar to the protocol used in the SCSI. The key to a fully asynchronous transfer (also called *handshaked* or *interlocked*) is that each phase can vary in length and will wait for the previous phase to occur.

#### **Read Cycle Data Transferred from Slave to Master (Figure 9.14).**

**Phase 1.** Master specifies the address and says, “Please give me data”

I/O=0 and fall of REQ (SCSI)

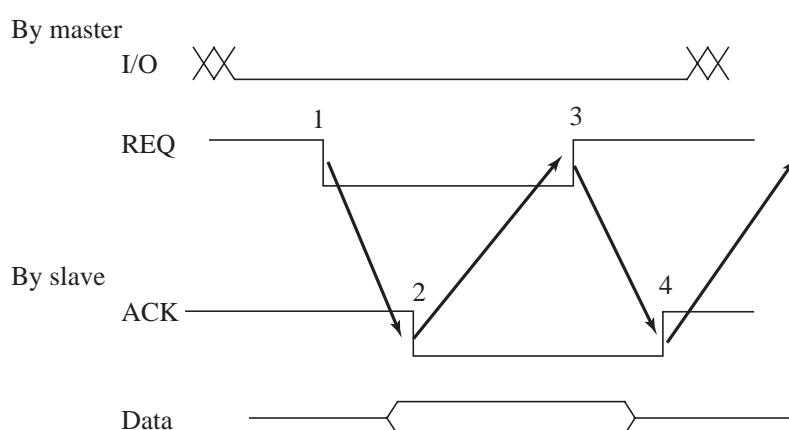
**Phase 2.** Slave puts data on the bus and sends an acknowledge saying, “Here is the data”  
data are driven and fall of ACK (SCSI)

**Phase 3.** Master accepts the acknowledge and says “Thank you”  
rise of REQ (SCSI)

**Phase 4.** Slave makes its data HiZ and says “You’re welcome”  
data are driven and rise of ACK (SCSI)

**Figure 9.14**

Fully asynchronous (interlocked or handshaked) read cycle bus timing.



#### **Write Cycle Data Transferred from Master to Slave (Figure 9.15).**

**Phase 1.** Master puts data on the bus and says, “Please save these data”

I/O=1 data driven by master and fall of REQ (SCSI)

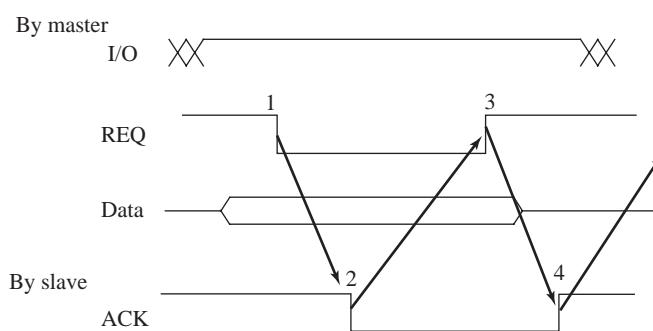
**Phase 2.** Slave accepts the data and sends an acknowledge saying, “I saved the data”  
fall of ACK (SCSI)

**Phase 3.** Master makes its data HiZ and says “Thank you”  
rise of REQ (SCSI)

**Phase 4.** Slave says “You’re welcome”  
rise of ACK (SCSI)

**Figure 9.15**

Fully asynchronous (interlocked or handshaked) write cycle bus timing.



To pass each data correctly and to prevent the same data from being passed twice, there are four transitions in a fully interlocked protocol.

**Observation:** If you place a time-out mechanism on the time the master waits for the slave response, then a hardware bus error can be generated on a broken or missing module.

**Observation:** If you replace any module in a fully asynchronous bus system, then the data transfer automatically occurs at the fastest possible rate without changing the software configuration.

Most computer systems do not implement fully asynchronous bus transfers with their memory because the complexity makes the system too expensive and too slow. On the other hand, fully asynchronous transfers are used in many I/O bus protocols such as IEEE-488 and SCSI.

## 9.4 External Bus Timing

### 9.4.1 Synchronized Versus Unsyncronized Signals

When designing the control signals for our memory, we have two choices to make. The first choice is relatively easy: Do we use positive or negative logic? A positive logic control signal goes high when the memory is accessed, and a negative logic signal goes low during the cycle of interest. The other choice is whether or not to synchronize to the E clock. These choices are presented in Table 9.4, where **command** refers to a signal derived directly from A15–A0 or R/W. In this discussion, **command**=1 means activate this cycle. We will also use this table to design the digital logic to create the control signal **CS**.

E	Command	Unsynchronized Positive	Unsynchronized Negative	Synchronized Positive	Synchronized Negative
0	0	0	1	0	1
1	0	0	1	0	1
0	1	1	0	0	1
1	1	1	0	1	0
		CS=command	CS=not(command)	CS=E-command	CS=not(E-command)

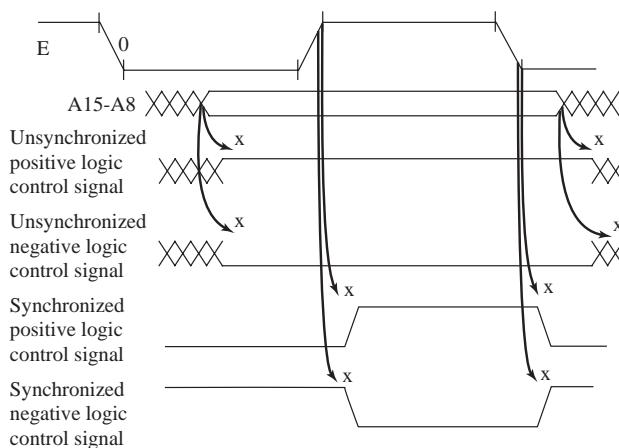
**Table 9.4**

A timing signal (like the E clock) can be combined with a positive logic command signal four ways.

The four possible shapes for the control signals are shown in Figure 9.16, where the “x” parameter refers to the gate delays through the digital logic required to implement the control signal. If we have a negative logic command signal, then the polarity of the command is reversed (Table 9.5).

**Figure 9.16**

The timing of four types of control signals.



E	<u>command</u>	Unsynchronized Positive	Unsynchronized Negative	Synchronized Positive	Synchronized Negative
0	1	0	1	0	1
1	1	0	1	0	1
0	0	1	0	0	1
1	0	1	0	1	0
		CS = not( <u>command</u> )	CS = <u>command</u>	CS = E·not( <u>command</u> )	CS = not(E·not( <u>command</u> ))

**Table 9.5**

A timing signal (like the E clock) can be combined with a negative logic command signal four ways.

#### 9.4.2. Freescale MC9S12C32 External Bus Timing

The MC9S12C32 can run in one of eight modes, as listed in Table 9.6. The initial mode is determined by the values of the MODC MODB MODA pins at the time of the rise of RESET. Special and emulation modes are for testing and system development, whereas normal modes are intended for embedded products. In particular, special mode allows you to access

MODC	MODB	MODA	Mode Description	Port A	Port B	MODx Write Ability
0	0	0	Special Single Chip	In/Out	In/Out	Write anytime, but not to peripheral
0	0	1	Emulation Expanded Narrow	A15-A8/ D7-D0	A7-A0	Cannot change mode
0	1	0	Special Test	A15-A8/ D15-D8	A7-A0/ D7-D0	Write anytime, but not to peripheral
0	1	1	Emulation Expanded Wide	A15-A8/ D15-D8	A7-A0/ D7-D0	Cannot change mode
1	0	0	Normal Single Chip	In/Out	In/Out	Write once to Normal Expanded Narrow or Wide
1	0	1	Normal Expanded Narrow	A15-A8/ D7-D0	A7-A0	Cannot change mode
1	1	0	Peripheral	—	—	Cannot change mode
1	1	1	Normal Expanded Wide	A15-A8/ D15-D8	A7-A0/ D7-D0	Cannot change mode

**Table 9.6**

There are eight execution modes for the MC9S12C32.

many test registers that are inaccessible in normal mode. In the *Single Chip* modes, Ports A, B, E, and K are available for general-purpose input/output. In the *Expanded* modes, Port E contains the bus control signals, and Port K contains the X19-14 address lines used in paging. In the *Expanded Narrow* mode, Port A implements the time-multiplexed A15-A8 address D7-D0 data bus, and Port B implements A7-A0, the low 8 bits of the address bus. In the *Expanded Wide* mode, Ports A and B implement the time-multiplexed 16-bit address 16-bit data bus. When paging is used, a 20-bit address allows access to external memory up to 1 Megabyte. *Peripheral* mode is used by Freescale for testing.

When a 16-bit access is required in expanded narrow mode, it is performed by two sequential 8-bit accesses. In most situations, running in expanded narrow mode is twice as slow as running in expanded wide mode. Table 9.7 shows the registers we use to configure the MC9S12C32 expanded mode external bus.

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	PORTA
\$0001	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	PORTB
\$0002	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	DDRA
\$0003	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	DDRB
\$0008	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0	PORTE
\$0009	DDRE7	DDRE6	DDRE5	DDRE4	DDRE3	DDRE2	0	0	DDRE
\$000A	NOACCE	0	PIPOE	NECLK	LSTRE	RDWE	0	0	PEAR
\$000B	MODC	MODB	MODA	0	IVIS	0	EMK	EME	MODE
\$000C	PUPKE	0	0	PUPEE	0	0	PUPBE	PUPAE	PUCR
\$000D	RDPK	0	0	RDPE	0	0	RDPB	RDPA	RDRIV
\$000E	0	0	0	0	0	0	ESTR	EBICTL	
\$0013	0	0	0	0	EXSTR1	EXSTR0	ROMHM	ROMON	MISC
\$001E	IRQE	IRQEN	0	0	0	0	0	0	IRQCR
\$0032	PK7	PK6	PK5	PK4	PK3	PK2	PK1	PK0	PORTK
\$0033	DDRK7	DDRK6	DDRK5	DDRK4	DDRK3	DDRK2	DDRK1	DDR0	DDRK
\$0034	0	0	SYN5	SYN4	SYN3	SYN2	SYN1	SYN0	SYNR
\$0033	0	0	0	0	REFDV3	REFDV2	REFDV1	REFDV0	REFDV
\$0037	RTIF	PROF	0	LOCKIF	LOCK	TRACK	SCMIF	SCM	CRGFLG
\$0039	PLLSEL	PSTP	SYSWAI	ROAWAI	PLLWAI	CWAI	RTIWI	COPWAI	CLKSEL
\$003A	CME	PLLON	AUTO	ACQ	0	PRE	PCE	SCME	PLLCTL

**Table 9.7**

MC9S12C32 registers used for external memory interfacing.

As the name implies, the **MODE** register specifies the operating mode (i.e., **MODC MODB MODA**), but there is a complex set of rules about changing the mode, as listed in the last column of Table 9.6. If MODA = 1, then MODC, MODB, and MODA cannot be changed. If MODC = MODA = 0, then MODC, MODB, and MODA are writable with the exception that you cannot change to special peripheral mode. If MODC = 1, MODB = 0, and MODA = 0, then MODC cannot be changed. In this case, MODB and MODA are write once, except that you cannot change to special peripheral mode. From normal single-chip, only normal expanded narrow and normal expanded wide modes are available. **IVIS** is the Internal Visibility bit. This bit determines whether internal bus signals can be seen on the external bus during accesses to internal locations. **EMK** and **EME** are the Emulate Port K and E bits. When these bits are set to one while in an expanded mode, PORTK/PORTE and DDRK/DDRE are removed from the internal memory map. In single-chip mode PORTK and PORTE are always in the map regardless of the state of these bits. Removing the registers from the map allows us to emulate the function of these registers externally.

Another register that affects expanded mode interfacing is **PEAR**. The NOACCE, PIPEO, LSTRB, and RDWE bits can be written once in normal mode. This means the initialization software can write to the PEAR register once; then the software will not be able to change any of these bits. In special mode, the bits of PEAR can be written anytime, but in emulation mode, they can not be written. The NOACCE, PIPEO, LSTRB, and RDWE bits have no effect in single-chip or special peripheral modes. **NOACCE** is the CPU No Access Output Enable bit. If NOACCE = 1, PE7 is an output and indicates whether the cycle is a CPU free cycle. If NOACCE = 0, then PE7 is general-purpose I/O. Recall from Chapter 1 that when an instruction is executed, there are four phases: fetch instruction, read memory data, operate, and write memory data. Free cycles occur when the processor is busy executing an instruction and does not require the memory bus. In this situation, the bus interface unit will issue a memory read cycle but ignore the data. **PIPOE** is the Pipe Status Signal Output Enable bit. If PIPOE = 1, then PE6:PE5 are outputs and indicate the state of the instruction queue. If PIPOE = 0, then PE6:PE5 are general-purpose I/O. **NECLK** is the No External E Clock bit. As is not the case with the other bits in the PEAR register, we can write to this bit anytime while in Normal and Special modes. If NECLK = 1, PE4 is a general-purpose I/O pin. If NECLK = 0, then PE4 is the external E clock pin. The PE4 external E clock is free-running if ESTR = 0. **LSTRE** is the Low Strobe (LSTRB) Enable bit. The signal LSTRB is used to implement 8-bit writes when running in expanded wide mode. LSTRE = 1 means PE3 is configured as the LSTRB bus control output; otherwise PE3 is a general-purpose I/O pin. After reset in normal expanded mode, LSTRB is disabled to provide an extra I/O pin. If LSTRB is needed, it should be enabled when using expanded narrow mode. External reads do not normally need LSTRB because all 16 data bits can be driven even if the system needs only 8 bits of data. **RDWE** is the Read/Write Enable bit. If RDWE = 1, PE2 is configured as the R/W pin. The R/W signal is 1 during read cycles and 0 during write cycles. If RDWE = 0, then PE2 is a general-purpose I/O pin. After reset in normal expanded mode, R/W is disabled to provide an extra I/O pin. If R/W is needed, it should be enabled before any external writes.

**Observation:** LSTRB is necessary in expanded wide modes but is not needed for expanded narrow interfaces.

The **EBICTL** register contains one bit called **ESTR**, which determines whether the E clock behaves as a simple free-running clock or as a bus control signal that is active only for external bus cycles. We will set ESTR to 1 when interfacing external memory so that E stretches high during stretched external accesses and remains low during non-visible internal accesses. If ESTR is 0, then E never stretches (always free running). This bit has no effect in single-chip modes.

In order to allow fast internal bus cycles to coexist in a system with slower external memories, the 9S12C32 supports the concept of stretched bus cycles. The **EXSTR1** **EXSTR0** bits in the **MISC** register specify the amount of stretch for all external memory cycles, as described in Table 9.8 and illustrated in Figures 9.17 through 9.20. While stretching, the CPU state machines are all held in their current state. At this point in the CPU bus cycle, write data would already be driven onto the data bus so that the length of time write data is valid is extended. Read data would not be captured by the system until the E clock falling edge. In the case of a stretched bus cycle, read data is not required until the specified setup time before the falling edge of the stretched E clock. The LSTRB and R/W signals remain valid during the period of stretching (throughout the stretched E high time). **ROMHM** is the FLASH EEPROM Only in Second Half of Memory Map bit. When ROMHM = 1, it disables direct access to the FLASH EEPROM or ROM in the lower half of the memory map. These physical locations of the FLASH EEPROM or ROM can still be accessed through the Program Page window. When ROMHM = 0, the fixed page(s) of FLASH EEPROM or ROM in the lower half of the memory map can be accessed. **ROMON**

is the Enable Flash EEPROM bit. If the internal RAM, registers, EEPROM, or BDM ROM (if active) are mapped to the same space as the Flash EEPROM, they will have priority over the Flash EEPROM. ROMON = 0 disables the Flash EEPROM, and ROMON = 1 enables the Flash EEPROM.

**Table 9.8**

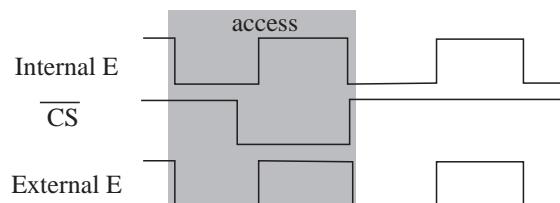
E clock stretching for the external access space.

EXSTR1	EXSTR0	E Clock Stretch
0	0	None (default)
0	1	1
1	0	2
1	1	3

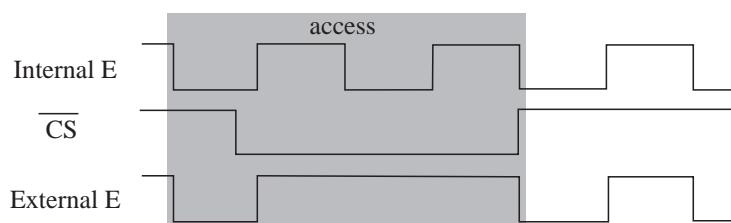
With no cycle stretching, the external E clock follows the internal E clock (Figure 9.17). With one-cycle stretching, the access cycle is 250 ns long (Figure 9.18). With two-cycle stretching, the access cycle is 375 ns long (Figure 9.19). With three-cycle stretching, the access cycle is 500 ns long (Figure 9.20).

**Figure 9.17**

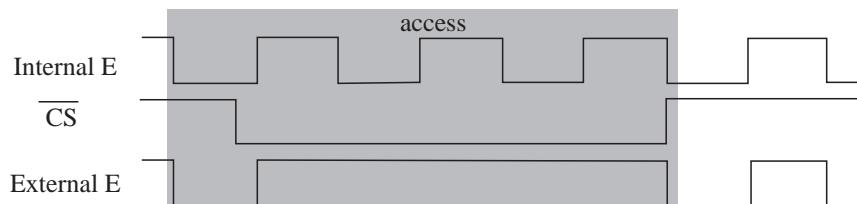
CS timing with no cycle stretching.

**Figure 9.18**

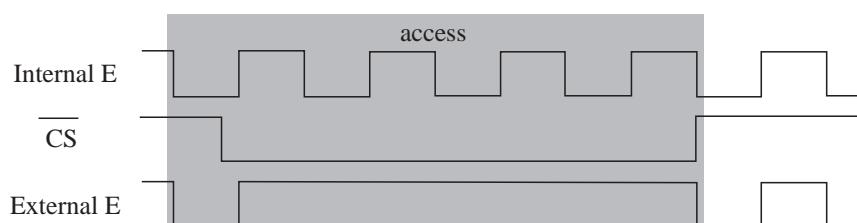
CS timing with one-cycle stretching.

**Figure 9.19**

CS timing with two-cycle stretching.

**Figure 9.20**

CS timing with three-cycle stretching.



**Observation:** The address bus portion of the memory cycle is not stretched, just the data portion.

**Observation:** TCNT, Pulse Accumulator, PWM, RTI, SCI, and SPI are not affected by bus stretching.

Accesses to the PORTA, PORTB, PORTE, and PORTK registers should be made only when the corresponding pins are configured as general purpose I/O. These port and direction registers have no function when a pin is configured as an external memory bus signal. Similarly, we use the Pull-Up Control Register (**PUCR**) to enable pull-up resistors on the corresponding ports when used as general purpose inputs. We use the Reduced Drive Register (**RDRIV**) to save power when using the corresponding ports as general-purpose outputs.

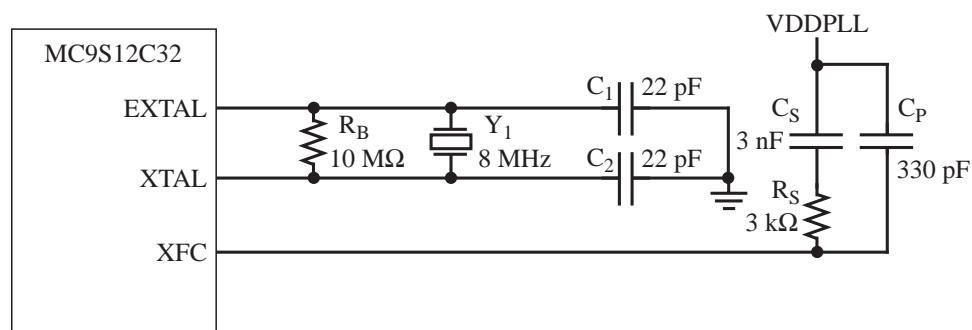
The alternate function of PE1 is IRQ, and the alternate function of PE0 is XIRQ. In particular, these pins, which are always input, can be used to request interrupts. **IRQEN** is the External IRQ Enable bit. We set IRQEN to one to use PE1 as an external IRQ interrupt request signal. We make IRQE = IRQEN = 1 to configure the 9S12C32 to request IRQ interrupts on falling edges of PE1/IRQ. We set IRQEN = 1 and IRQE = 0 to configure PE1/IRQ as low-level interrupt request signal. We clear IRQEN to use PE1 as a general-purpose input. To use PE0 as a general-purpose input, we simply never enable XIRQ interrupts (i.e., we leave the X-bit in the CCR equal to 1)

**Observation:** To use low-level IRQ interrupts, the software needs to be able to acknowledge the interrupt making IRQ = 1 again before returning from the interrupt service routine.

**Common error:** When using PE1 as a regular input, the software must make IRQEN = 0, because the default value of this bit is enable.

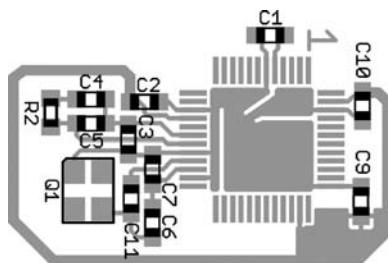
The E clock is a timing signal of the MC9S12C32, meaning the rising and falling edges occur at predictable and precise times. The clock frequency is determined by a crystal placed between the EXTAL and XTAL inputs, as shown in Figure 9.21. The MC9S12C32 system requires an external crystal that is twice the E clock frequency. For example, a 4 MHz E clock period is created using an 8 MHz crystal. The values of  $R_B$ ,  $C_1$ , and  $C_2$  are suggested by the crystal manufacturer (Figure 9.22). The MC9S12C32 allows the software to modify the E clock frequency by programming parameters to the phase-lock loop (PLL). The external components connected to XFC help the PLL circuitry create E clock frequencies faster than the crystal frequency. The values of  $R_s$ ,  $C_s$ , and  $C_p$  are calculated from the ratio of the crystal frequency to PLL frequency, and the design equations can be found in the 9S12C data sheet.

**Figure 9.21**  
MC9S12C32 clock circuit; values taken from the Technological Arts Nanocore12.



**Figure 9.22**

Layout for the clock crystal, Q1, on the 48-pin 9S12C32.



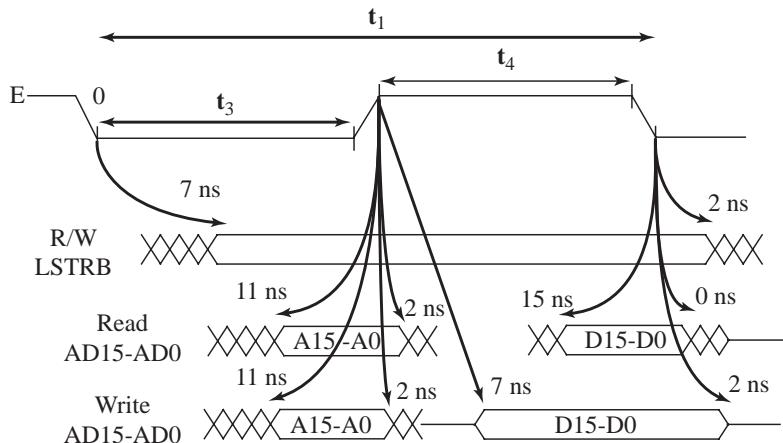
**Observation:** Microcontroller data sheets suggest a PCB layout pattern for interfacing the crystal.

Figure 9.23 is a simplified timing diagram, showing both the read and write timing. The bus cycle time is  $t_1$ . For example, the 8 MHz crystal (period=125 ns) in Figure 9.21 will create a  $t_1$  of 250 ns without cycle stretching. Let OSCCLK be the frequency of the crystal, and let PLLCLK be the frequency of the PLL. The **SYNR** and **REFDV** registers determine the PLL frequency,

$$\text{PLLCLK} = 2 * \text{OSCCLK} * \frac{(\text{SYNR} + 1)}{(\text{REFDV} + 1)}$$

**Figure 9.23**

Simplified bus timing for the MC9S12C32 in expanded mode.



The **CLKSEL** register contains the PLL Select Bit, **PLLSEL**. When **PLLSEL** is 1, the system clocks are derived from **PLLCLK** (bus clock frequency is **PLLCLK**/2). When **PLLSEL** is 0, the system clocks are derived from the crystal oscillator (bus clock frequency is **OSCCLK**/2). Let  $t_{\text{cyc}}$  be the period of the bus clock.

$$t_{\text{cyc}} = \frac{2}{\text{OSCCLK}} \text{ (if } \text{PLLSEL} = 0 \text{)} \quad \text{or} \quad t_{\text{cyc}} = \frac{2}{\text{PLLCLK}} \text{ (if } \text{PLLSEL} = 1 \text{)}$$

We can execute software that activates the PLL, creating a bus cycle time as short as 40 ns (25 MHz). Program 9.1 shows the sequence of steps required to engage the PLL. First, it sets the **SYNR** and **REFDV** registers to specify the PLL frequency. Next, it sets the **PLLCTL** register. The Clock Monitor Enable Bit (**CME**) is set to enable the clock monitor, so that slow or stopped clocks will cause a clock monitor reset sequence. The Phase

Lock Loop On Bit (**PLLON**) is set to turn on the PLL. The Acquisition Bit (**ACQ**) is set to select the high bandwidth filter. The Self Clock Mode Enable Bit (**SCME**) is set so the detection of crystal clock failure forces the MCU in Self Clock Mode.

**Checkpoint 9.3:** Modify Program 9.1 to change from 8 to 24 MHz

### Program 9.1

MC9S12C32 code to change the E clock from 4 to 24 MHz.

```
void PLL_Init(void){
    CLKSEL = 0x00;           // make sure PLL is deselected
    SYNR = 2;    REFDV = 0;   // PLLCLK=2*OSCCLK*(SYNR+1)/(REFDV+1)
    PLLCTL = 0xD1;          // Turn on PLL
    while((CRGFLG&0x08) == 0){} // Wait for PLLCLK to stabilize.
    CLKSEL |= 0x80;          // Switch to PLL clock
```

When stretching the E clock, confusion sometimes arises. Changing the bus cycle time with the PLL will modify both halves of the bus cycle,  $t_3$  and  $t_4$ . On the other hand, when a bus cycle is stretched, times  $t_1$ ,  $t_4$  are increased, whereas  $t_3$  is fixed. The rising edge of the E clock occurs at the same time regardless of stretching, which will be at  $1/2t_{cyc}$ . Notice in Figures 9.17 through 9.20 that there is a stretched E clock and an unstretched E clock. The clocks used by SPI, SCI, PulseAcc, PWM, and TCNT are derived from the unstretched E clock. External memory accesses use the stretched E clock. In other words, the only activity affected by cycle stretching is the time for external memory accesses and hence the execution of the instructions causing those accesses. Internal memory accesses, the timer, and I/O functions are not affected by cycle stretching. In Figure 9.23, the time delays with numerical values (e.g., 7 ns, 11 ns, etc.) are parameters of the MC9S12C32 and not affected by the PLL or cycle stretching. Let  $n$  be the number of stretches (0, 1, 2, or 3) as specified by the **EXSTR1** **EXSTR0** bits. When interfacing external memory, we need to know when the E clock rises and when it falls.

$$\uparrow E = \frac{1}{2}t_{cyc}$$

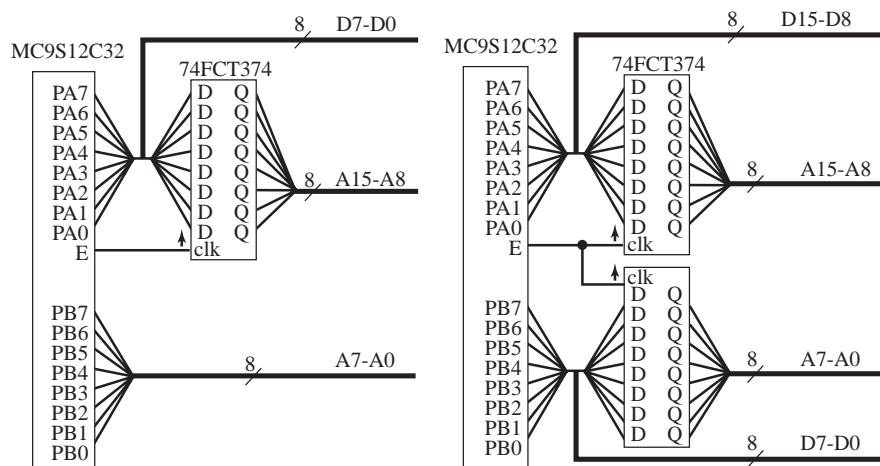
$$\downarrow E = t_1 = (n + 1) * t_{cyc}$$

**Observation:** Most of the activities (such as executing instructions, memory access, TCNT, SCI, SPI) are affected by the PLL. Conversely, RTI interrupts always operate using the oscillator crystal.

In expanded narrow mode, the data bus is only 8 bits (on pins PA7–PA0), and the low address is available throughout the cycle (on pins PB7–PB0). In narrow mode, we need only one octal latch to capture the A15–A8 address, as shown in Figure 9.24. In expanded wide mode,

**Figure 9.24**

Address latch for the MC9S12C32 in expanded narrow and wide modes.



the data bus is 16 bits, and we need two octal latches to capture the A15–A0 address from PA7–PA0 on the rising edge of E. The 74FCT374 octal flip flop is chosen because of its speed and the fact that it captures on the rising edge of the clock. The setup and hold times for the 74FCT374 are 2 ns and 1.5 ns, respectively. In Figure 9.23, we see the 9S12 maintains the address 11 ns before and 2 ns after the rise of E, so the setup and hold times of the octal D flip-flop are satisfied. The propagation delay from clock to output valid is [2,6.5] ns.

There are three important timing intervals we must determine from the microcomputer timing diagram. The first is the address available (AA) interval. This interval defines when during the cycle the address is valid. In particular, AdV is the time when the address lines A15–A0 are valid, and AdN is the time when the address lines are no longer valid. In order to calculate when the address is available, we first consider the 74FCT374 octal D flips used to capture the address. In expanded wide mode, all 16 bits of the address are clocked into the 74FCT374 on the rising edge of E, so the address is available during the second half of the cycle and continues to be valid until the next rising edge of E. The +[2,6.5] occurs because of the delay in the 74FCT374s.

$$\text{AA} = (\text{AdV}, \text{AdN}) = (\frac{1}{2}t_{\text{cyc}} + [2,6.5], t_1 + \frac{1}{2}t_{\text{cyc}})$$

In expanded narrow mode, the least significant address lines are not latched; therefore, AdV is determined by the latched address, and AdN is determined by the unlatched address, which has the same timing as R/W in Figure 9.40.

$$\text{AA} = (\text{AdV}, \text{AdN}) = (\frac{1}{2}t_{\text{cyc}} + [2,6.5], t_1 + 2)$$

The second important timing interval is read data required (RDR). During a read cycle the data are required by the MC9S12C32. Thus, to determine the data required interval, we look in the MC9S12C32 data sheet. For data required, the worst case is the longest interval.

$$\text{RDR} = \text{Read Data Required} = (t_1 - 15, t_1)$$

The last important timing interval we get from the microcomputer is write data available (WDA). During a write cycle, the data are supplied by the MC9S12C32. Thus, to determine the data available interval we look in the MC9S12C32 data sheet. We will specify the worst-case timing. For write data available, the worst case is the shortest interval.

$$\text{WDA} = \text{Write Data Available} = (\frac{1}{2}t_{\text{cyc}} + 7, t_1 + 2)$$

**Observation:** The speed of CMOS logic is strongly dependent on capacitive load. The 9S12C32 and 74FCT374 timings are specified for capacitive loads of 50 pF. If the actual load is larger than 50 pF, the timing will be significantly slower.

Table 9.9 presents these three intervals calculated for a 250 ns cycle time with 0,1,2,3 stretches. The address times are given for expanded narrow mode with the shortest interval.

**Table 9.9**

Timing intervals for the MC9S12C32 with a 4-MHz clock.

n	0	1	2	3
$\uparrow E = \frac{1}{2}t_{\text{cyc}}$	125	125	125	125
$\downarrow E = t_1$	250	500	750	1000
AA	(131.5,252)	(131.5,502)	(131.5,752)	(131.5,1002)
RDR	(235,250)	(485,500)	(735,750)	(985,1000)
WDA	(132,252)	(132,502)	(132,752)	(132,1002)

**Checkpoint 9.4:** What are AA, RDR, and WDA with no stretches, if  $t_{\text{cyc}}$  is 100 ns?

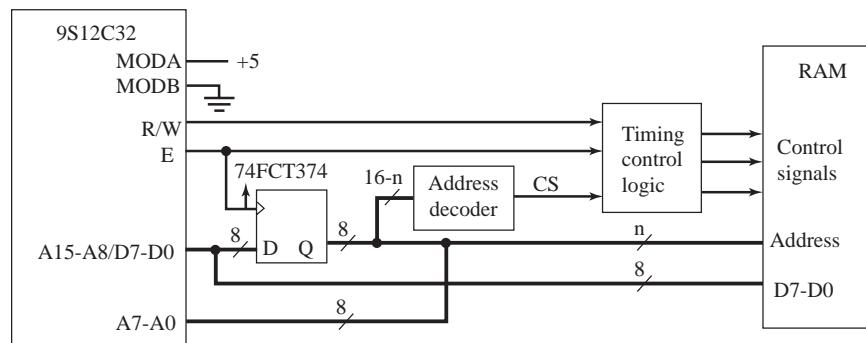
**Checkpoint 9.5:** What are AA, RDR, and WDA with 2 stretches, if  $t_{\text{cyc}}$  is 40 ns?

## 9.5 General Approach to Interfacing

### 9.5.1 Interfacing to a 9S12 in Expanded Narrow Mode

In expanded narrow mode, all external memory accesses utilize only 8 bits of data. We connect MODA, MODB so that the 9S12 executes in expanded narrow mode. The 9S12 will automatically divide 16-bit reads and writes into two separate 8-bit accesses. The MC9S12C32 interface will require an address decoder to generate the chip select, CS (Figure 9.25). A common reason for using narrow mode is that many I/O devices support only 8-bit accesses.

**Figure 9.25**  
General approach to memory interfacing on an MC9S12C32 in narrow expanded mode.

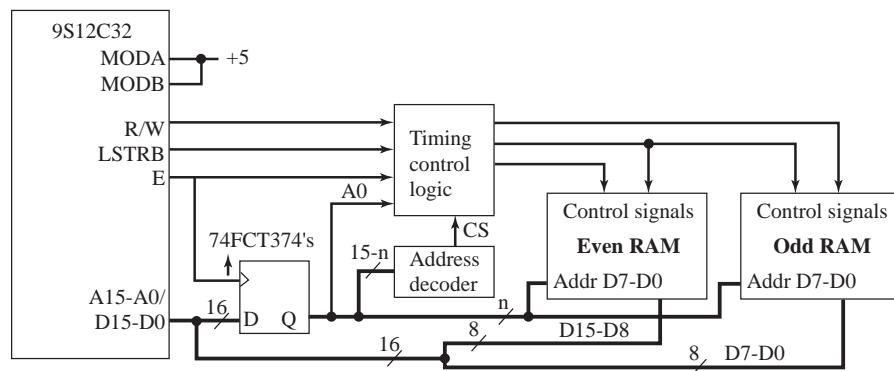


If the RAM contains  $2^n$  bytes, then the low  $n$  bits of the address go directly to the RAM and specify which cell to access. If the CS signal is true, the *Timing Control Logic* will generate appropriate control signals for the memory. The control signals will activate a read operation (drive data out of memory onto the bus) if the CS is true and R/W is 1. Similarly, the control signals will activate a write operation (store data from the bus into the memory) if the CS is true and R/W is 0. The software must configure the cycle stretching, and enable the E clock and R/W signals as needed in the PEAR register.

### 9.5.2 Interfacing to a 9S12 in Expanded Wide Mode

In expanded wide mode, external memory accesses utilize 16 bits of data. We connect MODA, MODB so that the computer executes in expanded wide mode: 16-bit reads and writes to even addresses (aligned) will be performed in a single access; 16-bit reads and writes to odd addresses (misaligned) will be performed in two separate 8-bit accesses. The LSTRB (=0 when low byte data are valid) is used to implement 8-bit memory writes. The advantage of wide mode is execution speed, because opcode fetches will always occur as aligned 16-bit reads (Figure 9.26).

**Figure 9.26**  
General approach to memory interfacing on an MC9S12C32 in wide expanded mode.



The typical approach to creating a 16-bit memory is to use two 8-bit memories called *even* and *odd*. Assume the size of the memory is  $2^n$  words (16 bits each). The two memory chips share address lines (An–A1) and most control signals. The MC9S12C32 will require an address decoder that accepts the most significant  $15 - n$  address lines and produces the

chip select, **CS**. The 16-bit data bus is divided so that 8 bits go to each chip. The “big endian” format places the most significant data, D15–D8, into the even address and the least significant data, D7–D0, into the odd address. When the 9S12 performs an 8-bit read, there is usually no problem if the memory system responds with 16 bits of data (the processor simply takes the data it wants). On the other hand, if the 9S12 performs an 8-bit write, it would be a mistake to save all 16 bits D15–D0 into the memory system.

The external signals LSTRB, R/W, and A0 can be used to determine the type of bus access. Accesses to the internal RAM module are the only situation that requires the even and odd bytes to be swapped, LSTRB = A0 = 1 (Table 9.10). The internal RAM is specifically designed to allow misaligned 16-bit accesses in a single cycle. In a misaligned 16-bit access, the data for the address that was accessed are on the low half of the data bus and the data for address +1 are on the high half of the data bus. We will use A0 and LSTRB to handle 8-bit data writes while running in 16-bit data mode.

LSTRB	A0	R/W	Type of Access	Even RAM	Odd RAM
1	0	1	8-bit read of an even address	Activate	No action
0	1	1	8-bit read of an odd address	No action	Activate
1	0	0	8-bit write of an even address	Activate	No action
0	1	0	8-bit write of an odd address	No action	Activate
0	0	1	16-bit read of an even address	Activate	Activate
1	1	1	16-bit read of an odd address*	Not applicable	Not applicable
0	0	0	16-bit write to an even address	Activate	Activate
1	1	0	16-bit write to an odd address*	Not applicable	Not applicable

\*Low/high data swapped.

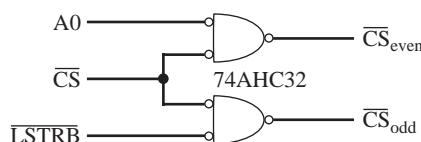
**Table 9.10**

LSTRB, A0, R/W specify when to activate even and odd RAM modules.

Notice that we should activate the *even RAM* when  $\overline{CS} = 0$  and  $A0 = 0$ , and we should activate the *odd RAM* when  $\overline{CS} = 0$  and  $LSTRB = 0$ . These digital logic functions can be implemented with one 74AHC32 package. AHC or FCT logic is used because the propagation delay is less than 10 ns (Figure 9.27).

**Figure 9.27**

Circuit needed to handle even and odd 8-bit accesses.



The software must configure the cycle stretching and enable the E clock, LSTRB, and R/W signals as needed in the PEAR register. The NDRC bit in the MISC register can be set to configure the 512 bytes following the register space (typically \$0200–\$03FF) for narrow mode. When NDRC is set, all addresses except \$0200–\$03FF can utilize the 16-bit data bus.

**Example 9.6** Interface the Motorola MCM60L64 8192 by 8-bit static RAM to the microcomputer. The memory will be placed at \$8000 to \$9FFF.

**Solution** The **command portion** determines which CPU cycles will activate the RAM. Using full decoding, the address **Select\*** line in negative logic is

$$\text{Select}^* = \overline{A15} \cdot A14 \cdot A13$$

This RAM chip has both a positive logic (**E2**) and a negative logic ( $\overline{\text{E1}}$ ) chip select. Negative logic is used because the output of the 74FCT139 decoder is in negative logic. With  $\text{E2}$  tied to + 5 V, the RAM has the functions shown in Table 9.11.

**Table 9.11**  
Function table for the MCM60L64 RAM.

$\overline{\text{E1}}$	$\overline{\text{W}}$	$\overline{\text{G}}$	Function
1	X	X	Disabled, low $I_{cc} = 30 \mu\text{A}$
0	1	1	Disabled, high $I_{cc} = 3 \text{ mA}$
0	0	X	Write data into RAM
0	1	0	Read data out of RAM

To reduce power, we will let  $\overline{\text{E1}} = 0$  only during accesses to this RAM. We will synchronize the read and write functions by synchronizing either  $\overline{\text{E1}}$  or both  $\overline{\text{W}}$  and  $\overline{\text{G}}$ . A read operation will occur when both  $\overline{\text{E1}}$  and  $\overline{\text{G}}$  are 0 and  $\overline{\text{W}}$  is 1.

$$\text{RD} = \overline{\overline{\text{E1}}} \cdot \overline{\text{W}} \cdot \overline{\overline{\text{G}}}$$

Similarly, a write operation will occur when both  $\overline{\text{E1}}$  and  $\overline{\text{W}}$  are zero.

$$\text{WR} = \overline{\overline{\text{E1}}} \cdot \overline{\overline{\text{W}}}$$

The signals RD and WR are generated internal to the RAM. The command part of the design will have RD = 1 only during read cycles from this RAM, and WR = 1 only during write cycles to this RAM (Table 9.12).

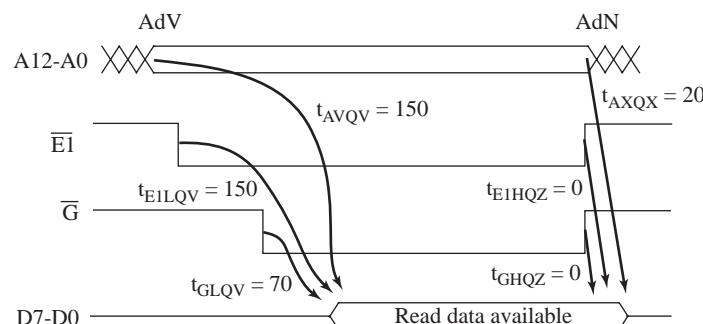
**Table 9.12**  
Command table for the MCM60L64 RAM interface.

Select	R/W	RD	WR	$\overline{\text{E1}}$	$\overline{\text{W}}$	$\overline{\text{G}}$	Function
0	0	0	0	1	X	X	Disable because address not on the chip
0	1	0	0	1	X	X	Disable because address not on the chip
1	0	0	1	0	0	X	Write cycle to this RAM
1	1	1	0	0	1	0	Read cycle from this RAM

The **timing portion** of the interface determines the timing of the rise and fall of the control signals. The objective is to design the interface such that the data available interval overlaps the data required. Since this is a RAM, we consider both the read and write cycles. Data are read from the memory when  $\overline{\text{E1}} = 0$ ,  $\overline{\text{G}} = 0$ , and  $\overline{\text{W}} = 1$ . During a read cycle, the data are supplied by the 60L64. Thus, to determine the read data available interval we look in the Motorola MCM60L64 data sheet. We will specify the worst-case timing. For read data available, the worst case is the shortest interval (Figure 9.28).

$$\text{Read data available} = (\text{later}(\text{AdV} + t_{AVQV}, \downarrow \overline{\text{E1}} + t_{E1LQV}, \downarrow \overline{\text{G}} + t_{GLQV}), \\ \text{earlier}(\text{AdN} + t_{AXQX}, \uparrow \overline{\text{E1}} + t_{E1HQZ}, \uparrow \overline{\text{G}} + t_{GHQZ}))$$

**Figure 9.28**  
Read timing for the 60L64 8K RAM chip.



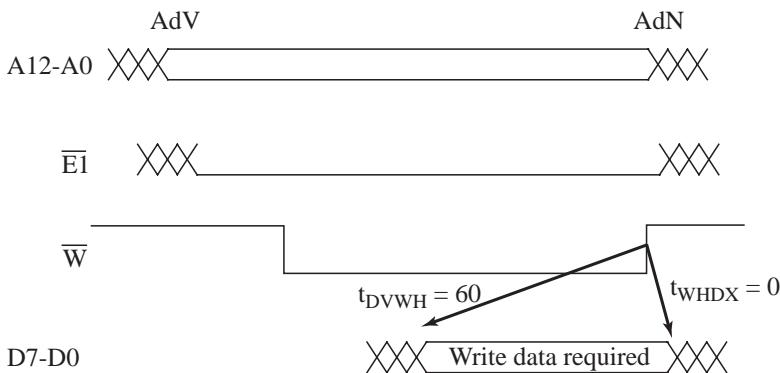
where AdV is the time when the address lines A12–A0 are valid and AdN is the time when the address lines are no longer valid. From the 60L64 data sheet,  $t_{AVQV} = 150$ ,  $t_{EILQV} = 150$ ,  $t_{GLQV} = 70$ ,  $t_{AXQX} = 20$ ,  $t_{EIHQZ} = 0$ , and  $t_{GHQZ} = 0$ .

During a write cycle, the data are required by the 60L64. Thus, to determine the write data required interval, we look in the 60L64 data sheet. Since the write operation occurs on the overlap of  $\overline{E1}$  and  $\overline{W}$ , the first of these two signals to rise will cause data to be written into the memory. There are two possible timing diagrams for the write cycle. If  $\overline{E1}$  is unsynchronized negative logic and  $\overline{W}$  is synchronized negative logic, then it is the rise of  $\overline{W}$  that stores data into the RAM (Figure 9.29).

$$\text{Write data required} = (\uparrow \overline{W} - t_{DVWH}, \uparrow \overline{W} + t_{WHDX}) = (\uparrow \overline{W} - 60, \uparrow \overline{W})$$

**Figure 9.29**

Write timing controlled by  $\overline{W}$  for the 60L64 8K RAM chip.

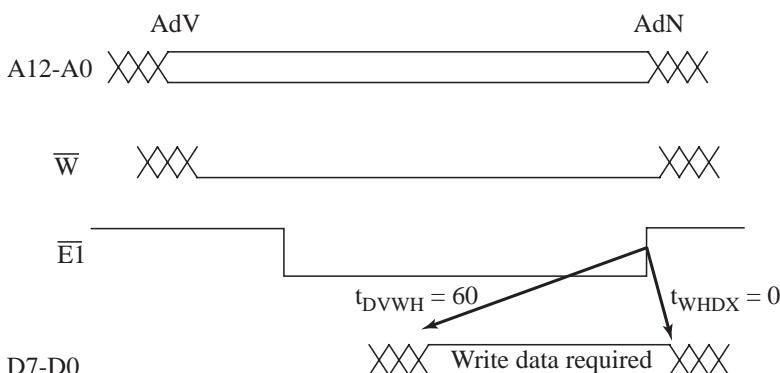


For write data required, the worst case is the longest interval. From the 60L64 data sheet, the setup time  $t_{DVWH} = 60$ , and the hold time  $t_{WHDX} = 0$ . Note also that the address must be valid whenever  $E1 = 0$ . If  $\overline{E1}$  is synchronized negative logic and  $\overline{W}$  is unsynchronized negative logic, then it is the rise of  $\overline{E1}$  that stores data into the RAM (Figure 9.30).

$$\text{Write data required} = (\uparrow \overline{E1} - t_{DVWH}, \uparrow \overline{E1} + t_{WHDX}) = (\uparrow \overline{E1} - 60, \uparrow \overline{E1})$$

**Figure 9.30**

Write timing controlled by  $\overline{E1}$  for the 60L64 8K RAM chip.



Therefore the one-cycle stretch needed for the read cycle also is sufficient for the write cycle.

**Observation:** In most situations the read cycle timing is more critical than the write cycle timing.

The design goal of this example is to interface an 8 KiB external RAM to the MC9S12C32, minimizing cost. The least expensive way to interface external RAM to the MC9S12C32 is to use expanded narrow mode. We will place the 8 KiB RAM at \$8000 to \$9FFF,

because there are no internal devices at these addresses. The address map of our system will be

\$0000-\$03FF	I/O ports
\$3800-\$3FFF	Internal RAM
\$4000-\$7FFF	Internal EEPROM
\$8000-\$9FFF	External RAM
\$C000-\$FFFF	Internal EEPROM

First, we design a minimal cost address decoder for \$8000 to \$9FFF using a 74FCT139. We need addresses A15 and A14 to differentiate our external RAM from the other three devices. The positive logic chip select will be A15  $\bar{A}14$ . We must select the proper amount of cycle stretching. Recall that for a MC9S12C32 running at 4 MHz,  $\uparrow E$  will be 125 ns, and  $\downarrow E$  will be  $(n + 1) * 250$  ns, where  $n$  is the number of stretches. The propagation delay from clock to output change through the 74FCT374 has a minimum of 2 ns and a maximum of 6.5 ns. In expanded narrow mode, the address available interval begins with the high address being clocked into the 74FCT374 and ends with the low address end time, which is 2 ns after  $\downarrow E$ . Thus,

$$AA = (AdV, AdN) = (125 + [2, 6.5], \downarrow E + 2) = ([127, 131.5], \downarrow E + 2)$$

The high address, which is used by the address decoder, is available all the way through until the next rise of  $E$ .

$$AA_{15-8} = ([127, 131.5], \downarrow E + 125)$$

To guarantee proper timing on the read and write cycles, we have three options. We must synchronize  $E2$ ,  $E1$ , or both  $\bar{G}$  and  $\bar{W}$ . The RAM has both a positive logic ( $E2$ ) and a negative logic ( $\bar{E}1$ ) chip select. Because we are going to use the positive logic chip select for the timing, we will use the negative logic chip select for the address decoder. We choose to synchronize  $E2$  because it is fastest. We need to connect directly to  $E2$  so that the end of WDA overlaps WDR. In summary, we will have

- $E2$  positive logic synchronized to  $E$
- $\bar{E}1$  negative logic unsynchronized address decoder
- $\bar{G}$  negative logic unsynchronized
- $\bar{W}$  negative logic unsynchronized

With time-multiplexed signals, we have to be careful to avoid address/data collisions during a read cycle. All read and write timing will be controlled by the  $E$  clock (without any gate delays) by connecting the  $E$  directly to  $E2$ . In this way, the memory data output will not collide with the microcomputer address output during a read cycle when  $E = 0$ . Assuming 9 ns 74FCT139 gate delay max and 1.5 ns delay minimum,  $E1$  falls at  $[127, 131.5] + [1.5, 9]$  ns and rises at  $\downarrow E + 125 + [1.5, 9]$  ns. The worst-case timing is the latest (maximum = 140.5) time for  $\downarrow E1$  and the earliest (minimum =  $\downarrow E + 126.5$ ) time for  $\uparrow E1$ .  $\bar{G}$  will be grounded, so it is removed from the timing equation. Entering this information into the memory timing

$$\begin{aligned} RDA &= (\text{later } (AdV + t_{AVQV}, \downarrow \bar{E}1 + t_{E1LQV}, \uparrow E2 + t_{E2HQV}, \downarrow \bar{G} + t_{GLQV}), \\ &\quad \text{earlier } (AdN + t_{AXQX}, \uparrow \bar{E}1 + t_{E1HQZ}, \downarrow E2 + t_{E2LQZ}, \uparrow \bar{G} + t_{GHQZ})) \\ &= (\text{later } (131.5 + 150, 140.5 + 150, 125 + 150), \\ &\quad \text{earlier } (\downarrow E + 2 + 20, \downarrow \bar{E}1 + 126.5, \downarrow \bar{G})) \end{aligned}$$

During a read cycle, the data are required by the 9S12. Thus, to determine the read data required interval, we look in the 9S12 data sheet. For read data required, the worst case is the longest interval. Recall that

$$RDR = \text{Read Data Required} = (\downarrow E - 15, \downarrow E)$$

We must choose the number of cycle stretches to make the read data available interval overlap the read data required interval. We must make = 1 during a read cycle. Thus,

$$\begin{array}{lll} \text{Address} & 131.5 + 150 \leq \downarrow E - 15 & \text{and} \\ \overline{E1} & 140.5 + 150 \leq \downarrow E - 15 & \text{and} \\ E2 & 125 + 150 \leq \downarrow E - 15 & \text{and} \\ & & \downarrow E \geq \downarrow E \end{array}$$

The  $\overline{E1}$  timing ( $305.5 \leq \downarrow E$ ) tells us that 1 extra cycle is needed to stretch the access time to 500 ns. If the read data available did not overlap the read data required, then we would have to

- increase the number of cycle stretches; or
- slow down the 9S12 by increasing the E period; or
- decrease  $t_{E1LQV}$  by spending more money on a faster RAM chip.

The beginning of RDA is determined by  $\downarrow \overline{E1}$  and the end by  $\downarrow E2$

$$\text{Read Data Available} = (\downarrow \overline{E1} + 150, \downarrow E2) = (290.5, 500).$$

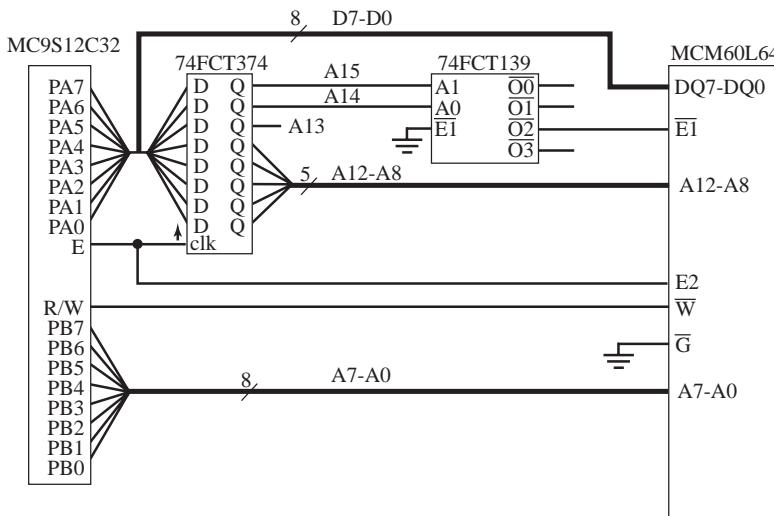
**Common error:** If you do not synchronize driving the data bus on a multiplexed address/data computer such as the MC9S12C32, memory data might be driven onto the bus during the first half of the cycle, and it might conflict with the address being driven at that time.

**Checkpoint 9.6:** How fast would the RAM read access time need to be to run the interface in Figure 9.31 with no cycle stretches?

Data are written into the memory when  $E2 = 1$ ,  $\overline{E1} = 0$  and  $\overline{W} = 0$ . During a write cycle, the data are supplied by the 9S12. For write data available, the worst case is the shortest interval. Recall that

$$\text{WDA} = \text{Write Data Available} = (\frac{1}{2}t_{cyc} + 7, \downarrow E + 2) = (132, \downarrow E + 2)$$

**Figure 9.31**  
Interface between the MC9S12C32 and the 60L64 8K RAM chip.



Since we have chosen to synchronize E2 to the E clock,

$$\text{Write Data Required} = (\downarrow E - 60, \downarrow E)$$

The number of cycle stretches will also affect whether or not the write data available interval overlaps the write data required interval. The worst case delay selects the largest WDR interval

$$132 \leq \downarrow E - 60 \quad \text{and} \quad \downarrow E \leq \downarrow E + 2$$

Thus,

$$192 \leq \downarrow E$$

Therefore, the 1-cycle stretch needed for the read cycle is also sufficient for the write cycle. Although the write cycle could have operated with no stretches, we have to choose 1-stretch so that both read and write timing are satisfied (Program 9.2).

```
void RAM_Init(void){
    MODE = 0xA0;                                // normal expanded narrow mode
    MISC = (MISC&0xF3) | 0x04;                  // 1-cycle stretch on external
    PEAR = 0x0C;                                // enable E, R/W, LSTRB(not needed)
```

### Program 9.2

Software to configure the MC9S12C32 for the external RAM.

**Common error:** If you do not synchronize clocking data into a memory during a write cycle, data might not be properly stored.

**Checkpoint 9.7:** The write-cycle timing does not work for this 9S12C32/MCM60L64 interface if it is synchronized to E1 or W instead of connecting E2 directly to E. Why?

We combine the command and timing aspects of the design to find the Boolean expression that will generate the control signals: E2,  $\overline{E1}$ ,  $\overline{G}$ , and  $\overline{W}$ , as shown in Table 9.13.

**Common error:** A capacitive load occurs both by the physical layout of the PCB as well as with each input pin the output must drive. The 9S12C32 and 74FCT374 timings are specified for capacitive loads of 50 pF. If the actual load is larger than 50 pF, the timing will be significantly slower.

**Table 9.13**  
Combined timing table  
for the MCM60L64  
RAM interface.

E	R/W	A15, A14	RD	WR	E2	$\overline{E1}$	$\overline{G}$	$\overline{W}$
0	0	00 01 or 11	0	0	X	1	X	X
1	0	00 01 or 11	0	0	X	1	X	X
0	1	00 01 or 11	0	0	X	1	X	X
1	1	00 01 or 11	0	0	X	1	X	X
0	0	10	0	0	0	0	X	0
1	0	10	0	1	1	0	X	0
0	1	10	0	0	0	0	X	1
1	1	10	1	0	1	0	0	1

Address not  
on the chip

Write cycle

Read cycle

$$\begin{aligned} E2 &= E \\ \overline{E1} &= \overline{A15} \cdot \overline{A14} \end{aligned}$$

$$\begin{aligned} \overline{G} &= 0 \\ \overline{W} &= R \cdot W \end{aligned}$$

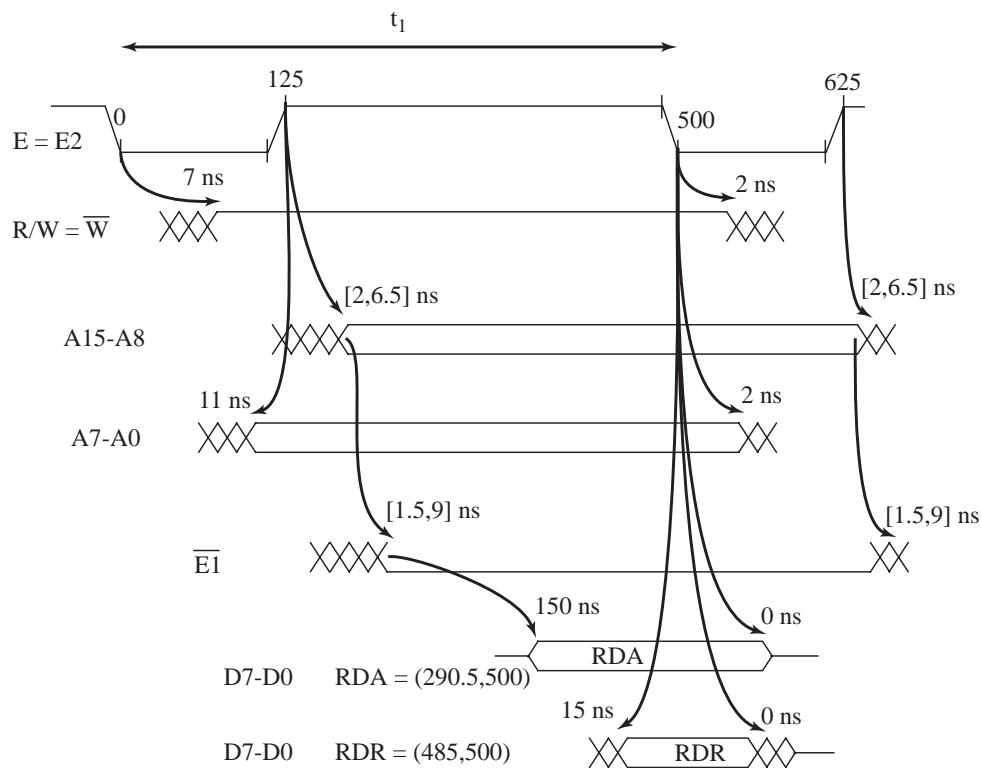
The next step is to build the interface. MODC, MODB, MODA are set to start the system in Special Single-Chip mode.

To verify proper read cycle timing we draw the *Combined Read Cycle Timing Diagram* as shown in Figure 9.32. We need to verify that read data available overlaps read data required before we build the design.

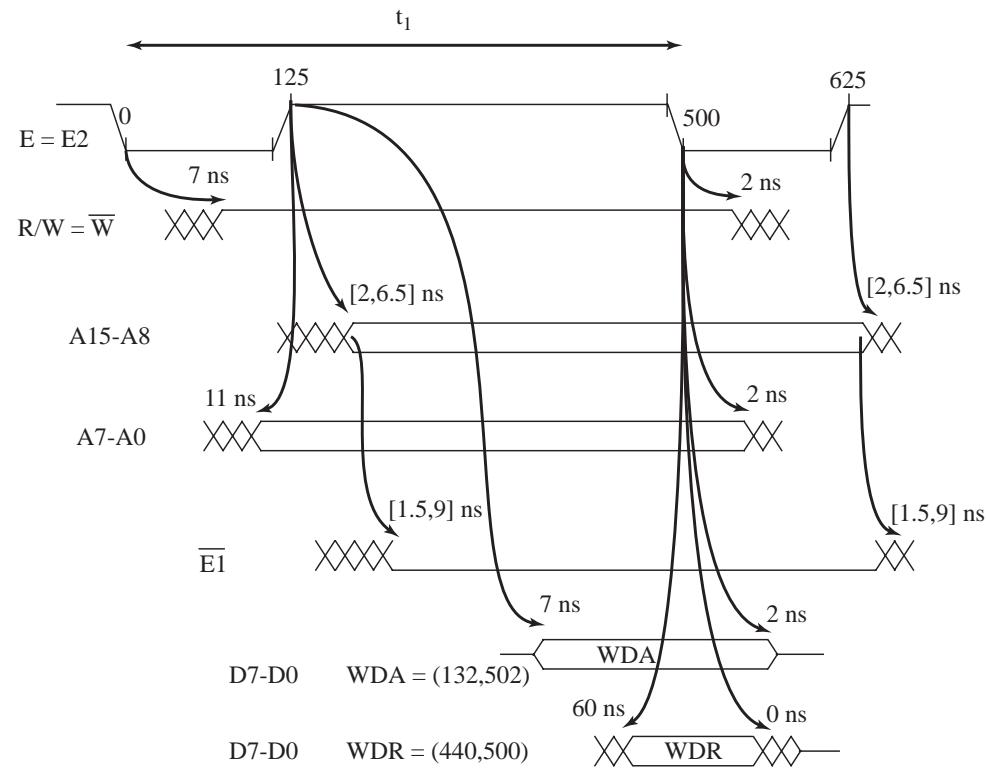
Similarly, to verify proper write cycle timing we draw the *Combined Write Cycle Timing Diagram*, as shown in Figure 9.33. Notice that write data available overlaps write

**Figure 9.32**

Read timing of the MC9S12C32 and the 60L64 8K RAM chip.

**Figure 9.33**

Write timing of the MC9S12C32 and the 60L64 8K RAM chip.



data required. After reset, the MC9S12C32 will be running in Special Single-Chip mode. The compiler generates initialization code to place the RAM at \$3800 to \$3FFF. Program 9.2 will change the mode from Special Single-Chip to Normal Expanded Narrow. Internal visibility is turned off to simplify debugging (IVIS = 0). The PEAR register is set to enable the R/W and E clock.

**Observation:** If we come out of reset in normal single-chip mode, the 6812 will allow one write attempt to the MODE register. Sometimes the compiler or debugger uses up this one change during initialization, before our code runs.

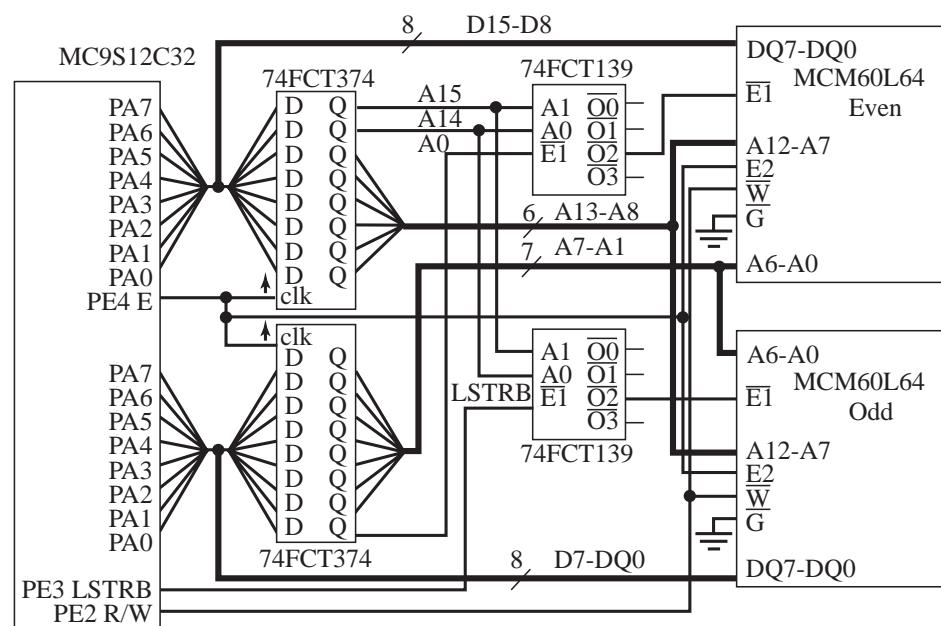
**Checkpoint 9.8:** How many cycle stretches would be needed for the 9S12C32/MCM60L64 interface in Figure 9.31 if the  $t_{cyc}$  were reduced from 250 to 125 ns?

**Example 9.7** Interface two Motorola MCM60L64 8-KiB by 8-bit static RAMs to the microcomputer using the 16-bit-wide data bus. The memory will be placed at \$8000 to \$BFFF.

**Solution** We will use full decoding, placing the RAM at addresses \$8000 to \$BFFF. The MC9S12C32 interface will require this external address decoder, and the positive logic chip select will be  $A_{15} \cdot A_{14}$  (see Figure 9.34). The operation and timing of the MCM60L64 were presented in Example 9.6. The 74FCT139 gates create chip selects that handle the situation where an 8-bit data set is written to this 16-bit RAM, as described in Table 9.10. The timing considerations are identical to the 8-bit interface. If the software performs an 8-bit write to an even address, just the even RAM is activated. If the software performs an 8-bit write to an odd address, just the odd RAM is activated. If the software performs a 16-bit write to an even address, both RAMs are activated. If the software performs a 16-bit write to an odd address, the 9S12 divides the request into two sequential 8-bit writes.

**Figure 9.34**

Interface between the MC9S12C32 and the MCM60L64 RAM.



Program 9.3 enables E, LSTRB, R/W outputs and selects 1-cycle stretch on external devices.

### Program 9.3

Software to configure the mode for the external RAM.

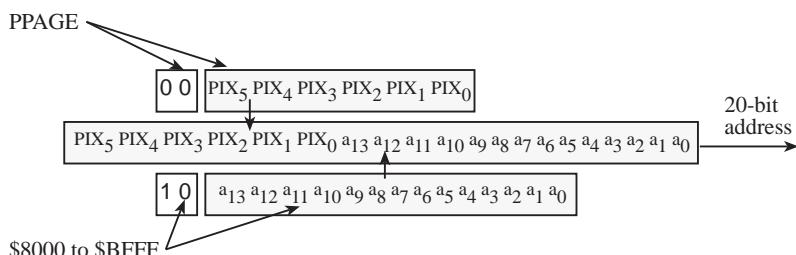
```
void RAM_Init(void){
    MODE = 0xE0;                                // normal expanded wide mode
    MISC = (MISC&0xF3) | 0x04;                  // 1-cycle stretch on external
    PEAR = 0x0C;                                // enable E, R/W, LSTRB
```

## 9.6 9S12 Paged Memory

16-bit pointers can access only up to 64 KiB of memory. The 9S12 uses a paged memory system to access memory beyond this 64 KiB barrier. On most of the 9S12 microcontrollers, the extended address contains 20 bits and thus can access up to 1 MiB of memory. The paged memory system is organized into a maximum of 64 pages with a fixed page size of 16 KiB. The software must first write the page number into **PPAGE**, which is an 8-bit register located at \$0030 (only the bottom six bits are used). On the 9S12, addresses in the \$8000 to \$BFFF window invoke the paged memory system. The top six bits of the 20-bit extended address are retrieved from the **PPAGE** register, and the bottom 14 bits come from the regular 16-bit address, as shown in Figure 9.35. In particular, when the software accesses any address in the \$8000 to \$BFFF window, the bottom six bits of **PPAGE** are concatenated to the bottom 14 bits of the window address to create the 20-bit extended address used to access memory. This logical-to-physical address translation occurs automatically whenever an address in the \$8000 to \$BFFF window is accessed.

**Figure 9.35**

The address is comprised of two components.



On the 9S12DP512, to access all of the 512 KiB flash, we must use this paging mechanism. On the 9S12DP512, there are only 32 pages needed for the 512 KiB flash EEPROM. In particular, it utilizes page numbers \$20 through \$3F. Page \$3E is actually the same as regular EEPROM at \$4000 to \$7FFF, and page \$3F is the same as EEPROM at \$C000 to \$FFFF.

**Observation:** If the software sets and leaves **PPAGE** at \$20 (actually any constant value from \$20 to \$3D), then the EEPROM behaves like a simple 48 KiB memory from \$4000 to \$FFFF.

We will present two applications of paged memory. In this first application, the flash EEPROM on the 9S12DP512 will contain a large data buffer. We can either have the compiler initialize the data at download time or use the programs in Section 9.7 to write the data dynamically. Because these data are located in EEPROM, we will consider them as constant, and provide a function to access the data. The flash EEPROM will be accessed using a two-field address. The first field is the page number \$20 to \$3D, and

the second field will be the offset address \$8000 to \$BFFF. In assembly, the page field is passed in Register B and the offset address is passed in Register X. In the C version, the entire two-field address is packed into one 32-bit value. Bits 31 through 24 will be zero. Bits 23 through 14 will contain the page numbers \$20 to \$3D, and bits 15 through 0 will contain the offset \$8000 to \$BFFF. We need to be careful to realize the 16 KiB space from \$4000 to \$7FFF also exists as page \$3E, and the 16 KiB space from \$C000 to \$FFFF also exists as page \$3F. On the 9S12DP512, the data buffer exists as 30 pages from \$20 to \$3D. The subroutine, shown as Program 9.4, first sets the PPAGE register to select the correct page and then reads from the \$8000 to \$BFFF window to retrieve the specified data.

<pre>;Read byte from buffer in Flash ;Input: B is the page number \$20 to \$3D ;        X is offset address \$8000 to \$BFFF ;Output: A is data Flash_ReadByte     stab PPAGE     ldaa 0,x ;A=8-bit data from buffer     rts ;Input B is the page number \$20 to \$3D ;        X is offset address \$8000 to \$BFFF ;Output Y is 16-bit data Flash_ReadWord     stab PPAGE     ldy 0,x ;Y=16-bit data from flash     rts</pre>	<pre>typedef unsigned char byte; typedef unsigned short word; typedef unsigned long dword; // Read byte from buffer in Flash // msword of addr=page, lsword=\$8000-\$BFFF byte Flash_ReadByte(dword addr){     PPAGE = (byte)(addr&gt;&gt;16);     return *( byte *) (addr&amp;0xFFFF); } // Read 16-bit data from flash word Flash_ReadWord(dword addr){     PPAGE = (byte)(addr&gt;&gt;16);     return *(word *) (addr&amp;0xFFFF); }</pre>
--	---

#### Program 9.4

Programs to read data from flash EEPROM using paged memory.

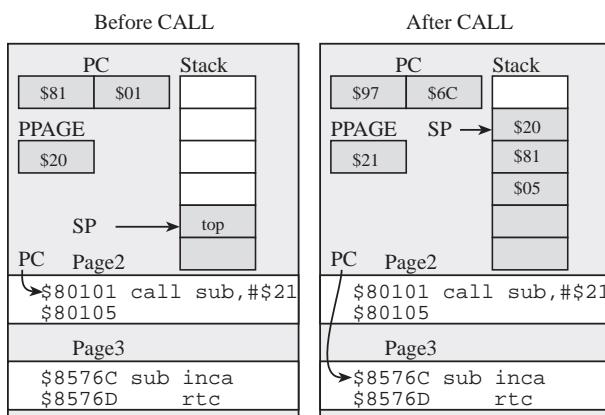
The second application implements a system with a code size of more than 48 KiB. Most compilers will generate this code automatically, so it is included here only as an academic exercise to illustrate how the paging works for large software systems. When using Metrowerks CodeWarrior, we select the *banked memory model* when creating a new project. On the 9S12, we will partition the code into separate 16 KiB pieces. The system will be most efficient if the partitioning is done according to access probability. In other words, if Module A frequently calls Module B, then A and B will be placed into the same 16 KiB page. We will place the most frequently used code and the starting location into the pages \$4000 to \$7FFF and \$C000 to \$FFFF. Accessing these locations is simple and uses standard 16-bit pointers. We place the remaining code into paged memory. Subroutine calls within the same page can utilize the standard `bsr` and `jsr` instructions. To call a subroutine located in a different page, the `call` instruction is used. Figure 9.36 shows the stack before and after the `call` instruction is executed on the 9S12DP512. The `call` instruction pushes the old **PPAGE** and PC values on the stack and then loads **PPAGE** and PC with the address of the subroutine.

When op codes are fetched from the \$8000 to \$BFFF window, the 6-bit **PPAGE** is combined with the lower 14 bits of the PC to form a 20-bit address. The translation occurs automatically in hardware. Consider the case where the **PPAGE** register equals \$20, and the PC is \$8101 (left part of Figure 9.36).

```
PPAGE = $20 = 00100000
PC = $8101 = 1000000100000001
PPAGE + Lower 14 bits of PC = 100000+00000100000001 = $80101
```

**Figure 9.36**

The call instruction is used to call a subroutine in paged memory.



After the call instruction, **PPAGE** register equals \$21, and the PC is \$976C (right picture of Figure 9.36).

$$\text{PPAGE} = \$21 = 00100001$$

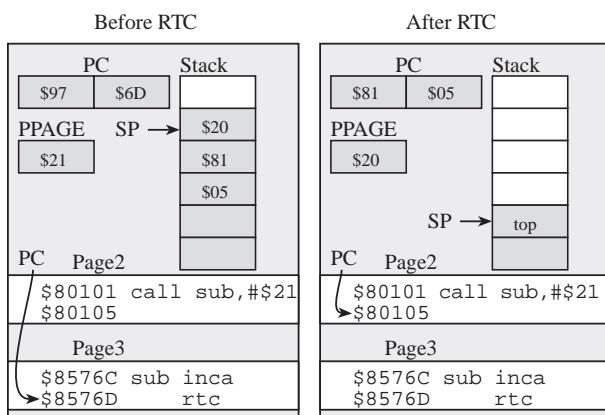
$$\text{PC} = \$976C = 1001011101101100$$

$$\text{PPAGE} + \text{Lower 14 bits of PC} = 100001+01011101101100 = \$8576C$$

The **rtc** instruction will return to the program that called the subroutine. Both the **PPAGE** and PC values are pulled off the stack. Figure 9.37 shows the stack before and after execution of the **rtc** instruction.

**Figure 9.37**

The **rtc** instruction is used to return from a subroutine in paged memory.



Figures 9.34, 9.35, and 9.36 and Programs 9.5, 9.6, and 9.7 illustrate the use of **call** and **rtc** to create a paged memory system on the 9S12. Program 9.5 will be programmed into main EEPROM. Program 9.6 will be programmed into external page \$21. Program 9.7 will be programmed into external page \$22.

### Program 9.5

Main memory programs for this paged memory system.

```

func1 equ 0          ; relative offset in paged memory
func2 equ 3          ; relative offset in paged memory
org $4000           ; main EEPROM memory
main lds #$4000     ; stack in main RAM
        clra
loop   call func1,#$21    ; call function 1 in page $21 (add 1)
        call func1,#$22    ; call function 1 in page $22 (add 2)
        call func2,#$21    ; call function 2 in page $21 (add 3)
        call func2,#$22    ; call function 2 in page $22 (add 4)
        bra loop

```

**Program 9.6**

Page \$21 programs for this paged memory system.

```
org $0000 ; page $21 external memory
lbra fun1 ; link to actual function
lbra fun2 ; link to actual function
fun1 adda #1
    rtc
fun2 adda #2
    rtc
```

**Program 9.7**

Page \$22 programs for this paged memory system.

```
org $0000 ; page $22 external memory
lbra fun1 ; link to actual function
lbra fun2 ; link to actual function
fun1 adda #3
    rtc
fun2 adda #4
    rtc
```

## 9.7 Programming Flash EEPROM

Many 9S12 microcontrollers have large flash EEPROMs. When downloading our object code, the debugger will first erase the EEPROM and then program our information into the EEPROM. It is nonvolatile, which means our information still exists if power is removed and later restored. It would be very useful if our software could write into this EEPROM, saving data at run time. We could log measurement data; we could record debugging or performance data; or we could remember decisions we made or conditions we encountered for the entire life of the device. In other words, these data would not be lost if power were removed or if a hardware reset were to occur.

In Section 9.6, we learned that the flash EEPROM is organized logically into 16 KiB pages. Physically, however, flash EEPROM is organized into sectors and blocks. The sizes of the sector and block vary according to the overall size of the flash EEPROM (see Table 9.14). As for most EEPROMs, there is an erase function that sets all bits of the data to one, and there is a program function that can set individual bits in the data to zero. Normally, we erase the EEPROM and then program individual locations to place zeros where needed. There is a sector erase and a block erase command.

**Table 9.14**

The sector and block sizes for some 9S12 microcontrollers.

Flash Size (kilobytes)	Page Numbers	Sector Size (bytes)	Pages per Block	Block Size (kilobytes)	Number of Blocks
DP512/DG512	\$20 to \$3F	1024	8	128	4
DP256/DG256	\$30 to \$3F	512	4	64	4
DP128/DG128	\$38 to \$3F	512	4	64	2
DP64/DG64	\$3C to \$3F	512	4	64	1
DP32/DG32	\$3E to \$3F	512	2	32	1
E128	\$38 to \$3F	1024	8	128	1
C128	\$3E to \$3F	1024	8	128	1
C64	\$3C to \$3F	1024	4	64	1
C32	\$3E to \$3F	512	2	32	1

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0030	0	0	PIX5	PIX4	PIX3	PIX2	PIX1	PIX0	PPAGE
\$0100	FDIVLD	PRDIV8	FDIV5	FDIV4	FDIV3	FDIV2	FDIV1	FDIV0	FCLKDIV
\$0103	CBEIE	CCIE	KEYACC	0	0	0	BKSEL1	BKSEL0	FCNFG
\$0105	CBEIF	CCIF	PVIOL	ACCERR	0	BLANK	0	0	FSTAT
\$0106	0	CMDB6	CMDB5	0	0	CMDB2	0	CMDB0	FCMD

**Table 9.15**  
9S12 flash EEPROM ports.

Table 9.15 shows the ports needed to program the flash EEPROM. The only initialization step we need to perform is to set the flash clock to a frequency between 150 kHz and 200 kHz. The **FCLKDIV** register can be set only once, and if the **FDIVLD** bit is set, the register has only been written and can not be changed until the next reset. If it is too fast, erasing and programming will not work. If the clock is too slow, the flash will be damaged. The flash clock is derived from the oscillator clock, not the E clock. If the 9S12 has a 16 MHz crystal, it means the oscillator clock frequency  $F_O$  is 16 MHz. Let  $n$  be the 6-bit number comprised of the bottom six bits of the **FCLKDIV** register. If the **PRDIV8** bit is set, the flash clock frequency is  $F_F = F_O/8/(n + 1)$ . If the **PRDIV8** bit is clear, the flash clock frequency is  $F_F = F_O/(n + 1)$ . One option for a 9S12 with a 16 MHz crystal running at 24 MHz is to set **FCLKDIV** to \$4A, making  $F_F$  equal to 182 kHz. Another option is to set it to \$49 and run the flash clock at the maximum of 200 kHz.

We write to the **FCMD** register to initiate a command. Writing a \$20 will program a 16-bit word, \$40 will erase a sector, and \$41 will erase the entire block. If we are using the EEPROM as dynamic logging of data, we can assume the debugger erased the flash when the program was downloaded. However, if we wish to change any bits from 0 to 1, we need to erase the memory. We can either erase an entire sector or erase an entire block. We can not erase individual bytes or words. The other tricky problem is that we cannot read the flash block during an erase or program operation to that same block. In other words, we cannot be executing software out of a flash block while locations within that same block are being erased or programmed. For a 9S12 with just one block (see Table 9.14), we have to copy our programming software from ROM into RAM, jump into the RAM version of the code, program the flash, and then jump back to ROM when the flash command is complete. For a 9S12 with multiple blocks, we could place our software in one block and use the other blocks as data storage. The example code in this section will allow us to erase and program three out of the four blocks on a 9S12DP512.

We will use the same two-field address described in Section 9.6. Program 9.8 contains private functions for the system. `FlashWrite` will take a 16-bit data set and write it to the location specified by `addr`. The page field will be written into **PPAGE**, and the data will be stored into \$8000 to \$BFFF as specified by the offset field of `addr`. There are four blocks on the 9S12DP512, and each block has a separate status register **FSTAT**. No program or erase function can occur if any of the status bits shows an error, so the helper function `ClearErrorFlags` will explicitly clear the error flags **PVIOL** and **ACCERR** in all four status registers. The bottom two bits of **FCNFG** specify which block will be accessed. The helper function `SelectBlock` converts the page number into a block number. Pages \$20 to \$27 are block 0, pages \$28 to \$2F are block 1, and pages \$30 to \$37 are block 2. In this system, we will not program block 3 (pages \$38 to \$3F), because our software exists in this block. Programming flash incorrectly can result in a serious crash (e.g., we can erase software.) The helper function `AddrValid` checks to make sure the address parameter correctly accesses one of three allowed blocks.

**Program 9.8**

Helper functions to access flash memory.

```

typedef unsigned char byte;
typedef unsigned short word;
typedef unsigned long dword;
// write 16-bit data to flash
static void FlashWrite(dword addr, word data){
    PPAGE = (byte)(addr>>16);
    *(word*)(addr&0xFFFF) = data;
}
static void ClearErrorFlags(void){ byte i;
    for(i = 0; i <= 3; i++) {
        FCNFG_BKSEL = i; // Select block
        FSTAT = 0x30; // Clear PVIOL & ACCERR
    }
}
static void SelectBlock(dword addr){
byte page = (byte)(addr >> 16);
    FCNFG_BKSEL = (0x3F - page)>>3;
}
static word AddrValid(dword addr){
byte page = (byte)(addr >> 16);
word offset = (word)addr;

if((offset<0x8000) || (offset>0xBFFF) || (page<0x20) || (page>0x37))
    return 1;
return 0;
}

```

Program 9.9 shows the public functions for the system. The `Flash_Init` function sets the flash clock to 200 kHz. On the 9S12DP512, the sector size is 1024 bytes. So, the `Flash_EraseSector` function will erase 1024 bytes as specified by the `addr` parameter. The bottom ten bits of `addr` are ignored. Erasing and programming involve delicate timing between steps, so interrupts are disabled during these functions. The `Flash_ProgramWord` function will program one 16-bit word into the location specified by the `addr` parameter. The flash address must be aligned on a word boundary. The `Flash_ProgramArray` function will program `count` bytes into the flash at `addr`. All functions begin by clearing the error flags, setting the block number, and checking to make sure the flash is ready. When the **CBEIF** flag in the **FSTAT** register is 1, the flash is ready to accept a new command. Since all functions will wait until completion, this bit should be 1 at the start of each new operation. The functions follow the same six steps:

1. Write data to the flash.
2. Set the **FCMD** to the desired operation.
3. Clear the **CBEIF** flag in the **FSTAT** register to start the command.
4. Check to see if a protection violation (**PVIOL**) or access error (**ACCERR**) occurred.
5. Wait for the command buffer to be empty (**CBEIF**).
6. Wait for the command to be complete (**CCIF**).

The data we write during an erase operation does not matter, but in each case the `addr` parameter specifies the block, sector, or word location affected by the command. To erase an entire block (128 kibibytes), we change the 0x40 to 0x41 in the erase section function. When we program a 16-bit word, the high-voltage charge pumps are activated, the word is programmed, and then the charge pumps are shut off. When programming multiple words, we can chain the operations together so the charge pump is turned on once, and all words are programmed; then the charge pump turns off. The `Flash_ProgramArray` function runs in 55% of the equivalent time to execute `Flash_ProgramWord` multiple times.

**Program 9.9**

Public functions to erase and program flash memory.

```

void Flash_Init(void){
    FCLKDIV = 0x49; // flash clock = 200 kHz
}
word Flash_EraseSector(dword addr){
    if(AddrValid(addr)) return 1;
    DisableInterrupts; // Enter critical section
    ClearErrorFlags(); // Clear all flags
    SelectBlock(addr);
    if(FSTAT_CBEIF==0){
        EnableInterrupts;
        return 1; // command buffer full
    }
    FlashWrite(addr,0x10); // Write any word
    FCMD = 0x40; // erase one sector
    FSTAT = 0x80; // Clear command buffer
    if((FSTAT_PVIOL==1) | (FSTAT_ACCERR==1)){
        EnableInterrupts;
        return 2; // PVIOL or ACCERR
    }
    while(FSTAT_CBEIF==0); // wait to buffer empty
    while(FSTAT_CCIF==0); // Wait to command complete
    EnableInterrupts;
    return 0; // OK
}
word Flash_ProgramWord(dword addr, word data){
    if(AddrValid(addr)) return 1;
    if(addr & 1) return 3; // unaligned address
    if(FSTAT_CCIF==0) return 4; // busy
    DisableInterrupts; // Enter critical section
    ClearErrorFlags(); // Clear all flags
    SelectBlock(addr);
    if(FSTAT_CBEIF==0){
        EnableInterrupts;
        return 1; // command buffer full
    }
    FlashWrite(addr,data); // Write word
    FCMD = 0x20; // program word
    FSTAT = 0x80; // Clear command buffer
    if((FSTAT_PVIOL==1) | (FSTAT_ACCERR==1)){
        EnableInterrupts;
        return 2; // PVIOL or ACCERR
    }
    while(FSTAT_CBEIF==0); // wait to buffer empty
    while(FSTAT_CCIF==0); // Wait to command complete
    EnableInterrupts;
    if(Flash_ReadWord(addr) != data) return 5;
    return 0; // OK
}
word Flash_ProgramArray(byte *source, dword addr, word count){
word i,data;
    if(AddrValid(addr)) return 1;
    if((addr&1)|(count&1)) return 3; // unaligned, odd count
    if(FSTAT_CCIF==0) return 4; // busy
    DisableInterrupts; // Enter critical section
}

```

*continued on p. 496*

**Program 9.9 (cont'd)**

Public functions to erase and program flash memory.

*continued from p. 495*

```

ClearErrorFlags();      // Clear all flags
SelectBlock(addr);
for(i = 0; i < count; i += 2) {
    if(i != 0) {                      // Not in first cycle
        while (FSTAT_CCIF == 0); // Wait for previous
    }
    FlashWrite(addr+i,*(word *)(source+i)); // Write
    FCMD = 0x20;                  // Word program command
    FSTAT = 0x80;                  // Clear command buffer
    if((FSTAT_PVIOL==1) || (FSTAT_ACCERR==1)){
        EnableInterrupts;
        return 2;                  // PVIOL or ACCERR
    }
} /* loop for all array bytes */
while(FSTAT_CBEIF==0); // wait to buffer empty
while(FSTAT_CCIF==0); // Wait to command complete
EnableInterrupts;
for(i = 0; i < count; i += 2){
    data = Flash_ReadWord(addr+i);
    if(data != *(word *)(source+i))
        return 5;
}
return 0;                  // OK
}

```

## 9.8 Dynamic RAM (DRAM)

Because of their high density and low cost, DRAMs are used for situations requiring a large amount of storage. Consequently, embedded systems will almost exclusively employ SRAMs for their RAM needs. As the complexity of embedded system software increases, the need for RAM space will grow as well. At some point, DRAMs may become cost-effective for embedded systems. The purpose of this section is to introduce the basic principles involved in DRAMs. The information in a DRAM cell is saved as a charge on a capacitor. The information in a SRAM cell is saved as the state of a set-reset flip-flop. This reduction in size comes at the expense of needing to refresh the capacitor charges. This refresh activity adds a fixed cost. Table 9.16 compares the two memory types.

**Table 9.16**

Comparison between DRAM and RAM.

DRAMs	SRAMs
High density	Low density
One transistor, one capacitor/bit	Three–four transistors/bit
Slower	Faster
High fixed cost (refresh)	Low fixed cost (address decoder)
Low incremental cost	Higher incremental cost
Address multiplexing	Direct addressing

The higher fixed cost and lower incremental cost of DRAM versus SRAM means for small memory applications (like embedded systems) the SRAMs will be cost-effective. At some point as the memory size gets large enough, the DRAMs will become cost-effective.

To interface a DRAM, certain support functions are required. Because the DRAM devices have so many address lines, they employ a technique called address multiplexing.

This means the computer first gives half the address and toggles a row address strobe, RAS. Then the computer gives the second half of the address and issues a column address strobe, CAS. A refresh cycle occurs when a RAS is issued without a following CAS. This will refresh the entire row. The interface logic must handle the refresh process controlling RAS and CAS. In addition, the interface must arbitrate between normal read/write and refresh cycles. This arbitration is usually handled using the partially asynchronous bus protocol shown in Figure 9.13. Because the 9S12 does not support dynamic bus cycle stretching, DRAM interfacing is difficult.

## 9.9 Exercises

- 9.1** For each term, give a definition in 32 words or less.
- |                      |                                  |                          |
|----------------------|----------------------------------|--------------------------|
| a) Memory mapped I/O | f) Partially asynchronous bus    | j) Synchronized signal   |
| b) Isolated I/O      | g) Fully asynchronous bus        | k) Unsynchronized signal |
| c) Command signal    | h) Full address decoding         | l) Expanded mode         |
| d) Timing signal     | i) Minimal-cost address decoding | m) Cycle stretching      |
| e) Synchronous bus   |                                  | n) Aligned address       |
- 9.2** For each term, give a definition in 32 words or less. Give the general timing requirements for a memory interface using these four terms.
- |                        |                         |
|------------------------|-------------------------|
| a) Read data available | c) Write data available |
| b) Read data required  | d) Write data required  |
- 9.3** Without the PLL, what is the relationship between the crystal frequency and the E clock frequency?
- 9.4** Briefly explain the difference between these three modes: (1) single chip, (2) expanded narrow, and (3) expanded wide. Give one sentence for each mode. In particular, explain the effect on Ports A, B, E, and K.
- 9.5** Assuming a crystal frequency of 16 MHz, write PPL initialization software to make the 9S12 run at 25 MHz.
- 9.6** Assuming a crystal frequency of 16 MHz, write PPL initialization software to make the 9S12 run at 1 MHz.
- 9.7** What is capacitive loading, and how does it apply to memory interfacing?
- 9.8** Design a minimal-cost, positive-logic address decoder for a device at addresses \$7000 to \$7FFF, assuming the other devices are \$0000 to \$0FFF, \$6000, and \$C000 to \$FFFF. Show the design procedure, Karnaugh map, digital equation, and digital circuit.
- 9.9** Design a minimal-cost, negative-logic address decoder for a device at addresses \$6000 to \$7FFF, assuming the other devices are \$0000 to \$0FFF, \$4000, and \$F000 to \$FFFF. Show the design procedure, Karnaugh map, digital equation, and digital circuit.
- 9.10** Design a fully decoded, positive-logic address decoder for a device at addresses \$5000 to \$6FFF. Show the design procedure, Karnaugh map, digital equation, and digital circuit.
- 9.11** Design a fully decoded, negative-logic address decoder for a device at addresses \$8000 to \$AFFF. Show the design procedure, Karnaugh map, digital equation, and digital circuit.
- 9.12** Calculate the entries in Table 9.9 if the E clock frequency were 8 MHz rather than 4 MHz.
- 9.13** Write software like Program 9.4 to read 32-bit data from flash EEPROM.
- For Exercises D9.14 through D9.18, use a search engine such as <http://octopart.com/> to find a data sheet for the memory. The delay through 74FCT logic is [0,2]. The delay through 74FCT138 and 74 HC139 logic is [1,5,9]. Choose the number of cycles to stretch. Run in expanded narrow mode. For each exercise, do the following five parts:
- Show the minimal-cost address decoder. Choose to build either a positive-logic or negative-logic decoder, depending on which would be most appropriate.

- b) Create a combined table, and develop equations the memory control signals.
- c) Show the digital interface between the 9S12 and the memory. Label all chip numbers but not pin numbers.
- d) Draw a Read Cycle Timing Diagram; use arrows to show causal relationships. Clearly label Read Data Available, showing that it does overlap Read Data Required.
- e) Draw a Write Cycle Timing Diagram; use arrows to show causal relationships. Clearly label Write Data Available, showing that it does overlap Write Data Required.

**D9.14** The objective is to interface an 8-KiB by 8-bit Cypress CY6264-55 RAM to the 9S12 at \$8000 to \$9FFF. The existing devices are \$0000 to \$0FFF, \$4000 to \$7FFF, and \$C000 to \$FFFF.

**D9.15** The objective is to interface an 8-KiB by 8-bit Cypress CY6264-70 RAM to the 9S12 at \$8000 to \$9FFF. The existing devices are \$0000 to \$0FFF, \$4000 to \$7FFF, and \$C000 to \$FFFF.

**D9.16** The objective is to interface two 8-KiB by 8-bit Sharp LH5164A-10 RAMs to the 9S12 at \$8000 to \$BFFF. The existing devices are \$0000 to \$0FFF, \$4000 to \$7FFF, and \$C000 to \$FFFF.

**D9.17** The objective is to interface a 32-KiB by 8-bit Cypress CY62256NLL-70 to the 9S12 at \$4000 to \$BFFF. The existing devices are \$0000 to \$0FFF and \$C000 to \$FFFF.

**D9.18** The objective is to interface a 32-KiB by 8-bit IDT IDT71256SA25 RAM to the 9S12 at \$4000 to \$BFFF. The existing devices are \$0000 to \$0FFF and \$C000 to \$FFFF.

## 9.10 Lab Assignments

**Lab 9.1** The overall objective of this lab is to interface a RAM and design a memory tester. The first step is to design, build, and test an external RAM module. During the testing phase, using a multichannel scope or a logic analyzer, you should draw the actual read and write timing diagrams, and compare them to the theoretical timing diagrams derived from the data sheets. The second part of the lab is to design a memory test program that detects and classifies the two types of errors: address bit not connected and data bit not connected. You can evaluate your software using a DIP socket with some of the pins removed. Insert the faulty DIP socket between the microcontroller circuit and the RAM. The output of your program should be very specific, e.g., "Pins A4 , A2 , D7 , D5 , D1 are not connected ." Do not attempt to simulate errors such as pins stuck together, pins stuck high, or pins stuck low, because it could cause damage to your system.

**Lab 9.2** The overall goal is to develop a solid-state disk. Your system will be able to create files, append data to the end of a file, print out the entire contents of a file, and delete files. In addition, your system will be able to list the names and sizes of the available files. Basically, you will interface a large RAM, and then write a series of software functions that make it appear as a disk. (See Lab 10.3.) On the 9S12, you should use the paging mechanism. If you use paging, you should select a disk block size different from the memory page size. You can use a 3V battery and two low-voltage drop diodes to create a nonvolatile RAM.

**Lab 9.3** The overall objective of this lab is to interface an external EEPROM (like the 28C64) to the microcontroller and develop mechanisms to program the device. The first step is to design, build, and test an external EEPROM module. During the testing phase, using a multichannel scope or a logic analyzer, you should draw the actual read timing diagram, and compare it to the theoretical timing diagram derived from the data sheets. The second part of the lab is to design a software system that allows you to program the EEPROM directly from the microcontroller.

**Lab 9.4** The overall objective is to develop a set of functions to program the flash EEPROM of a 9S12C32, which has only one block. You will not be able to do a block erase, because that would erase all of EEPROM, including your program. The first step is to partition the EEPROM into code and data space. For example, you could place all your software in \$C000 to \$FFFF and use the EEPROM from \$4000 to \$7FFF as data storage. In summary, you will redesign the functions in Programs 9.4, 9.8, and 9.9 so they work on a 9S12C32 for programming and erasing the flash at \$4000 to \$7FFF. You will have to develop relocatable assembly subroutines and copy them into RAM, because you will not be able to execute software out of \$C000 to \$FFFF while you are erasing or

programming \$4000 to \$7FFF. Use these functions to implement a data logger. For example, when your software is downloaded, the memory from \$4000 to \$7FFF is erased. Every time the user calls `Flash_Log(Buffer[ ])`, 256 bytes from `Buffer` are saved in the flash. This routine returns 0 if successful and nonzero if the flash is full or if there was a flash memory error. The data are never destroyed until your program is downloaded again. Write another function `Flash_Dump` that outputs all stored data via the serial port. Develop a first main program to test the low-level functions, and develop a second main program to test the data logging features.

# 10 High-Speed I/O Interfacing

## Chapter 10 objectives are to:

- ❖ Define the terms bandwidth, latency, and priority
- ❖ Introduce the concept of DMA synchronization
- ❖ Discuss the alternatives of hardware FIFOs, dual port memory, and bank-switch memory
- ❖ Interface a secure digital card (SDC)
- ❖ Present high-bandwidth/low-latency applications
- ❖ Introduce the basics of file systems

**E**mbedded system designers will not need DMA to solve most of their problems. But like the last chapter, there are two motivations for this chapter. The first motivation is basic knowledge, and the second is solving specific high-performance applications. DMA is an important yet complicated interfacing process. One of the advantages of learning DMA on a simple device like a graphics controller is that it maintains all the fundamental concepts without most of the complexities found in larger computer systems. As the performance requirements of our embedded system grow, there comes a point when the simple methods of I/O interfacing are not adequate. This chapter introduces a number of techniques that produce high bandwidth and low latency.

### 10.1 The Need for Speed

Bandwidth, latency, and priority are quantitative parameters we use to evaluate the performance of an I/O interface. The basic function of an input interface is to transfer information about the external environment into the computer. In a similar way, the basic function of an output interface is to transfer information from the computer to the external environment. The bandwidth is the number of information bytes transferred per second. The bandwidth can be expressed as a maximum or peak that involves short bursts of I/O communication. On the other hand, the overall performance can be represented as the average bandwidth. The latency of the hardware/software is the response time of the interface. It is measured in different ways, depending on the situation. For an input device, the *interface latency* is the time from when new input is available to when the data are transferred into memory. We can also define *device latency* as the response time of the external I/O device. For example, if we request that a certain sector be read from a disk, then the device latency is the time it takes to find the correct track and spin the disk (seek) so that the proper sector is positioned under the read head. For an output device, the interface latency is the time from when the output device is idle to when the interface writes new data. A *real-time* system is one that

can guarantee a worst-case interface latency. Table 10.1 illustrates specific ways to calculate latency. In each case, however, latency is the time from when the need arises to when the need is satisfied.

**Table 10.1**

Interface latency is a measure of the response time of the computer to a hardware event.

The Time a Need Arises	The Time the Need is Satisfied
New input is available	Input data are read
New input is available	Input data are processed
Output device is idle	New output data are written
Sample time occurs	ADC is triggered, input data
Periodic time occurs	Output data, DAC is triggered
Control point occurs	Control system executed

If we consider the busy/done I/O states introduced in Chapters 3 and 4, the interface latency is the time from the *busy to done* state transition to the time of the *done to busy* state transition. Sometimes we are interested in the worst-case (maximum) latency and sometimes in the average. If we can put an upper bound on the latency, then we define the system as real time. A number of applications involve performing I/O functions on a fixed-interval basis. In a data acquisition system, the ADC is triggered (a new sample is requested) at the desired sampling rate.

**Checkpoint 10.1:** What is the difference between bandwidth and latency?

## 10.2 High-Speed I/O Applications

Before introducing the various solutions to a high-speed I/O interface, we will begin by presenting some typical applications.

### 10.2.1 Mass Storage

The first application is mass storage, including floppy disk, hard disk, tape, and CD-ROM. Writing data to disk with these systems involves:

1. Establishing the physical location to write (positioning the record head at the proper block, sector, track, cylinder, etc.)
2. Specifying the block size
3. Waiting for the physical location to arrive under the record head
4. Transmitting the data

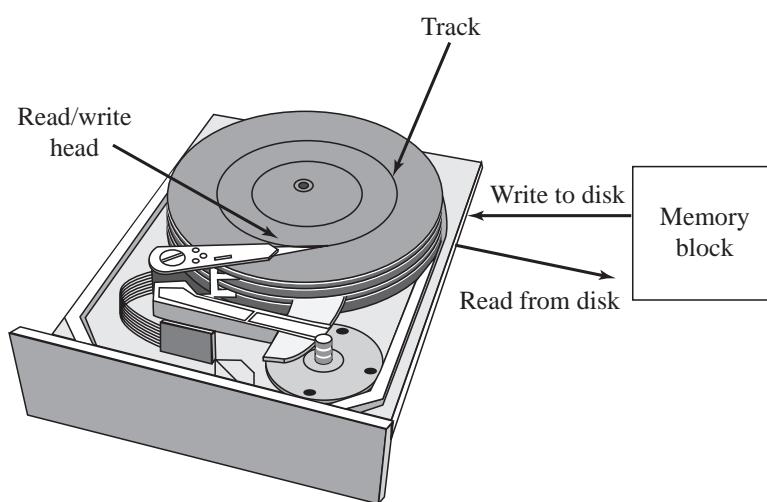
Reading data from disk with these systems is similar and involves:

1. Establishing the physical location to read (positioning the read head at the proper block, sector, track, cylinder, etc.)
2. Specifying the block size
3. Waiting for the physical location to arrive under the read head
4. Receiving the data

Under most situations the size of the data block transferred is fixed. The bandwidth depends on the rotation speed of the disk and the information density on the medium (Figure 10.1). A typical hard drive can sustain about 10 to 40 Mbytes/s. The time to locate the physical location is called the *seek time*. Although seek time has a significant impact on the disk performance, it does not affect the latency or bandwidth parameters. A 32X CD-ROM has a peak bandwidth of 4.8 Mbytes/s. There is a wide range of disk speeds, but it is important to note that for most situations, the disk bandwidth will be

**Figure 10.1**

Data to/from the read/write head of a hard drive has a high bandwidth.



less than the computer bus bandwidth but greater than the maximum bandwidth that a software-controlled interface can achieve. If the disk interface is not buffered, then the interface must respond to each data byte at a rate as fast as the peak disk bandwidth. For example, in a disk read, once the data become available, the interface must capture and store them in memory before the next data become available. If we do not meet the response time requirement in the disk interface, the rotation speed will have to be reduced. Notice that because of the seek time (time for the physical location to arrive under the head), the average and peak bandwidth will be quite different. Also notice that without buffering, the maximum interface latency will be inversely related to the peak bandwidth.

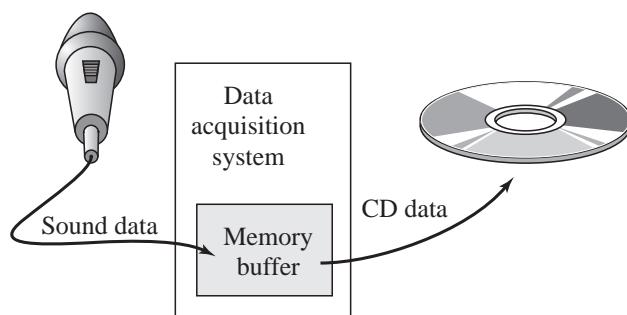
**Checkpoint 10.2:** What happens if we are reading data off a hard drive and it doesn't satisfy the latency requirement? In other words, the read data is ready, but we do not capture it in time.

### 10.2.2 High-Speed Data Acquisition

Examples of high-speed data acquisition are CD-quality sound recording (16-bit, two-channel, 44 kHz), real-time digital image recording, and digital scopes (8-bit 200 MHz). Sound recording actually has two high-speed data channels: one for recording into memory and a second for storing the memory data on hard disk or CD (Figure 10.2). A spectrum analyzer combines the high-speed data acquisition of a digital scope with the Discrete Fourier Transform (DFT) to visualize the collected data in the frequency domain. In the context of this chapter, we will define a high-speed data acquisition as one that samples faster than a software-controlled interface would allow.

**Figure 10.2**

Sound data from the microphone are stored in memory, processed, then saved on CD.



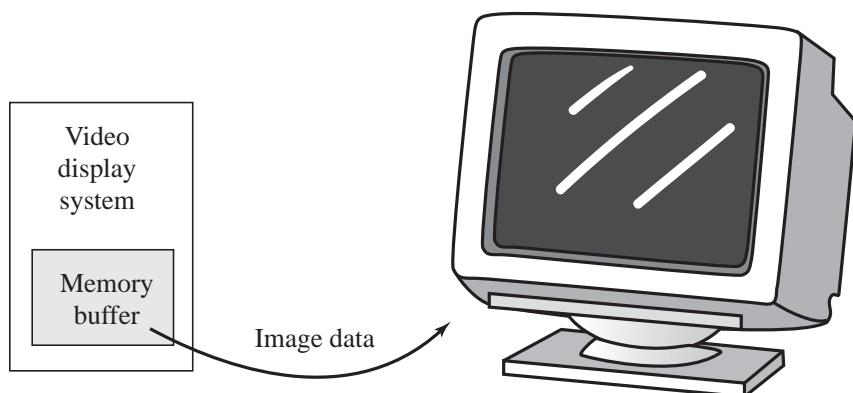
**Checkpoint 10.3:** What happens to the sound recording if data are missed?

### 10.2.3 Video Displays

Real-time generation of TV or video images requires an enormous data bandwidth. Consider the information bandwidth required to maintain an image on a graphics display (Figure 10.3). A video graphics array image might have 256 colors (8-bit), 480 rows, and 640 columns, all refreshed at about 60 Hz. Calculating the bandwidth in bytes per second, we get  $1 \cdot 480 \cdot 640 \cdot 60$ , which is 18,432,000 bytes/s. Luckily, we don't have to communicate each pixel for each image but rather just transmit the changes from the previous image. To achieve the necessary bandwidth, video interface hardware will use a combination of DMA and dual port memories.

**Figure 10.3**

Image data from memory are displayed on the graphics screen.

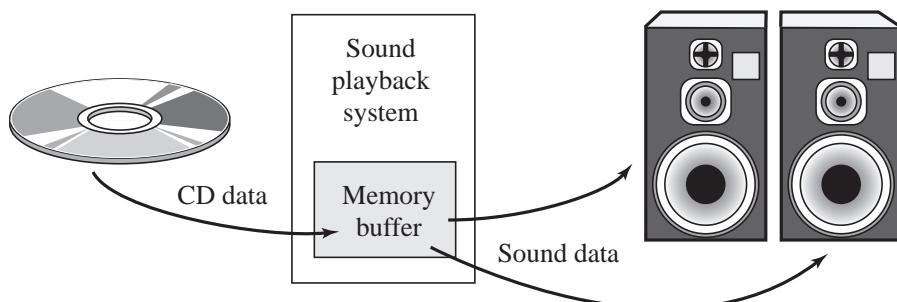


### 10.2.4 High-Speed Signal Generation

Examples of high-speed signal generation are CD-quality sound playback (16-bit, two-channel, 44 kHz) and real-time waveform generation. Sound playback also has two high-speed data channels: one for loading sound data into memory from CD and a second for playing the memory data out to the speakers (Figure 10.4). In the context of this chapter, we will define a high-speed interface as one that samples faster than a software-controlled interface would allow.

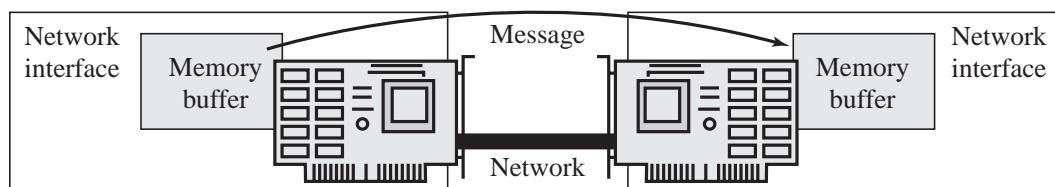
**Figure 10.4**

Sound data from the CD are loaded in memory, processed, then output to the speakers.



### 10.2.5 Network Communications

For many networks the communication bandwidth of the physical channel will exceed the ability of the software to accept or transmit messages. For these high-speed applications, we will look for ways to decouple the software that creates outgoing messages and processes incoming messages from the hardware that is involved in the transmission

**Figure 10.5**

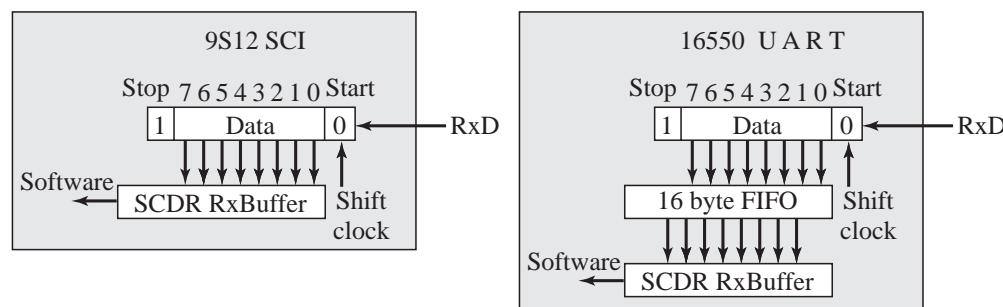
Message data are transmitted from the buffer of one computer to the buffer of another.

and reception of individual bits (Figure 10.5). Because the network load will vary, the average bandwidth (determined by how fast the transmission software can create outgoing messages and the reception software can process incoming messages) will be slower than the peak/maximum bandwidth that by the network hardware can achieve during transmission. This mismatch allows one network to be shared among multiple potential nodes.

**Checkpoint 10.4:** What happens in a communication system when packets are lost?

## 10.3 General Approaches to High-Speed Interfaces

**10.3.1 Hardware FIFO** If the software-controlled interface can handle the average bandwidth but fails to satisfy the latency requirements, then a hardware FIFO can be placed between the I/O device and the computer. A common application of the hardware FIFO is the serial interface. Assume in this situation that the average serial bandwidth is low enough for the software to read the data from the serial port and write them to memory. We saw in Chapter 7 that the latency requirement of a SCI input port like that of the 9S12 is the time it takes to transmit one data frame. To reduce this interface latency requirement (without changing the average bandwidth requirement), we can add a hardware FIFO between the receive shift register and the receive data register, as illustrated in Figure 10.6.

**Figure 10.6**

The 16550 UART employs a 16-byte FIFO to reduce the latency requirement of the interface.

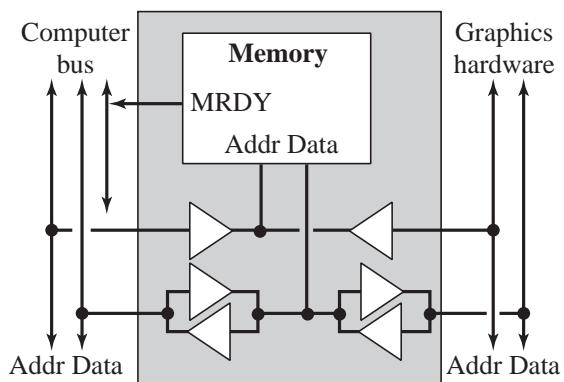
**Observation:** With a serial port that has a shift register, a FIFO of size  $n$ , and one data register, the latency requirement of the input interface is the time it takes to transmit  $n+1$  data frames.

A hardware FIFO, placed between the SCI data register and the transmit shift register, allows the software to write multiple bytes of data to the interface and then perform other tasks while the frames are being sent.

### 10.3.2 Dual Port Memory

One approach that allows a large amount of data to be transmitted from the software to the hardware is dual port memory. Dual port memory allows shared access to the same memory between the software and hardware (Figure 10.7). For example, the software can create a graphics image in the dual port memory using standard memory write operations. At the same time the video graphics hardware can fetch information out of the same memory and display it on the computer monitor. In this way, the data need not be explicitly transmitted from the computer to the graphics display hardware. To implement dual port memory, there must be a way to arbitrate the condition when both the software and hardware wish to access the device simultaneously. One mechanism to arbitrate simultaneous requests is to halt the processor (using a MRDY signal), so that the software temporarily waits while the video hardware fetches what it needs. Once the video hardware is done, the MRDY signal is released and the software resumes. None of the 9S12 expanded modes supports this sort of hardware-initiated cycle stretching. This hardware-controlled cycle stretching was defined in Chapter 9 as a partially asynchronous memory bus. Notice that except for the access conflict, both the software and graphics hardware can operate simultaneously at full speed.

**Figure 10.7**  
Dual port memory can be accessed by two different modules.



**Checkpoint 10.5:** Explain how the bidirectional tristate buffers connected to the memory data lines in Figure 10.7 work.

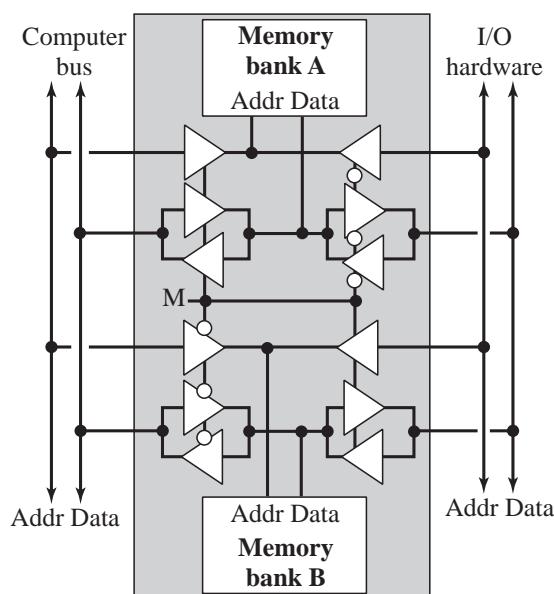
### 10.3.3 Bank-Switched Memory

Another approach similar to dual port memory is bank-switched memory. Bank-switched memory also allows shared access to the same memory between the software and hardware (Figure 10.8). The difference between bank-switched and dual port memory is that bank-switched memory has two modes. In one mode ( $M=1$ ), the computer has access to memory bank A, and the I/O hardware has access to memory bank B. In the other mode ( $M=0$ ), the computer has access to memory bank B, and the I/O hardware has access to memory bank A. Because access is restricted in this way, there are no conflicts to resolve. The master controller determines the value of M.

**Observation:** With bank-switched memory, the latency requirement of the software is the time it takes the hardware to fill (or empty) one memory bank.

**Figure 10.8**

Bank-switched memory can be accessed by two different modules, one at a time.



Many high-speed data processing systems employ this approach. The ADC hardware can write into one bank, while the computer software processes previously collected data in the other. When the ADC sampling hardware fills a bank, the memory mode is switched (by changing the M signal), and the software and hardware swap access rights to the memory banks. In a similar way, a real-time waveform generator or sound playback system can use the bank-switched approach. The software creates the data and stores them into one bank, while the hardware reads data from the other bank that was previously filled. Again, when the hardware is finished, the memory bank mode is switched by changing the M signal.

**Checkpoint 10.6:** How would you redesign the bank-switched memory in Figure 10.8 if the communication channel were simplex (data flows left to right only)?

## 10.4 Fundamental Approach to DMA

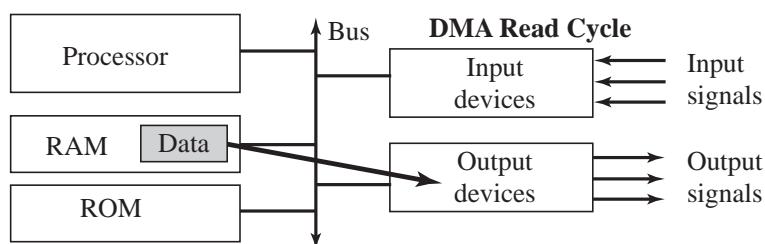
With a software-controlled interface (gadfly or interrupts) if we wish to transfer data from an input device into RAM, we must first transfer them from the input to the processor, then from the processor into RAM. To improve performance, we may transfer data directly from input to RAM or RAM to output using DMA. Because DMA bandwidth can be as high as the bus bandwidth, we will use this method to interface high-bandwidth devices like disks and networks. Similarly, because the latency of this type of interface depends only on hardware and is usually just a couple of bus cycles, we will use DMA for situations that require a very fast response. On the other hand, software-controlled interfaces have the potential to perform more complex operations than simply transferring the data to/from memory. For example, the software could perform error checking, convert from one format to another, implement compression/decompression, and detect events. These more complex I/O operations may preclude the use of DMA.

### 10.4.1 DMA Cycles

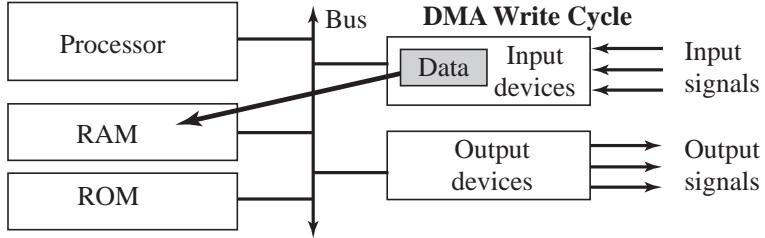
During a *DMA read cycle*, the processor is halted and data are transferred from memory to the output device (Figure 10.9). During a *DMA write cycle*, the processor is halted and data are transferred from the input device to memory (Figure 10.10). In some DMA

**Figure 10.9**

A DMA read cycle copies data from RAM or ROM to an output device.

**Figure 10.10**

A DMA write cycle copies data from the input device into RAM.



interfaces, two DMA cycles are required to transfer the data. The first DMA cycle brings data from the source into the DMA module, and the second DMA cycle sends the data to their destination.

#### 10.4.2 DMA Initiation

We can classify DMA operations according to the event that initiates the transfer. A *software-initiated* transfer begins with the program setting up and starting the DMA operation. Using DMA to transfer data from one memory block to another greatly speeds up the function. The efficiency of memory block transfers is very important in larger computer systems. Benchmarks on the Motorola 6808 show that even for block sizes as small as 4 bytes, it is faster to initialize a DMA channel and perform the transfer in hardware. As the block size increases, the performance advantage of DMA hardware over traditional software becomes more dramatic.

Most DMA applications, however, involve a *hardware-initiated* DMA transfer. For an input device, the DMA is triggered when new data are available. For an output device, the DMA is triggered on when the output device is idle. For periodic events, like data acquisition and signal generation, the DMA is triggered by a periodic timer. These are the exact triggers involved in gadfly and interrupt synchronization, as discussed in Chapters 3 and 4. The difference with DMA is that the servicing of the I/O need will be performed by the DMA controller hardware without software explicitly transferring each byte.

#### 10.4.3 Burst Versus Cycle Steal DMA

When the desired I/O bandwidth matches the computer bus bandwidth, then the computer can be completely halted, while the block of data is transferred all at once (Figure 10.11). Once an input block is ready, *burst mode DMA* is requested, the computer is halted, and the block is transferred into memory.

**Figure 10.11**

An input block is transferred all at once during burst mode DMA.

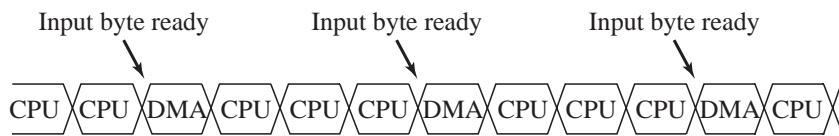


Figure 10.11 describes an input interface, but the same timing occurs on an output interface using burst mode DMA. For an output interface, the DMA is requested when the interface needs another block of data. During burst mode DMA, the computer is halted, and an entire block is transferred from memory to the output device.

If the I/O bandwidth is less than the computer bus bandwidth, then the DMA hardware will steal cycles and transfer the data one DMA cycle at a time. In *cycle steal mode*, the software continues to run, although a little bit slower. In either case the processor is halted during the DMA cycles. Figure 10.12 describes an input interface, but the same timing occurs on an output interface using cycle steal mode DMA. For an output interface, the DMA is requested when the interface needs another byte of data. During cycle steal DMA, one byte at a time is transferred from memory to the output device.

**Figure 10.12**

Each time an input byte is ready, it is transferred to memory using cycle steal DMA.



**Observation:** Some computers must finish the instruction before allowing a burst DMA. In this situation, the latency will be higher than cycle steal DMA, which does not need to finish the current instruction.

Since most I/O bandwidths are indeed less than the memory bandwidth, one technique to enhance speed is I/O buffering. In this approach a dedicated I/O memory buffer exists in the I/O interface hardware. This buffer is like the bank-switched memory discussed earlier. For example, on a hard disk read block operation, raw data come off the disk and into the buffer. During this time the processor is not halted. When the buffer is full, burst DMA is used to transfer the data into the system memory. Similarly, on a hard disk write block operation, the software initiates burst DMA to transfer data from system memory into the I/O buffer. Once the buffer is full, the I/O interface can write the data onto the disk.

**Checkpoint 10.7:** What is the maximum latency in a cycle steal DMA system?

#### 10.4.4 Single-Address versus Dual-Address DMA

Some computer systems allow the transfer of data between memory and the I/O interface to occur in one bus cycle, while others need two bus cycles to complete the transfer. In a *single-address DMA cycle*, the address and R/W line dictate the memory function to be performed, and the I/O interface is sophisticated enough to know it should participate in the transfer.

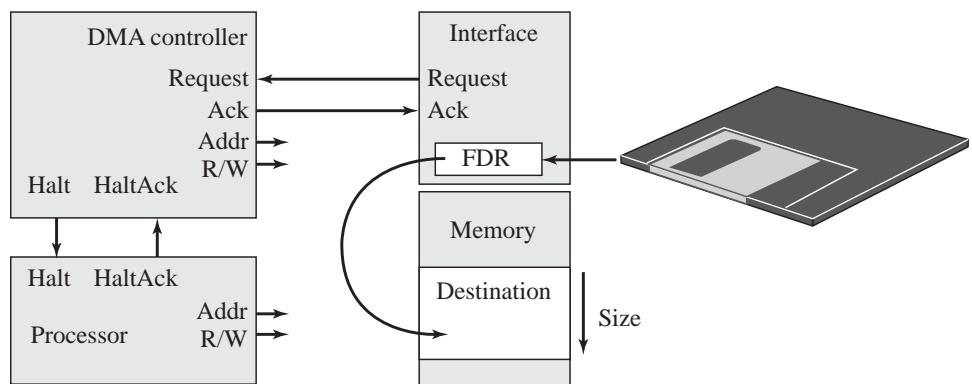
In this single-address example, the disk interface is reading bytes from a floppy disk (Figure 10.13). Cycle steal mode will be used, because the floppy disk bandwidth is slower than the bus. When a new byte is available, **Request** will be asserted, and this will request a DMA cycle from the DMA controller. The DMA controller will temporarily suspend the processor and drive the address bus with the memory address and the R/W to write. During this cycle the DMA controller will respond to the floppy interface by asserting the **Ack**. The floppy uses the **Ack** (ignoring the address bus and R/W) to know when to drive its data onto the bus (Figure 10.14).

**Observation:** Most microcontrollers do not support single-address DMA.

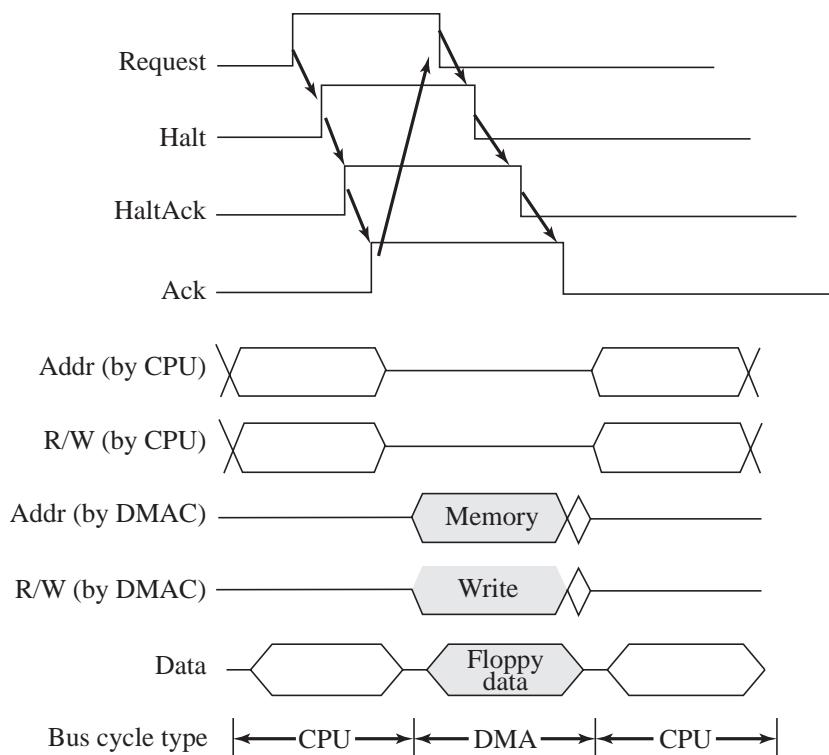
In a *dual-address DMA cycle*, two bus cycles are required to achieve the transfer (Figure 10.15). In the first cycle the data are read from the source address and copied into the DMA controller. During the first cycle the address bus contains the source address, and R/W signifies read. The information from the data bus is saved in the **Temp** register within

**Figure 10.13**

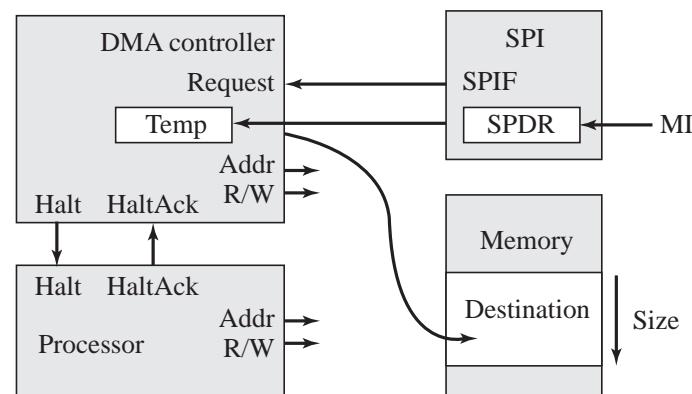
Block diagram showing the modules involved in a floppy disk read.

**Figure 10.14**

Timing diagram of a single-address DMA-controlled floppy disk read.

**Figure 10.15**

Block diagram showing the modules involved in a SPI read.

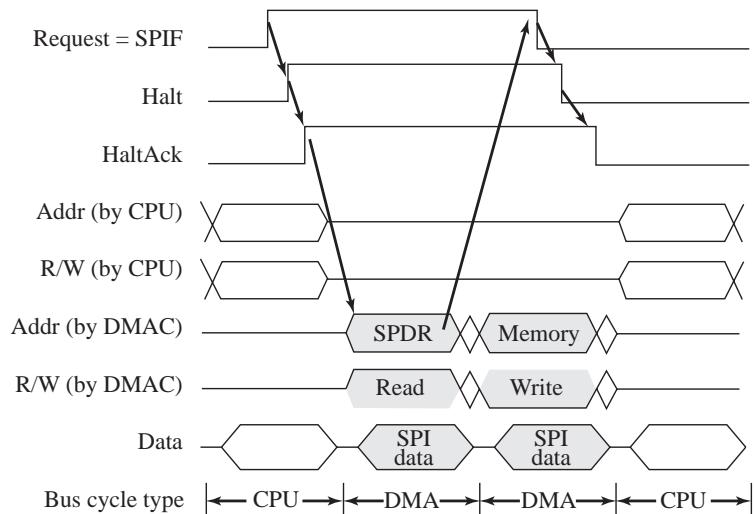


the DMA controller. In the second cycle, the data are transferred to the destination address. During the second cycle, the address bus contains the destination address, the data bus has the **Temp** data, and R/W signifies write.

In this dual-address example, the SPI interface is receiving bytes from a synchronous serial network. Cycle steal mode will be used because the SPI bandwidth is slower than the bus. When a new byte is available, **Request** will be asserted, and this will request a DMA cycle from the DMA controller. The DMA controller will temporarily suspend the processor and first drive the address bus with the SPI data register address (R/W=read), then in the second cycle the DMA controller will drive the address bus with the memory address (R/W=write). The SPI knows it has been serviced, because its data register has been read (Figure 10.16).

**Figure 10.16**

Timing diagram of a dual-address DMA-controlled SPI read.



**Observation:** Single-address DMA is twice as fast as dual-address DMA.

**Observation:** Dual-address DMA can be used with I/O devices not configured to support DMA.

**10.4.5 DMA Programming** Although DMA programming varies considerably from one system to another, there are a few initialization steps that most require. Table 10.2 lists the mode parameters that must be set to utilize DMA. There are two categories of DMA programming: initialization and completion. During initialization, the software sets DMA parameters so that DMA will begin.

Parameter	Possible Choices
What initiates DMA Type	Software trigger, input device, output device, periodic timer Burst versus cycle steal
Autoinitialization mode	Single event or continuous transfer
Precision	8-bit byte or 16-bit word
Mode	Single address or dual address
Priority	Should software completely halt? Should interrupts be serviced during DMA?
Synchronization	Set gadfly flag, or interrupt on block transfer complete

**Table 10.2**

DMA initialization usually involves specifying these parameters.

At the end of a block transfer, a done flag is set, and a number of additional actions may occur. If the system is armed, an interrupt can be generated. At the end of a block transfer in a continuous-transfer DMA, the parameters are autoinitialized so that the DMA process continues indefinitely. Table 10.3 lists additional parameters we will need to initialize.

Parameter	Definition
Source address	Address of the module (memory or input) that generates the data
Increment/decrement/static	Automatically $+1/-1/+0^*$ the source address after each transfer
Destination address	Address of the module (memory or output) that accepts the data
Increment/decrement/static	Automatically $+1/-1/+0$ the destination address after each transfer
Block size	Fixed number of bytes to be transferred

\*Obviously the DMA controller will  $+2/-2/+0$  when the precision is 16 bits.

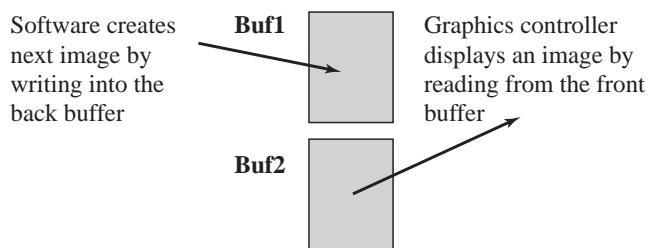
**Table 10.3**  
More DMA initialization parameters.

## 10.5 LCD Graphics

### 10.5.1 LCD Graphics Controller

The graphics controller implements a **double buffer** scheme. A double buffer consists of two buffers of fixed size, and is typically many bytes. In our case, the size of each of the two buffers will match the size required to store one image of the LCD display. A data flow graph is drawn in Figure 10.17. In this system, the software writes to one buffer, while the hardware simultaneously reads data from the other buffer. The **front buffer** contains the image that we see on the display, whereas the **back buffer** is used to create the image we will see next. The graphics controller is configured to refresh the display over and over from the data contained in the front buffer. Independent from the refresh operation, the software writes data to the back buffer. The differences between a FIFO queue and a double buffer are data size and queue length. The data size of a FIFO is typically one or two bytes. This means that one puts and gets single bytes into and out of the FIFO queue. The data size of the double buffer is typically large (e.g., 80, 256, or 1024 bytes). This means that one always saves and removes big blocks into and out of the double buffer. The FIFO queue length is large (typically ranging from 16 to 60,000 bytes). On the other hand, the double buffer has exactly two buffers. In Figure 10.17, Buf1 is shown as the back buffer and Buf2 is the front buffer. When a complete image is ready in the back buffer, the software triggers the double buffer to switch, making Buf2 the back buffer and Buf1 the front buffer.

**Figure 10.17**  
A double buffer allows you to store data into one buffer while retrieving data from the other buffer.



In general, we cannot interface an LCD graphics display directly to a microcontroller, because microcontrollers don't usually have enough RAM to support graphics, and the processing speeds of microcontrollers are often not fast enough to refresh an LCD graphics display and perform the necessary functions of the embedded system in real time. Furthermore, even with systems with more memory and processing power, it is appropriate to employ a separate graphics processor to perform the low-level functions of the graphics display. In this section, we will design a simple graphics controller for a 320 by 240 monochrome LCD display. The specific device we will interface is the Optrex F-51477, but its features are typical of other displays of this size. Although designing a graphics controller with discrete digital logic is not usually practical, the design does illustrate the concepts of a graphics controller and high bandwidth interfacing. However, in the next section we will design a more practical system using integrated components. Figure 10.18 shows a block diagram of the controller. Because the LCD crystal does not have persistence, the image must be redrawn periodically. As long as the refresh rate is faster than 60 times per second, the image will look continuous to our eyes. The 320 by 240 monochrome display has 76800 pixels, and if each pixel is controlled by one binary bit, then 9600 bytes of storage are required to define one image. One refresh cycle involves sending the entire 76,800-bit pattern to the LCD display.

**Figure 10.18**

A simple graphics controller built with bank-switched memory.

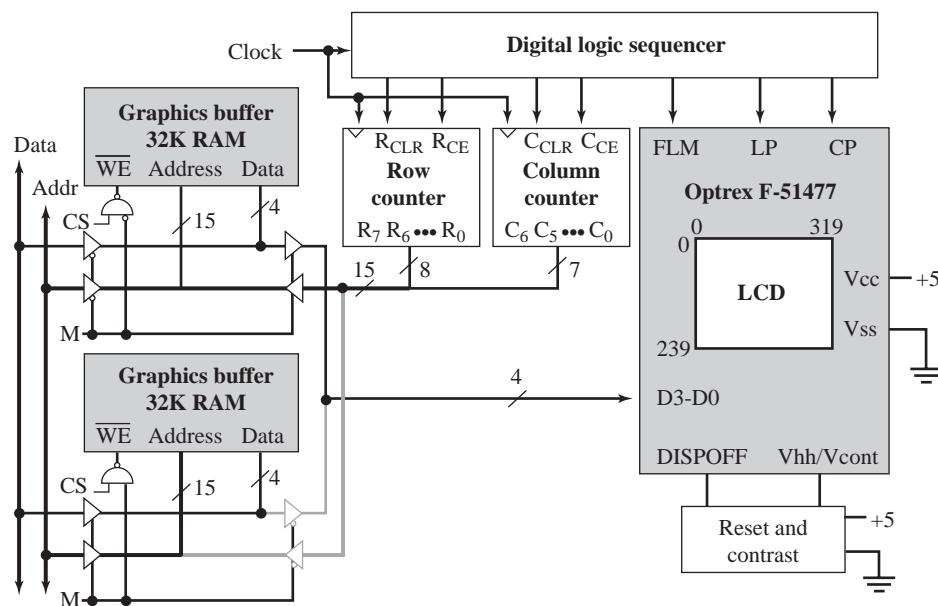
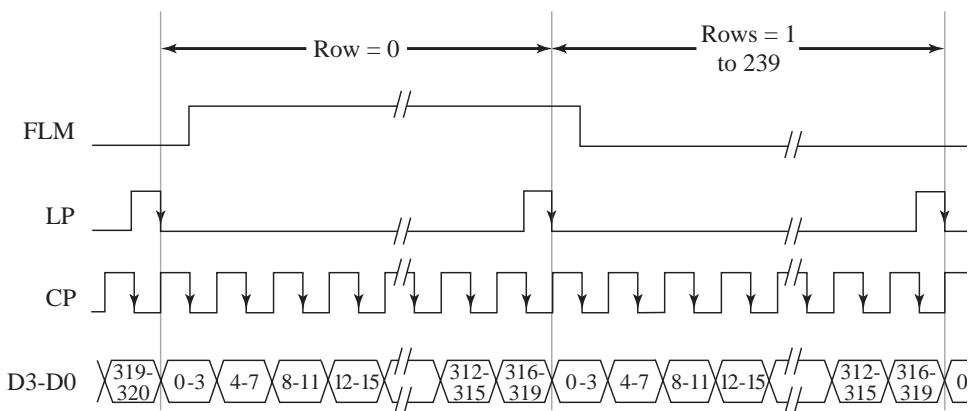


Figure 10.19 shows the basic timing required to send data to the LCD. Four bits are clocked into the device on each falling edge of **CP**. Therefore, our controller will change the data pins on the rising edge of **CP**. In particular, our address counters will be incremented on the rising edge of **CP**. Since there are 320 pixels in each row, it takes 80 **CP** pulses to enter data for one row. The falling edge of **LP** specifies the start of a new row, and there are 240 **LP** pulses required to update the entire display. The **FLM** signal will be high during the sending of the entire first row. If we wish to refresh at 60 Hz or faster, the period of the **CP** clock period must be smaller than  $4\text{s}/(60*78,800)$ , which is 846 ns. The column address specifies the x-coordinate (0 to 319), and the row address specifies the y-coordinate (0 to 239). Since the LCD display accepts 4 bits at a time, the **Column Counter** goes from 0 to 79, whereas the **Row Counter** steps from 0 to 239. The row and column counters can be implemented with a pair of 8-bit binary counters, such as 74LS590. To simplify addressing,

**Figure 10.19**

Basic timing used to refresh a 240 by 320 monochrome LCD display.



a 128-column by 256-row scheme will be employed. Although we allocate memory for 128 column steps (each step has 4 bits), we will use only 80 of them. Similarly, we allocate memory space for 256 rows, but use only 240 of them. The Optrex display implements a 4-bit data bus, but it will be easier to use an 8-bit RAM and just neglect four data pins. This means we will use a 32,768-byte static RAM (128\*256) to store the 9600 bytes contained in one graph image.

**Checkpoint 10.8:** What would happen if you tried to write 9S12 software to output four bits directly to the LCD faster than every 842 ns?

**Checkpoint 10.9:** What would the display look like if the refresh rate were only 10 Hz?

The other part of the controller is its interface with the microcontroller. In a computer system with DMA, we could use that high speed channel to load the back buffer, but in a simple microcontroller like the 9S12 we can interface the back buffer to the address/data bus. It is also possible to connect the back buffer to regular output ports of a 9S12 running in single-chip mode. The signal **M** will be interfaced to a standard output pin of the microcontroller, allowing the software to determine which RAM will be the front buffer and which will be the back buffer. When the signal **M** is equal to 1, the top RAM contains the front buffer and the bottom RAM contains the back buffer. In this mode, the microcontroller performs a write cycle to the interface and the 4-bit data are stored in the bottom RAM. Similarly, in this mode the counter address controls the top RAM, and the data from the top RAM is fed to the LCD. When **M** is equal to 0, the microcontroller write cycle stores into the top RAM, and the data from the bottom RAM goes to the LCD. If the address range of the interface is 0 to \$7FFF, then the positive logic **CS** signal combines the E clock, R/W, and address decoder.

$$CS = E \cdot \overline{R/W} \cdot \overline{A15}$$

The last part of the design will be the digital logic sequencer. The input to the sequencer will be a 2 MHz clock, which will be connected to the clock input of both counters. This clock will also be the CP signal. Notice that the LCD captures data on the falling edge, and the 74HC590 will increment on the rising edge. This is fast enough to refresh the display, but slow enough to simplify the memory interfacing. Whether or not the counter will increment is determined by its negative logic counter enable, **R<sub>CE</sub>** and **C<sub>CE</sub>**. Since the column address is to be incremented each time, it is always enabled.

$$C_{CE} = 0$$

The binary counters have negative logic clears. In order to make the Column Counter step from 0 to 79, we will set its clear condition to occur when the 7-bit counter value equals

80 (1010000). Let  $C_{CLR}$  be the negative logic clear signal, and  $C_0-C_6$  be the output of the row counter. Because the counter never reaches values 81 to 255, we can simplify the clear logic to

$$C_{CLR} = \overline{C_6 \cdot C_4}$$

We will enable the Row Counter to increment when the Column Counter equals 79 (1001111). The negative logic counter enable for the Row Counter will be

$$R_{CE} = \overline{C_6 \cdot \overline{C_5} \cdot \overline{C_4} \cdot C_3 \cdot C_2 \cdot C_1 \cdot C_0}$$

The LP signal will be high when  $CP=0$  and the Column Counter equals 79 (1001111).

$$LP = \overline{CP} \cdot \overline{R_{CE}}$$

In order to make the Row Counter step from 0 to 239, we will set the clear condition to occur when the counter value equals 240 (11110000). Let  $R_{CLR}$  be the negative logic clear signal and  $R_0-R_7$  the output of the row counter. Because the counter never reaches values 241–255, we can simplify the clear logic to

$$R_{CLR} = \overline{R_7 \cdot R_6 \cdot R_5 \cdot R_4}$$

Finally, the FLM signal is high during the output of the first row,

$$FLM = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

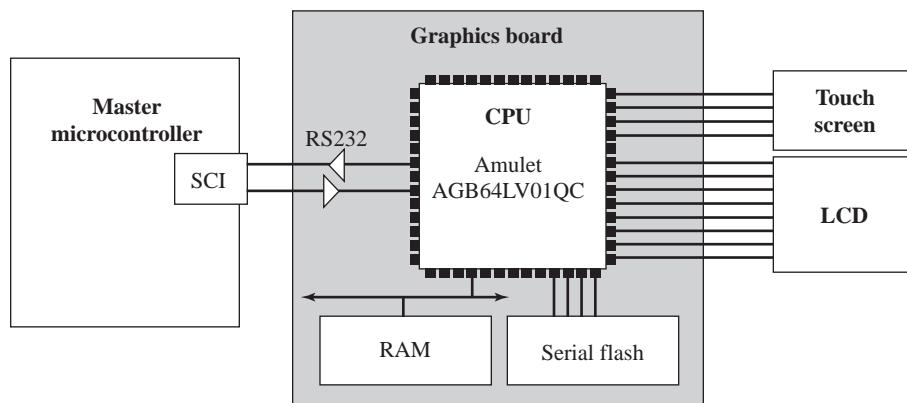
**Observation:** A single-buffer implementation of this LCD interface to a 9S12C32 was built and tested, and its details can be found on the website <http://users.ece.utexas.edu/~valvano/metroworks>.

### 10.5.2 Practical LCD Graphics Interface

Although HD44780-controlled LCD displays are simple and inexpensive, many embedded applications require the display of both graphics and text. When faced with a project requiring LCD graphics, the most efficient solution is to use an integrated system that includes the graphics controller. Some of the parameters to consider when choosing a graphics system are physical size, the number of x-y pixels and colors, contrast, brightness, and power requirements. Some of the display systems include a touch panel, like those used in a personal digital assistant (PDA). The simplest approach to designing a graphics system is to purchase a graphics controller board like the one shown in Figure 10.20. The interface board includes a graphical processing unit (GPU), such as the Amulet AGB64LV01-QC, which handles the graphical functions in the background. A GPU is similar to a regular CPU except that it performs many functions specific to graphics. The RAM contains one, two, or three buffers, where each buffer contains a mixture of graphics and text elements to draw on the display. The flash memory can hold GPU programs, graphical images, or fonts. The GPU coordinates the transfer of data from the front buffer to the display while simultaneously

**Figure 10.20**

A block diagram of an LCD graphics system using an Amulet CB-GT570 interface board.



interacting with the master controller as it draws images in the back buffer(s). In this configuration, the master microcontroller communicates with the graphics system using a standard interface such as RS232 serial. Manufacturers that produce LCD displays (e.g., Amulet, Hantronix, Lumex, Microtips, Optrex, Seiko, and Varitronix) often produce integrated graphics controllers as well.

## 10.6 Secure Digital Card Interface

The *Secure Digital Card* (SDC) is a popular standard for data storage in embedded systems (see Figure 10.21). The SDC is placed here in the high-speed I/O interfacing chapter because on high-end microcontrollers, the SDC interface would use DMA synchronization. However, when interfacing to a 9S12, we will use busy-wait synchronization with the understanding that peak bandwidth will be limited by 9S12 software and not SDC performance. The SDC is upward-compatible with a *Multi-Media Card* (MMC), so that the SDC-compliant interfaces can also use an MMC with an appropriate adapter. There are also smaller versions, such as *miniSD* and *microSD*, where the differences are in the connector rather than the electrical specification. The card itself has a microcontroller in it. The flash memory operations (such as erasing, reading, and writing) are performed on this microcontroller. The data is transferred between the memory card and the host controller as 512-byte blocks. In this way, the SDC can be viewed like a generic hard disk drive. In other words, the low-level drivers perform block reads and writes. A 2-gibabyte SDC will have over 4 million ( $2^{31}/2^9$ ) blocks, and the low-level driver will allow you to read or write any of these blocks. Program 10.1 shows a possible header file for such a low-level software interface. The file system, written as a higher-level driver, will format and partition this storage in a logical manner. You can download from the Internet fully-functioning SDC drivers and FAT16 file system code for the 9S12. The FAT16 file system will allow data exchange between the 9S12 and a personal computer. However, this section will serve as an introduction providing the basic ideas and fundamental theories. The file systems described in the next section will be much simpler than FAT16.

**Figure 10.21**  
Secure digital cards.



Courtesy of Jonathan Valvano

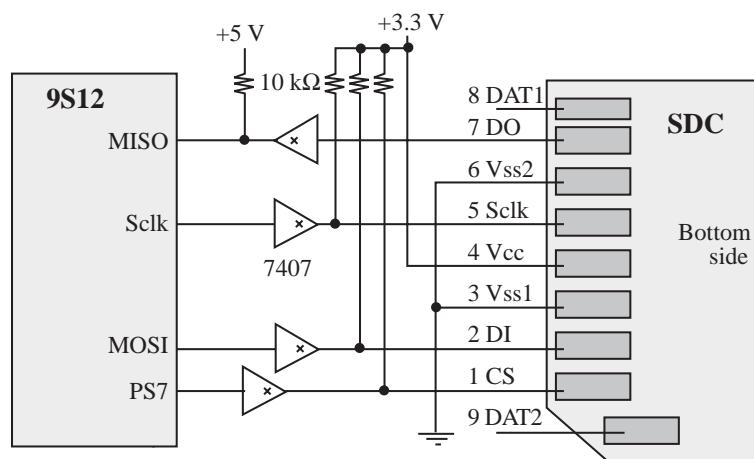
**Program 10.1**  
Header file for the SDC  
driver.

```
byte SDC_Initialize(void);
// sector is the 32-bit logical block address
// buff is a 512-byte buffer
byte SDC_Read(byte *buff, dword sector);
byte SDC_ReadBlocks(byte *buff, dword sector, word Count);
byte SDC_Write(const byte *buff, dword sector);
byte SDC_WriteBlocks(const byte *buff, dword sector, word Count);
```

**Checkpoint 10.10:** Why do we need to define the sector as a 32-bit number?

Figure 10.22 shows the connector pin-out and 9S12 interface. The SDC has nine contact pads, including four pins that compose the synchronous serial interface. **MOSI**, **MISO**, and **Sclk** are the usual SPI signals, and **PS7** is regular simple output. The three contacts are assigned for power supply. The **DAT1** and **DAT2** pins will not be used in this system. The SDC works at supply voltages from 2.7 to 3.6 V; hence, we will need a level shifter to interface the SDC. The 7407 open-collector noninverting buffers are used to connect +5 V logic to +3.3 V logic. About a 10 ns delay is added by the 7407 interface. The current consumption can reach up to 15 mA in standby and 50 mA during operation. Some SD card connectors provide an additional pin to let the software know whether or not a SDC is inserted into the slot.

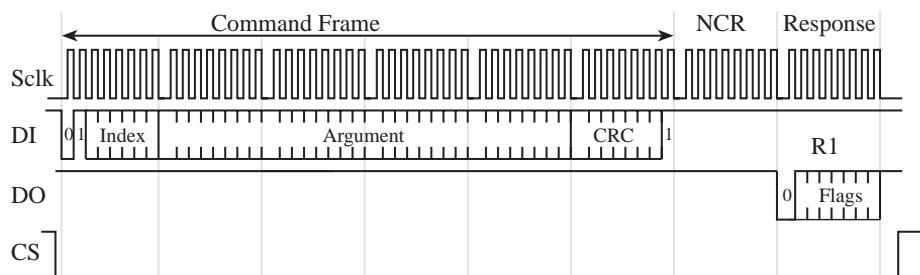
**Figure 10.22**  
SDC connector and  
9S12 interface.



**Checkpoint 10.11:** Explain how the 7407 interfaces +5 and +3.3 logic.

There are three possible modes to interface the SD card: SD 4-bit mode, SD 1-bit mode, and SPI mode. The communication protocol for the SPI mode is simple compared to the native SD modes. Therefore, the SPI mode is suitable for low-cost embedded applications. The SDC serial protocol matches the CPOL = 0, CPHA = 0 mode on the SPI. In SPI mode, the pin **7 DO** is always an output of the SDC, and pin **2 DI** is always an input. Data are transferred in a byte-oriented synchronous serial fashion. The command frame from 9S12 to SDC is a fixed-length, six-byte packet shown in Figure 10.23. When a command frame is transmitted to the card, a response to the command (R1, R2, or R3) will eventually come from the card. The microcontroller must continue to send 0xFF frames to **DI** and receive frames from **DO** until it receives a valid response. The command response time is 0 to 8 SPI frames (labeled as **NCR** in Figure 10.23). The **CS** signal must be held low during the entire transaction (command, response, and data transfer if data exist). The 7-bit CRC field is optional in SPI mode, but it is required as a bit field to compose a command frame. The **DI** signal must be kept high during read transfer.

**Figure 10.23**  
SDC command frame.



There are many SD commands, some of which are shown in Table 10.4. For details on all commands, please refer to the *SDA-SD Card Association* at <http://www.sdcards.org/>. There are three command response formats: R1, R2, and R3, depending on the command index. Response R1 is eight bits long and is returned for most commands. The R1 response has seven status bits, and a value of 0x00 means successful. Bit 6 is a parameter error, bit 5 is an address error, bit 4 is an erase sequence error, bit 3 is a communication CRC error, bit 2 is an illegal command, bit 1 is an erase reset, and bit 0 means the SDC is in the idle state. Most cards cannot change the block size, and it is fixed at 512 bytes.

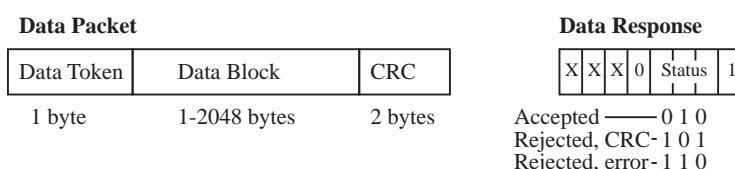
**Table 10.4**  
SD commands.

Index	Argument	Response	Data	Description
0	None	R1	No	Software reset
1 or 41	None	R1	No	Initiate initialization process
16	Block length[31:0]	R1	No	Change R/W block size
17	Address[31:0]	R1	Yes	Read a block
18	Address[31:0]	R1	Yes	Read multiple blocks
24	Address[31:0]	R1	Yes	Write a block
25	Address[31:0]	R1	Yes	Write multiple blocks
58	None	R3	No	Read OCR

After power-on reset, the SDC enters its native operating mode. To put the SDC in SPI mode, the following procedure must be performed. After the supply voltage reaches at least 2.2 V, wait a least one more millisecond. Set **DI** and **CS** high, and send 74 or more clock pulses to **Sclk**, and the card will become ready to accept native commands. Set the SPI clock rate between 100 and 400 kHz and then send an *Index = 0* command with **CS** low to reset the card. The card samples the **CS** signal when an *Index = 0* command is received. If the **CS** signal is low, the card enters SPI mode. Since the *Index = 0* command must be sent as a native command, the CRC field must have a valid value. When once the card enters SPI mode, the CRC feature is disabled, and the CRC is not checked, so that command transmission routine can be written with the hardcoded CRC value that is valid for only this command. When the *Index = 0* is accepted, the card will enter idle state and sends an R1 response with the idle bit (0x01).

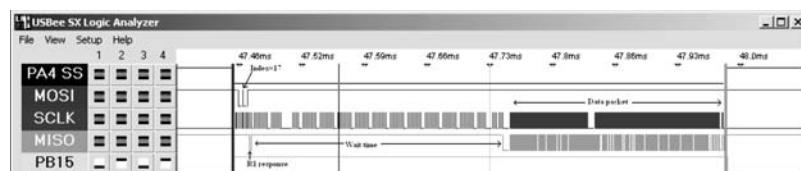
In idle state, the card accepts only commands with index values of 0, 1, 41, and 58. Any other commands will be rejected. Command *index = 58* allows you to check the working voltage range. Response R3 is an R1 plus information about the supply voltage. If the supply voltage is out of range, the card must be rejected. The card initiates initialization when a command with *index = 41* is received. To poll the end of the initialization, the host controller must repeatedly send commands with *index = 41* until the idle bit goes low. When the card is initialized successfully, the idle bit in the R1 response is cleared. That is, the R1 response will change from 0x01 to 0x00. The initialization process can take *hundreds of milliseconds* and large cards make take longer. After the idle bit is cleared, read/write commands can be sent. Command *index = 41* is recommended instead of *index = 1* for SDC. *Index = 1* initiation can be tried if *index = 41* is rejected. After initialization, the SPI clock rate should be changed as fast as possible to optimize the read/write performance. Most SD cards can handle SPI rates of 25MHz. For a 9S12 interface, the speed will be dominated by 9S12 software and limited by the 7407 interface.

**Figure 10.24**  
SD data packets.



In a transaction with data transfer, one or more data packets will be sent/received after the command response (see Figure 10.24). The data block is transferred as a data packet that consists of Token, Data Block, and CRC. The token for command indices 17, 18, and 24 is \$FE. The token for command index 25 is \$FC. A logic analyzer trace for a single-block read is shown in Figure 10.25. The resolution on the plot is not enough to see all the Sclk pulses. However, we see that the CS line (labeled PA4 SS) goes low and remains low for the entire transaction. The microcontroller begins by sending an *index* = 17 read block command. The argument for this command will be the sector address from which to read. The command response will be R1 with a value of 0x00, which means okay. Next, the microcontroller sends many frames (more than 300 s on this system) waiting for the SDC. The last half of the transfer is a data packet being sent from the SDC to the microcontroller containing the 512 bytes read from that sector. On this system, it took 535 s to read one block. If any error occurred during the read operation, an error token will be returned instead of a data packet.

**Figure 10.25**  
Single-block read packet.



To write a block, the controller sends a write command. If the response R1 is 0x00, the microcontroller sends a data packet to the card after an eight-clock pause. The write data packet has the same format as a read data packet. The CRC field can have any value unless the CRC function is enabled. When a data packet has been sent, the card responds with a Data Response immediately following the data packet. A 9S12 solution can be found at <http://users.ece.utexas.edu/~valvano/Starterfiles/>.

Original CD drives could read data at 150 kilobytes per second, and as faster drives arrived, manufacturers referred to their read speeds as a multiple of the original speed, referred to as X. Therefore, a 2X CD drive reads data at 300 kilobytes/sec. For DVDs the speeds are nine times faster than CDs. That is, a 1X DVD can read/write at 1,385,000 bytes/sec. Therefore, a 16X DVD can transfer 16 times faster than a 1X DVD. SD Cards and SDHC Cards have speed class ratings defined by the SD Association. The SD speed class ratings specify the following minimum write speeds based on “the best fragmented state where no memory unit is occupied” ([www.SDCard.org](http://www.SDCard.org)). Because of the software overhead in the 9S12, the transfer rates to the SDC will be much slower than the maximum. Table 10.5 shows example transfer rates or bandwidth for various mass-storage devices. Under most situations, the size of the data block transferred is fixed. The time to locate the physical location is called the **seek time**. Although seek time has a significant impact on disk performance, it does not affect the latency or bandwidth parameters. The bandwidth depends on the rotation speed of the disk and the information density on the medium. The transfer rates vary according to the physics of the drive

**Table 10.5**  
Bandwidth for various mass-storage devices.

Drive Type	Bandwidth in Mebibytes/sec
SATA channel	300
7200 RPM hard drive	70
16X DVD	22
52X CD-ROM	7.8
Class 2 SD card	2
Class 4 SD card	4
Class 6 SD card	6
1X CD-ROM	0.15

## 10.7 File System Management

### 10.7.1 Introduction

Solid-state disks can be made from battery-backed RAM or flash EEPROM. A personal computer uses disks made with magnetic storage media and moving parts. While this hard disk technology is acceptable for the personal computer because of its large size (>100s of gibibytes) and low cost (<\$100 OEM), it is not appropriate for an embedded system because of its physical dimensions, electrical power requirements, noise, sensitivity to motion (maximum acceleration), and weight. Embedded applications that might require disk storage include data acquisition, database systems, and signal generation. You can also use a solid-state disk in an embedded system to log debugging information. The personal desk accessory (PDA) devices currently employ solid-state disks because of their small physical size and low power requirements. Unfortunately, solid-state disks have smaller sizes and higher cost/bit than the traditional magnetic storage disk. In Lab 5, you will implement a solid-state disk using an SD card. A typical 2-Gibibyte SD card costs about \$10. The cost/bit is therefore about \$5/Gibibyte. Compare this cost to a 1-Tibibyte hard drive that costs about \$100. This cost/bit is 50 times cheaper at \$0.1/Gibibyte.

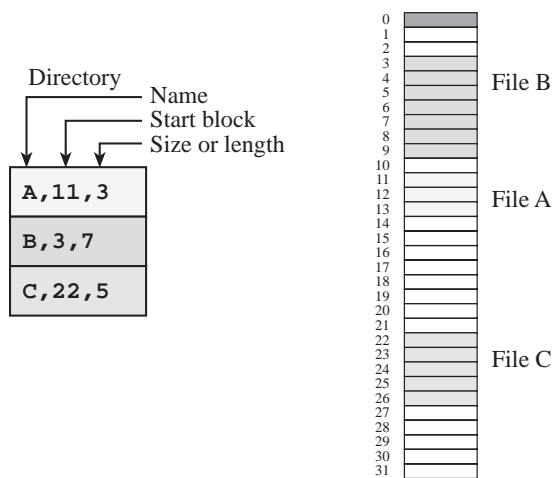
### 10.7.2 File System Allocation

There are three components of the file system: directory, allocation, and free-space management. This section introduces fundamental concepts and describes a simple file system. When designing a file system, it is important to know how it will be used. For example, during recording and playing back of sound, the data will be written and read in a sequential manner. Conversely, an editor produces more of a random access pattern for data reading and writing. Furthermore, an editor requires data insertion and removal anywhere from the beginning to the end of a file. The reliability of the storage medium and the cost of lost information will also affect the design of a file system.

*Contiguous allocation* places the data for each file at consecutive blocks on the disk, as shown in Figure 10.26. Each directory entry contains the file name, the block number of the first block, and the length in blocks. This method has similar problems as a memory manager. You could choose first-fit, best-fit, or worst-fit algorithms to manage storage. If the file can increase in size, either you can leave no extra space and copy the file elsewhere if it expands, or you can leave extra space when creating a new file. Assuming the directory is in memory, it takes only one disk-block read to access any data in the file. A disadvantage of this method is you need to know the maximum file size when a file is created, and it will be difficult to grow the file size beyond its initial allocation.

**Figure 10.26**

A simple file system with contiguous allocation.

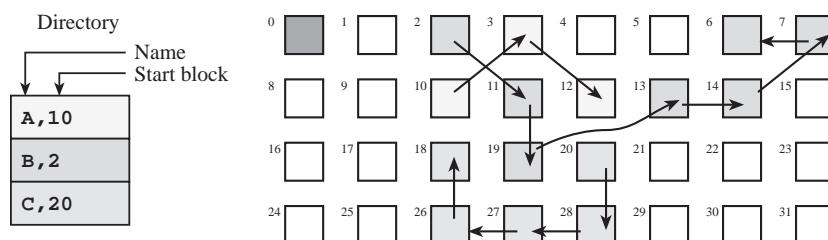


**Checkpoint 10.12:** The disk in Figure 10.26 has 32 blocks with the directory occupying block 0. The disk-block size is 512 bytes. What is the largest new file that can be created?

**Checkpoint 10.13:** You wish to allocate a new file requiring 1 block on the disk in Figure 10.26. Using first-fit allocation, where would you put the file? Using best-fit allocation, where would you put the file? Using worst-fit allocation, where would you put the file?

*Linked allocation* places a block pointer in each data block that contains the block address of the next block in the file, as shown in Figure 10.27. Each directory entry contains a file name and the block number of the first block. There needs to be a way to tell the end of a file. The directory could contain the file size, each block could have a counter, or there could be an end-of-file marker in the data itself. Sometimes, there is also a pointer to the last block, making it faster to add to the end of the file. Assuming the directory is in memory and the file is stored in  $N$  blocks, it takes on average  $N/2$  disk-block reads to access any random piece of data on the disk. Sequential reading and writing are efficient, and it also will be easy to append data to the end of the file.

**Figure 10.27**  
A simple file system with linked allocation.



**Checkpoint 10.14:** If the disk holds 2 Gibibytes of data broken into 512-byte blocks, how many bits would it take to store the block address?

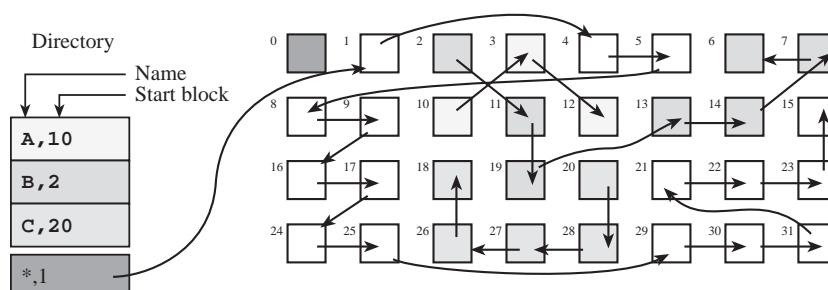
**Checkpoint 10.15:** If the disk holds 2 Gibibytes of data broken into 32k-byte blocks, how many bits would it take to store the block address?

**Checkpoint 10.16:** The disk in Figure 10.27 has 32 blocks with the directory occupying block 0. The disk-block size is 512 bytes. What is the largest new file that can be created?

**Checkpoint 10.17:** How would you handle the situation where the number of bytes stored in a file is not an integer multiple of the number of data bytes that can be stored in each block?

We can also use the links to manage the free space, as shown in Figure 10.28. If the directory were lost, then all file information except the filenames could be recovered. Putting the number of the last block into the directory with double-linked pointers improves recoverability. If one data block were damaged, then the remaining data blocks could be rechained, causing the information in the one damaged block to be lost.

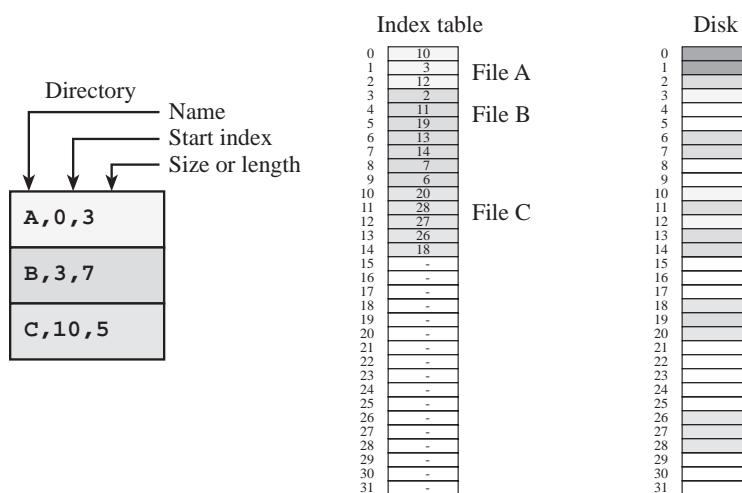
**Figure 10.28**  
A simple file system with linked allocation and free space management.



*Indexed allocation* uses an index table to keep track of which blocks are assigned to which files. Each directory entry contains a file name, an index for the first block, and the total number of blocks, as shown in Figure 10.29. One implementation of indexed allocation places all pointers for all files on the disk together in one index table. Another implementation allocates a separate index table for each file. Often, this table is so large it is stored in several disk blocks. For example, if the block number is a 16-bit number and the disk block size is 512 bytes, then only 256 index values can be stored in one block. Also, for reliability we can store multiple copies of the index on the disk. Typically, the entire index table is loaded into memory while the disk is in use. The RAM version of the table is stored onto the disk periodically and when the system is shut down. Indexed allocation is faster than linked allocation if we employ random access. If the index table is in RAM, then any data within the file can be found with just one block read. One way to improve reliability is to employ both indexed and linked allocation. The indexed scheme is used for fast access, and the links can be used to rebuild the file structure after a disk failure.

**Figure 10.29**

A simple file system with indexed allocation.



**Checkpoint 10.18:** If the block number is a 16-bit number and the block size is 512 bytes, what is the maximum disk size?

**Checkpoint 10.19:** A disk with indexed allocation has 2 gibibytes of storage. Each file has a separate index table, and that index occupies just one block. The disk block size is 1024 bytes. What is the largest file that can be created? Give two ways to change the file system to support larger files.

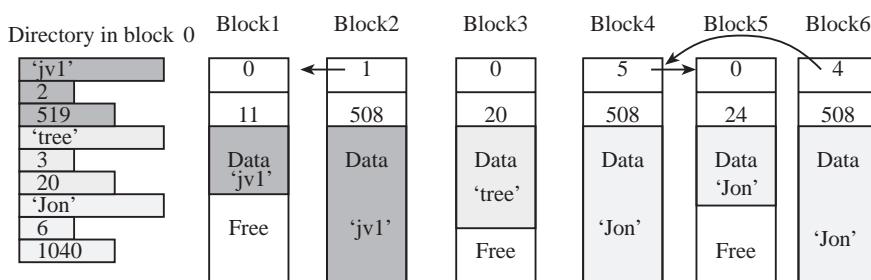
**Checkpoint 10.20:** The disk in Figure 10.29 has 32 blocks, with the directory occupying block 0 and the index table in block 1. The disk-block size is 512 bytes. What is the largest new file that can be created?

### 10.7.3 Simple File System

The first component is the *directory*, as shown in Figure 10.30. The block size is 512 bytes. In order to support disks larger than 32 Mebibytes, 32-bit block pointers will be used. The directory contains a mapping between the symbolic filename and the physical address of the data. Specific information contained in the directory might include the filename, the number of the first block containing data, and the total number of bytes stored in the file. One possible implementation places the directory in block 0. In this simple system, all files are listed in this one directory (there are no subdirectories). There is one fixed-size directory entry for each file. A filename is stored as an ASCII string in a fixed-size array. A null string (first byte 0) means no file. Since the directory itself is located in block 0, zero can

be used as a null-block pointer. In this simple scheme, the entire directory must fit into block 0, and the maximum number of files can be calculated by dividing the block size by the number of bytes used for each directory entry. In Figure 10.30, each directory entry is 16 bytes, so there can be up to  $512/16 = 32$  files. We will need one directory entry to manage the free space on the disk, so this disk format can have up to 31 files.

**Figure 10.30**  
Linked file allocation  
with 512-byte blocks.



Other information that one often finds in a directory entry includes a pointer to the last block of the file, access rights, date of creation, date of last modification, and file type.

The second component of the file system is the *logical-to-physical address translation*. Logically, the data in the file are addressed in a simple linear fashion. The logical address ranges from the first to the last. There are many algorithms one could use to keep track of where all the data for a file belong. This simple file system uses *linked allocation*, as illustrated in Figure 10.30. Recall that the directory contains the block number of the first block containing data for the file. The start of every block contains a link (the block number) of the next block and a byte count (the number of data bytes in this block). If the link is zero, this is last block of the file. If the byte count is zero, this block is empty (contains no data). Once the block is full, the file must request a free block (empty and not used by another file) to store more data. Linked allocation is effective for systems that employ sequential access. Sequential read access involves two functions similar to a magnetic tape: rewind (start at beginning) and read the next data. Sequential write access simply involves appending data to the end of the file. Figure 10.30 assumes the block size is 512 bytes and the filename has up to nine characters. The null-terminated ASCII string is allocated 10 bytes, regardless of the size of the string. The size entry in the directory (e.g., file 'jv1' has 519 bytes) is 32 bits, but the block pointers have only 16-bit precision. Since each data block has a 2-byte link and a 2-byte counter, each block can store up to 508 bytes of data.

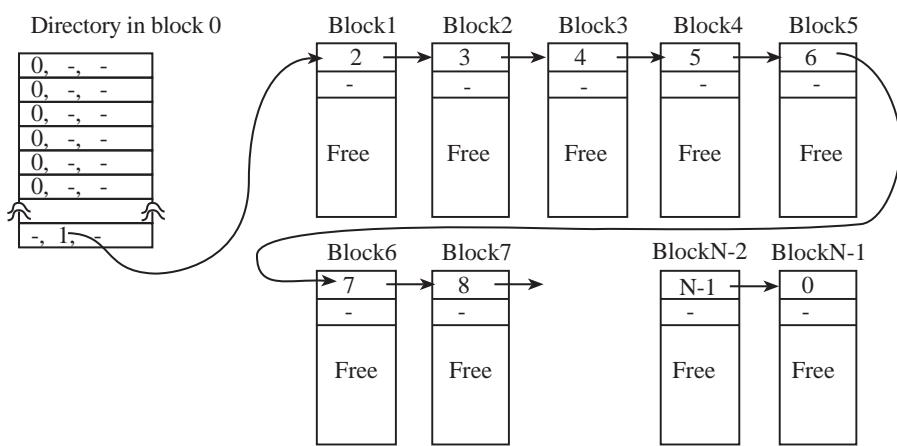
The third component of the file system is *free-space management*. Initially, all blocks except the one used for the directory are free and available for files to store data. To store data into a file, blocks must be allocated to the file. When a file is deleted, its blocks must be made available again. One simple free-space management technique uses *linked allocation*, similar to the way data is stored. Assume there are N blocks numbered from 0 to N – 1. An empty file system is shown in Figure 10.31. Block 0 contains the directory, and blocks 1 to N – 1 are free. You could assign the last directory entry for free-space management. This entry is hidden from the user. For example, this free-space file cannot be opened, printed, or deleted. It doesn't use any of the byte count fields, but it does use the links to access all of the free blocks. Initially, all of the blocks (except the directory itself) are linked together with the special-directory entry pointing to the first one and the last one having a null pointer.

When a file requests a block, it is unlinked from the free space and linked to the file. When a file is deleted, all of its blocks are linked to the free space again.

**Checkpoint 10.21:** If the directory shown in Figure 10.30 allocated 6 bytes for the filename instead of 10, how many files could it support?

**Figure 10.31**

Free-space management.

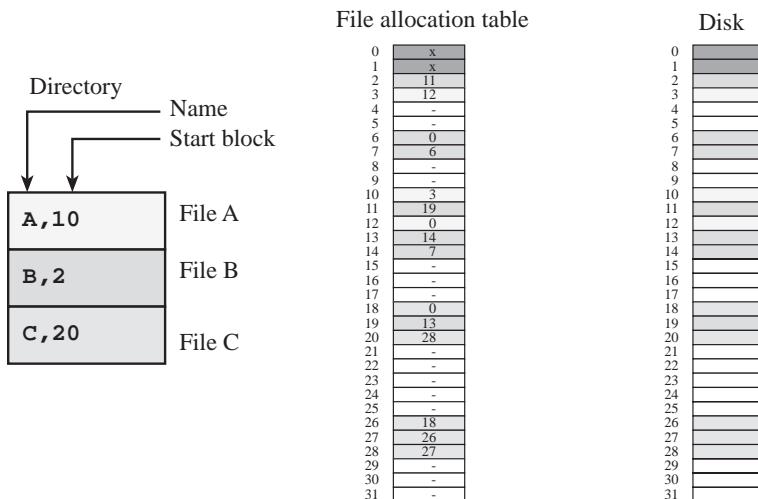


#### 10.7.4 File Allocation Table

The file allocation table (FAT) is a mixture of indexed and linked allocation, as shown in Figure 10.32. Each directory entry contains a file name and the block number of the first block. The FAT is just a table containing a linked list of blocks for each file. Figure 10.32 shows File A in blocks 10, 3, and 12. The directory has block 10, which is the initial block. The FAT content at index 10 is a 3, so 3 is the second block. The FAT content at index 3 is a 12, so 12 is the third block. The FAT content at index 12 is a NULL, so there are no more blocks in file A.

**Figure 10.32**

A simple file system with a file allocation table.

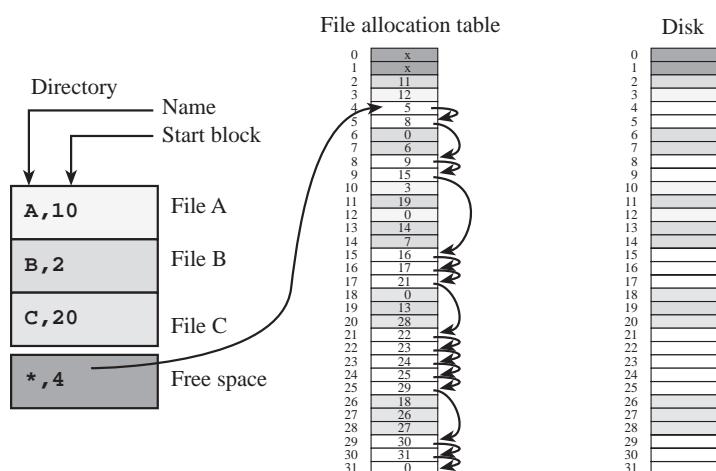


Many scientists classify FAT as a “linked” scheme, because it has links. However, other scientists call it an “indexed” scheme, because it has the speed advantage of an “indexed” scheme when the table for the entire disk is kept in main memory. If the directory and FAT are in memory, it takes just one disk read to access any data in a file. If the disk is very large, the FAT may be too large to fit in main memory. If the FAT is stored on the disk, then it will take 2 or 3 disk accesses to find an element within the file. The (-) in Figure 10.32, represent free blocks. In Figure 10.33, we can chain them together in the FAT to manage free space.

**Checkpoint 10.22:** The disk in Figure 10.32 has 32 blocks with the directory occupying Block 0 and the FAT in Block 1. The disk block size is 512 bytes. What is the largest new file that can be created?

**Figure 10.33**

The simple file system with a file allocation table showing the free-space management.



### 10.7.5 Internal Fragmentation

*Internal fragmentation* is storage that is allocated for the convenience of the operating system but contains no information. This space is wasted. Often this space is wasted in order to improve speed or to provide for a simpler implementation. The fragmentation is called “internal” because the wasted storage is inside the allocated region. In most file systems, whole blocks (or even clusters of blocks) are allocated to individual files, because this simplifies organization and makes it easier to grow files. Any space leftover between the last byte of the file and the first byte of the next block is a form of internal fragmentation called *file slack* or *slack space*. A file holding 26 bytes is allocated an entire block capable of storing 508 bytes of data. However, only 26 of those locations contained data, so the remaining 482 bytes can be considered internal fragmentation. The pointers and counters used by the OS to manage the file are not considered internal fragmentation, because even though the locations do not contain data, the space is not wasted. Whether or not to count the OS pointers and counters as internal fragmentation is a matter of debate. As is the case with most definitions, it is appropriate to document your working definition of internal fragmentation whenever presenting performance specifications to your customers.

Many compilers will align variables on a 32-bit boundary. If the size of a data structure is not divisible by 32 bits, it will skip memory bytes so the next variable is aligned onto a 32-bit boundary. This wasted space is also internal fragmentation.

Standard ASCII requires just seven bits per character. Most computer systems assign 8 or 16 bits for each character. Similarly, if you store 12-bit ADC data into two bytes per sample, there will be four bits of wasted space for each sample. The unused bits are a form of internal fragmentation.

### 10.7.6 External Fragmentation

*External fragmentation* exists when the largest file that can be allocated is less than the total amount of free space on the disk. External fragmentation occurs in systems that require contiguous allocation, like a memory manager. External fragmentation would occur within a file system that allocated disk space in contiguous blocks. Over time, free storage becomes divided into many small pieces. It is a particular problem when an application allocates and deallocates regions of storage of varying sizes. The result is that, although free storage is available, it is effectively unusable because it is divided into pieces that are too small to satisfy the demands of the application. The term “external” refers to the fact that the unusable storage is outside the allocated regions.

For example, assume we have a file system employing contiguous allocation. A new file with five blocks might be requested, but the largest contiguous chunk of free disk space is only three blocks long. Even if there are ten free blocks, those free blocks may be separated by allocated files, so that one still cannot allocate the requested file with five blocks, and the allocation request will fail. This is external fragmentation because there are ten free blocks but the largest file that can be allocated is three blocks.

## 10.8 Exercises

**10.1** For each term, give a definition in 32 words or less.

- |              |                          |                           |
|--------------|--------------------------|---------------------------|
| a) Latency   | f) Dual-port memory      | k) Indexed allocation     |
| b) Real time | g) Bank-switched memory  | l) FAT                    |
| c) DMA       | h) Double buffer         | m) Internal fragmentation |
| d) Seek time | i) Free-space management | n) External fragmentation |
| e) Bandwidth | j) Linked allocation     |                           |

**10.2** For each pair of terms, explain the similarities and differences in 32 words or less

- a) Burst versus cycle-steal DMA
- b) Single address versus dual address DMA
- c) Back buffer versus front buffer
- d) Write data required versus write data available

**10.3** The objective of this exercise is to interface various devices to the computer using DMA synchronization. You may assume the bus bandwidth is at least 8 million bytes/s. For each device you are asked to select the most appropriate DMA mode. Assume the devices support single-address DMA. The 16-bit address of the memory buffer used in each case is 0x1234.

**Write tape drive** Each tape block is 256 bytes. When a tape head is ready, the controller will signal that it is ready to accept all 256 bytes. At this time, the tape interface chip is ready to transfer as fast as possible all 256 bytes from the memory buffer at 0x1234 to the tape.

**Sound input** The sound waveform buffer is located in memory at 0x1234. Your interface will read the 8-bit ADC 1024 times at 22 kHz and store the data in the buffer. Your software will be smart enough to create two 512-byte buffers out of the 1024 bytes (double buffer) so that it can process one buffer while the A/D data are being stored automatically under DMA control into the other buffer. That is, when the 1024-byte wave buffer has been filled, the DMA system should repeat and fill it up again.

**Read hard drive** There is a 256-byte buffer at 0x1234 that your DMA system will fill with data from the hard disk. When a hard drive read head is ready, the controller will signal that it has the next byte from the disk. It takes 10 ms for the read head to be ready, then the 256 bytes of data can be transferred from the disk to memory at 2 million bytes/s.

Fill in the following table that specifies the most appropriate mode for each device.

	Tape	Sound	Disk
Cycle steal or block transfer			
Read or write transfer			
Autoinitialization (yes or no)			
Address increment or decrement			
DMA address register value			
DMA count register value			

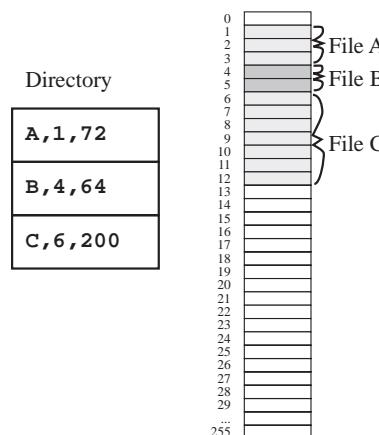
**10.4** When a 256-byte block is written to a floppy disk, there are 256 separate single-address DMA cycles in cycle steal mode. This question deals with just one of these DMA transfers. There are 14 events listed below. First you will eliminate the events that do not occur during the DMA cycle that saves one byte on the disk. In particular, list the events that will not occur. Second, you will list the events that do occur in the proper sequence.

- a) An interrupt is requested.
- b) Registers are pulled from the stack.
- c) Registers are pushed on the stack.

- d) The DMAC asks the processor to halt by activating its **Halt** signal.
- e) The DMAC deactivates its **Halt** request to the processor.
- f) The DMAC tells the FDC interface that a FMA cycle is occurring by activating its **Ack** signal; the DMA Controller drives the address bus with the FDC address; the DMAC drives the control bus to signify a write cycle (e.g., R/W=0); the memory drives the data bus; the FDC accepts the data.
- g) The DMAC tells the FDC interface that a DMA cycle is occurring by activating its **Ack** signal; the DMAC drives the address bus with the memory address; the DMAC drives the control bus to signify a memory read cycle (e.g., R/W=1); the memory drives the data bus; the FDC accepts the data.
- h) The FDC deactivates its **DMA Request** signal to the DMAC.
- i) The FDC requests a DMA cycle to the DMAC by activating its **Request** signal.
- j) The interrupt service routine is executed.
- k) The write head is properly positioned over the place on the disk.
- l) The processor address and control lines float; the processor responds to the DMAC that it is halted by activating its **HaltAck** signal.
- m) The processor resumes software execution.
- n) Wait until the current instruction is finished executing.

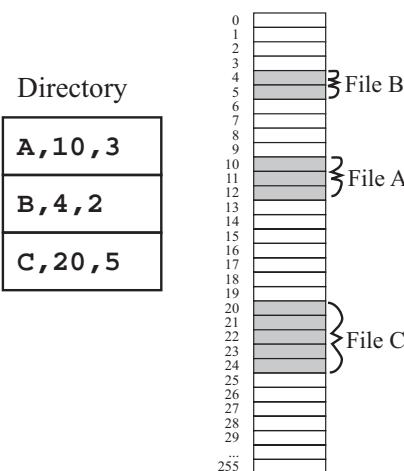
- 10.5** Consider a file system that uses **contiguous allocation** to define the set of blocks allocated to each file, as shown in Figure 10.34. There are 8192 bytes on this disk made up of 256 blocks, where each block is 32 bytes. This file system is used to record important “black box” information. Therefore, the file system is initialized to empty when the device is manufactured. Each time the system is turned on, a new file is created. While running, important data are stored into that file (open file, append data at the end, close file). Files are never deleted. Block 0 contains the directory and is not available for data. Each directory entry has three fields: name, block number of the first block, and total number of bytes stored. The example in Figure 10.34 shows File A with 3 allocated blocks (1,2,3 containing 32,32,8 bytes), File B with 2 blocks (4,5 containing 32,32 bytes), and File C with 7 blocks (6,7,8,9,10,11,12 containing 32,32,32,32,32,32,8). All 32 bytes of each data block can contain data for the file.
- a) Does this file system have any external fragmentation? Justify your answer.
  - b) Assume a file has **n** data blocks. It takes one *block read* to fetch the **directory**. On **average**, how many more *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of additional *block reads* that it takes to read a single byte in the file (worst case)?
  - c) Describe a simple mechanism to manage free blocks in this system. Be as explicit as possible, describing how many bytes in the directory are needed to manage the free space. Describe what the free space looks like after the disk is erased/formatted. Describe what the free space looks like when the disk is full.
  - d) File names are a single character. How many files can be stored? Justify your answer.
  - e) Assume you have **n** files each with of random size. Quantify the number of wasted bytes due to internal fragmentation. You may assume **n** is less than the number determined in part d).

**Figure 10.34**  
File system for  
Exercise 10.5.



- 10.6** Consider a file system that uses contiguous allocation, as illustrated by Figure 10.35. The block size is 32 bytes and all 256 blocks can be used to store data. The directory is not stored on the disk. Each directory entry contains the file name (e.g., A, B, C), the start block (e.g., File B starts at Block 4), and the number of blocks used in the file (e.g., File C has 5 blocks). The file sizes are always multiples of 32 bytes. That is, a file can contain only 32, 64, 96, ..., 8192 bytes. For example, File A is  $3 \times 32 = 96$  bytes, File B is  $2 \times 32 = 64$  bytes and File C is  $5 \times 32 = 160$  bytes. Does this system have internal fragmentation? Explain your answer.

**Figure 10.35**  
File system for  
Exercise 10.6.



- 10.7** Consider a file system that manages a 16 Megabyte ( $2^{24}$  bytes) EEPROM storage for a battery-powered embedded system. You are free to select from a range of EEPROM chips with different block sizes. The block size can be any power of 2 from 1 to  $2^{24}$  bytes. **Chip<sub>n</sub>** has a total of 16 Megabytes with block size  $2^n$  bytes. **Chip<sub>n</sub>** can perform a  $2^n$  byte block-write operation in 1 ms regardless of block size. For bandwidth reasons, therefore, you wish to choose a large block size. A block will be completely allocated to a file (you are not allowed to split one block between two files). 16 bytes of each block are used by the file system to manage pointers, type, size, and free space. However, if the file were to contain 1 byte of data, an entire block would be allocated, and the remaining  $2^n - 17$  bytes would be wasted. File sizes in this system are uniformly distributed from 50,000 to 150,000 bytes (this means any file size from 50,000 to 150,000 bytes is equally likely with an average size of 100,000 bytes). **You are asked to choose the largest block size with the constraint that the average internal fragmentation be less 5% of the total number of bytes stored.** Show your work.

- D10.8** One way to manage free space on a disk is to implement a **bit vector**. Each block is 32 bytes long, and there are 256 blocks. For each block on our 8-KiB disk, there will be a single bit specifying whether the block is free (1) or allocated. In C, we can define 256 bits as a byte-array with 32 entries.

```
unsigned char BitVector[32]; // 256 bits
```

Similar to the directory, the **BitVector** will exist both in RAM, as the above C definition, and on the disk as block 1. The format operation will initialize 254 of these bits to 1, performing:

```
BitVector[0] = 0x3F; // blocks 0,1 used (directory, BitVector)
for(i=1;i<32;i++) BitVector[i]=0xFF; // blocks 8-255 are free
eDisk_WriteBlock(BitVector,1); // update disk copy
```

- a)** Write a helper function that allocates a free-block updating the disk copy of **BitVector**.

```
// allocate a free block, returns a block number of a free block
// Output: block number 2 to 255 if successful and 0 if full
unsigned char AllocateBlock(void){
    eDisk_ReadBlock(BitVector,1); // fresh RAM copy
```

- b) Write a helper function that deallocates a block, updating the disk copy of BitVector.

```
// deallocate a free block
// Input: block number 2 to 255
void DeallocateBlock(unsigned char blockNum){
    eDisk_ReadBlock(BitVector,1);           // fresh RAM copy
```

**D10.9** Consider a file system that uses a **file translation table (FTT)** to define the set of blocks allocated to each file. There are 65536 bytes on this disk made up of 256 blocks, where each block is 256 bytes. Block 0 contains the directory and is not available for data. Each file has its own **FTT**, which is a null-terminated list of block numbers assigned to the file. Figure 10.36 shows a file with four allocated blocks: the first block at 12 and the last block at 22. The directory entry includes the file name, the total number of bytes, and the block number of its **FTT**. All 256 bytes of each data block can contain data for the file. For example, Figure 10.36 shows a file with 1024 bytes of data stored in five blocks (**FTT** and four data blocks).

- a) Does this file system have any external fragmentation? Justify your answer.
- b) Assume a file has **n** data blocks. It takes one *block read* to fetch the **FTT**. On **average**, how many more *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of additional *block reads* that it takes to read a single byte in the file (worst case)?
- c) Consider the linked allocation scheme described in Section 10.7.3. Assume the directory is in memory and the file has **n** data blocks. On **average**, how many *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of *block reads* that it takes to read a single byte in the file (worst case)?
- d) Assume you are given the following function that reads a 256-byte block from the disk.

```
int eDisk_ReadBlock(unsigned char *pt, // result returned by reference
                   unsigned char blockNum); // which block to read
```

Write a C function that returns a byte from a file at a random location. Do not worry about error handling (e.g., `eDisk_ReadBlock` error or address too big). The inputs to the function are `numFTT` (the block number of the file's **FTT**) and `address` (the byte address, where 0 is the first byte, 1 means second byte, etc.). You can use two buffers.

```
unsigned char FTTbuf[256]; // place to store FTT
unsigned char Databuf[256]; // place to store data
```

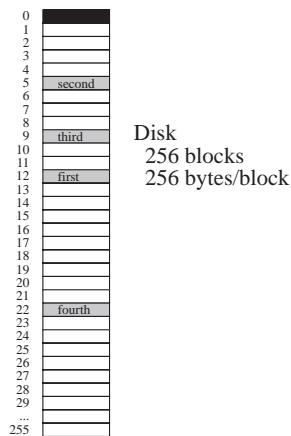
The prototype of the C function you have to write is

```
unsigned char eFile_Read(unsigned char numFTT, unsigned short address);
```

**Figure 10.36**  
File system for  
Exercise D10.9.

File translation table

	0	1	2	3	4	5	...	255
0	12							
1	5							
2	9							
3	22							
4	0							
5	0							
...								
255	0							



## 10.9 Lab Assignments

**Lab 10.1.** The overall objective of this lab is to build an LCD graphics display like the one described in Section 10.5.1. This is a complicated design that requires organization and testing. The first step is to choose digital logic devices fast enough for both the computer interface and the LCD timing. At 2 MHz 74HC logic will suffice, but at 25 MHz careful design and LCD layout will be required. One option for interfacing the system to a microcontroller running in single-chip mode is to interface a 19-bit shift register (15 address lines and 4 data lines) to the SPI. To write data into the back buffer, the software outputs three bytes to the SPI (address and data) then pulses it into the back buffer with a simple digital output line. Another option is to design a single buffered solution. In this system the counters and 19-bit shift registers have tristate outputs. To write to the buffer, the software activates the outputs of the shift registers, disables the outputs of the Row and Column counters, and stops the display clock. After the memory write cycle is complete, the software disables the outputs of the shift registers, enables the outputs of the Row and Column counters, and restarts the display clock.

The second part of the lab is the design of a low-level graphics device driver, which allows for drawing lines, drawing dots, displaying text, and erasing portions of the screen.

**Lab 10.2.** The overall goal is to design a dual port memory between two microcontrollers, as described in Section 10.3.2. There are a few ways to handle simultaneous requests that do not require dynamic bus cycle stretching with a MRDY signal. The first possibility is to design a hardware semaphore that the two microcontrollers can use to request access to the memory. In this scheme, only one microcontroller can access the memory at a time. A second approach is to give priority to the master computer, and if there is a conflict, let the master have access and ignore the slave. In this scheme, you'll have to add software checking to the slave to verify that a read/write access occurred. The third scheme requires design of both microcontrollers. The microcontrollers can be designed to run off external clocks. In this scheme, you generate a bus clock separately and feed both microcontrollers with the same clock, but out of phase. Then you design the memory interface to occur in the second half of the cycle (when E=1). In this way, the bus conflict is avoided.

The second part of the lab is the design of low-level device driver software to implement bidirectional communication. Run main programs in each microcontroller that determine the maximum bandwidth of this channel. Will it be faster than a simple SPI channel?

**Lab 10.3.** The overall goal is to develop a solid-state disk. Your system will be able to create files, append data to the end of a file, printout the entire contents of a file, and delete files. In addition, your system will be able to list the names and sizes of the available files. Basically, you will interface a SDC using the SPI port, then write a series of software functions that make it appear as a disk. Solid-state disks also can be made from battery-backed RAM or flash EEPROM. There must be a low-level device driver, and this software alone can directly access the SDC. A driver means there are separate header and code files. The high-level structure should include a directory that supports multiple logical files. Your system must support at least 10 files. The files are dynamically created and can grow in size (shrinking is easy to do but not necessary in this lab). There must be three software layers, including a high-level command interpreter (e.g., a `main.c` file with the editor program), a middle-level logical file system (e.g., `file.h` and `file.c`), and a low-level memory-access system (e.g., `SDC.h` and `SDC.c`). Each layer should have its own code file, and careful thought should go into deciding which components are private and which are public. All information (directory, linking, and data, must be stored on the SDC. The SDC is nonvolatile. This means if the microcontroller were to lose power, all information would be accessible when power is restored back to the microcontroller. The specific function prototypes are included for illustration purposes. A possible low-level driver prototype is given in Program 10.1. You are free to change function names and parameters, as long as the basic operations are supported. You may choose any size for the fixed-size disk blocks. You are free to adjust the specific syntax of the interpreter, as long as similar functions are available. You may implement any disk allocation method, as long as files can grow in size.

The first step is to interface the SDC to the microcontroller using the SPI interface. The second step is to develop a low-level device driver for the disk. Disks are partitioned into fixed-size blocks. The function of the low-level device is to allow read/write access to the SDC. The third step is to develop a file system. There should be separate `File.c` and `File.h` files containing software that implements the file system. The following prototypes illustrate the operations to be performed by the file system. It is okay to make copies of pointers and counters into regular memory while a command is being executed. But after each operation, all counters and pointers should be written back onto the

disk. Similarly, it is okay for the `Open` command to make copies of pointers and counters to be used by the `Write` command, but the `Close` command should leave the disk in a consistent state (ready for a power loss). All routines return a 1 if successful and a 0 on failure.

```
int File_Format(void);           // initialize file system
int File_Create(unsigned char name[8]); // create new file, make it empty
int File_Open(unsigned char name[8]); // open a file for appending
int File_Write(unsigned char byte); // save at end of the open file
int File_Close(void);           // close the file
int File_Print(unsigned char name[8]); // print entire contents
int File_Directory(void);       // print directory contents
int File_Delete(unsigned char name[8]); // delete this file
```

`File_Format` should return an error if the call to the low-level `Disk_Open` returns an error. `File_Create` should return an error if the directory is full or if the file already exists. `File_Open` should return an error if another file is already open (only one file can be open at a time in this simple system) or if the file doesn't exist. `File_Write` should return an error if the disk is full or if no file is open. `File_Close` should return an error if no file is open. `File_Print` and `File_Delete` should return an error if the file doesn't exist. The fourth step is to develop a simple interpreter/editor that illustrates the features of your file system. This software includes the main program, which first initializes and then implements the interpreter/editor. The following commands illustrate the types of features you need to develop:

f	format the disk, erasing all data and all files
d	display the directory, including names and sizes of the files
c mine	create a new empty file called "mine"
p yours	display the contents of file "yours"
a hers	open the file "hers", and add characters subsequently typed characters are added to the file
e his	close the file and return to the interpreter when an <esc> is typed (\$1B)
	erase file "his"

# 11 Analog Interfacing

## Chapter 11 objectives are to:

- ❖ Design analog amplifiers and filters
- ❖ Study building blocks for data acquisition, including sample and hold, multiplexers, DAC, and ADC
- ❖ Discuss the functionality of the built-in ADCs

**M**ost embedded systems include components that measure and/or control real-world parameters. These real-world parameters (like position, speed, temperature, and voltage) usually exist in a continuous (or analog) form. Therefore, the design of an embedded system involving these parameters rarely uses only binary (or digital) logic. Rather, we often will need to amplify, filter, and eventually convert these signals to digital form. In this chapter we will develop the analog circuit building blocks used in the design of data acquisition systems and control systems.

## 11.1 Resistors and Capacitors

### 11.1.1 Resistors

As engineers, we use resistors and capacitors for many purposes. The resistor or capacitor type is defined by the manufacturing process, the materials used, and the testing performed. The performance and cost of these devices varies significantly. For example, a 5% 0.25-W carbon resistor costs about 1 cent, while a 0.01% wire-wound resistor may cost \$20. It is important to understand both our circuit requirements and the resistor parameters so that we match the correct resistor type to each application, yielding an acceptable cost/performance ratio. We must specify in our technical drawings the device type and tolerance (e.g., 1% metal film) so that our prototype can be effectively manufactured. The characteristics of various resistor types are shown in Table 11.1.

**Table 11.1**

General specification of various types of resistor components.

Type	Range	Tolerance	Temperature Coef	Max Power
Carbon composition	1 Ω to 22 MΩ	5 to 20%	0.1%/°C	2 W
Wire-wound	1 Ω to 100 kΩ	>0.0005%	0.0005%/°C	200 W
Metal film	0.1 Ω to 10 <sup>10</sup> Ω	>0.005%	0.0001%/°C	1 W
Carbon film	10 Ω to 100 MΩ	>0.5%	0.05%/°C	2 W

*Source: Wolf and Smith, Student Reference Manual, Prentice-Hall, p. 272, 1990.*

The most common type of resistor is carbon composition. It is manufactured with hot-pressed carbon granules. Various amounts of filler are added to achieve a wide range of

resistance values. We add them to digital circuits as +5-V pull-ups. To improve the accuracy and stability of our precision analog circuits, we will use resistors with a lower tolerance and better temperature coefficient. For most applications 1% metal film resistors will be sufficient to build our analog amplifier circuits. For very high precision analog circuits we could use wire-wound resistors. Wire-wound resistors are manufactured by twisting a very long, very thin wire like a spring. The wire is coiled up and down a shaft in such a way to try and cancel the inductance. Since some inductance remains, wire-wound resistors should not be used for high-frequency (above 1 MHz) applications.

**Observation:** All resistors produce white (thermal) noise.

**Observation:** Metal film and wire-wound resistors do not generate 1/f noise, whereas carbon resistors do.

**Common error:** If you design an electronic circuit and neglect to specify explicitly the resistor types, then an incorrect substitution may occur in the layout/manufacturing stage of the project.

### 11.1.2 Capacitors

Similarly, capacitors come in a wide variety of sizes and tolerances. Polarized capacitors operate best when only positive voltages are applied. Nonpolarized or bipolar capacitors operate for both positive and negative voltages. We select a capacitor based on the following parameters: capacitance value, polarized/nonpolarized, voltage level, tolerance, leakage current (resistance), temperature coefficient, useful frequency response, and temperature range. Another parameter of capacitors is the maximum voltage rating. This parameter is important for high-voltage circuits but is of lesser importance for embedded microcomputer systems. Table 11.2 compares various capacitor types we could use in our circuit.

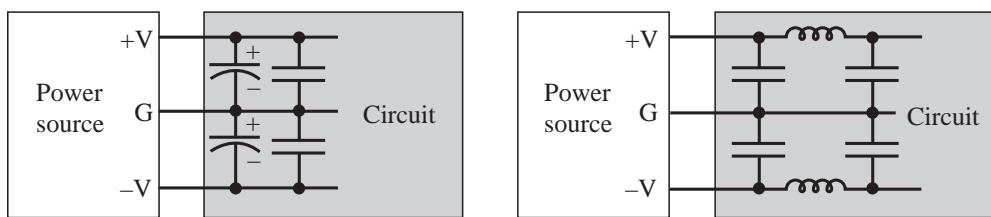
Type	Range	Tolerance	Temp coef	Leakage	Frequencies
Polystyrene	10 pF to 2.7 $\mu$ F	$\pm 0.5\%$	Excellent	10 G $\Omega$	0 to $10^{10}$ Hz
Polypropylene	100 pF to 50 $\mu$ F	Excellent	Good	Excellent	
Teflon	1000 pF to 2 $\mu$ F	Excellent	Best	Best	
Mica	1 pF to 0.1 $\mu$ F	$\pm 1$ to $\pm 20\%$		1000 M $\Omega$	$10^3$ to $10^{10}$ Hz
Ceramic	1 pF to 0.01 $\mu$ F	$\pm 5$ to $\pm 20\%$	Poor	1000 M $\Omega$	$10^3$ to $10^{10}$ Hz
Paper (oil-soaked)	1000 pF to 50 $\mu$ F	$\pm 10$ to $\pm 20\%$		100 M $\Omega$	100 to $10^8$ Hz
Mylar (polyester)	5000 pF to 10 $\mu$ F	$\pm 20\%$	Poor	10 G $\Omega$	$10^3$ to $10^{10}$ Hz
Tantalum	0.1 $\mu$ F to 220 $\mu$ F	$\pm 10\%$	Poor		
Electrolytic	0.47 $\mu$ F to 0.01 F	$\pm 20\%$	Ghastly	1 M $\Omega$	10 to $10^4$ Hz

*Source:* Modified from Wolf and Smith, *Student Reference Manual*, Prentice-Hall, p. 302, 1990; and Horowitz and Hill, *The Art of Electronics*, Cambridge University Press, p. 22, 1989.

**Table 11.2**

General specification of various types of capacitor components.

We will use capacitors for two purposes in our microcomputer-based embedded systems. First, we will place them on the DC power lines to filter the supply voltage to our circuits (Figure 11.1). A voltage supply typically will include ripple, which is added noise on top of the DC voltage level. There are two physical locations to place the supply filters. The first location is at the entry point of the supply voltage onto the circuit board. There are two approaches to this filter. If the supply noise is mostly voltage ripple, then two capacitors in parallel can be used. The large-amplitude polarized capacitor (e.g., 1 to 47  $\mu$ F electrolytic, tantalum) will remove low-frequency, large-amplitude voltage noise, and the nonpolarized capacitor (e.g., 0.01 to 0.1  $\mu$ F ceramic) will remove high-frequency noise. Two different types of capacitors are used because they are effective (i.e., behave like a capacitor) at different frequencies. The filter (CLC) is effective in situations where the supply is very noisy. The

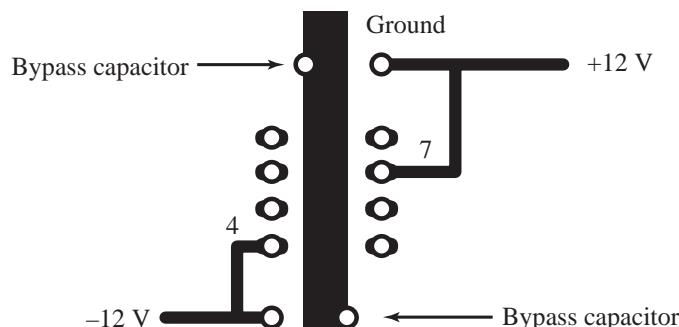


**Figure 11.1**  
DC supply filters.

filter is good for separating boards that have large current spikes from stealing power from each other. The magnitudes of the parameters depend on the amplitude of the supply ripple. The inductor in Figure 11.1 can be a ferrite bead. At DC, the bead is essentially a short current. The ferrite bead increases both its real and reactive impedance at high frequencies. The bead should be selected to have large impedance at the digital clock frequency. Panasonic makes a series of ferrite beads. The Panasonic EXC-ELDR25C has a DC resistance of  $0.08\ \Omega$ , can conduct 7 A DC, but has an  $80\ \Omega$  impedance at 24 MHz.

In addition to the supply filter, we will add bypass capacitors at the supply pins of each chip. It will be important to place these capacitors as close to the pin as physically possible (Figure 11.2). A nonpolarized capacitor (e.g., 0.01 to  $0.1\ \mu\text{F}$  ceramic or polyester) will smooth the supply voltage as seen by the chip. Placing the capacitor close to the chip prevents current surges from one chip from affecting the voltage supply of another.

**Figure 11.2**  
Printed circuit board layout positioning the bypass capacitors close to the chip.



The second application of capacitors in our embedded systems will be in the linear analog circuits of the low-pass filter, the high-pass filter, the derivative circuit, and the integrator circuit. For these applications we will select a nonpolarized capacitor even if the signal amplitude is always positive. In addition, we usually want a low-tolerance, low-leakage capacitor to improve the accuracy of the linear analog circuit. Ceramic capacitors are a low-cost medium quality choice for analog circuit design. They come in three tolerances. The best ceramic is **C0G**, which has a 5% tolerance and a temperature coefficient of  $30\ \text{ppm}/^\circ\text{C}$  or  $\pm 0.3\%$  over  $-55$  to  $125^\circ\text{C}$ . The middle grade is **X7R** ceramic, which has a 10% tolerance and a temperature coefficient of  $\pm 15\%$  over  $-55$  to  $125^\circ\text{C}$ . The lowest cost ceramic is **Z5U**, which has a 20% tolerance and a temperature coefficient of 22 to  $-56\%$  over 10 to  $86^\circ\text{C}$ . We can use **Z5U** for bypass capacitors on power lines, but we should use either **C0G** or **X7R** for analog filters.

**Performance tip:** Teflon, polystyrene, and polypropylene capacitors are excellent choices when designing precision analog filters, and Mylar or ceramic capacitors may be as well.

**Observation:** The maximum voltage rating of a capacitor is limited by its size.

**Common error:** Polarized capacitors often have a nonlinear capacitance versus frequency response, therefore using them in an analog filter will cause distortion.

**Observation:** A computer engineer interested in the field of embedded systems will find more job opportunities if he or she can develop microcontroller skills along with some analog circuit design skills.

## 11.2 Operational Amplifiers (Op Amps)

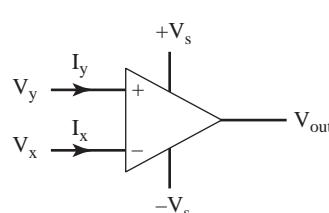
**11.2.1 Op Amp Parameters** Although the design of analog electronics is not an explicit objective of this book, we will include a brief discussion of analog circuit design issues often related to embedded systems. For example, small size, the ability to operate on a single voltage supply, and low supply current are three parameters typical of embedded systems. Other factors to consider are package size, cost, availability, and temperature range. There are hundreds of op amps currently available, making the choice of which devices to use confusing. The manufacturers of op amps publish a selection guide of their products to assist in finding an appropriate part. Table 11.3 lists performance parameters of four op amps, and these particular devices were chosen because they typify the kinds of op amps used in embedded systems, but the choice is not meant as a recommendation. These devices are all available in small packages with 1, 2, or 4 op amps per package. We use rail-to-rail op amps, such as the TLC2274, OPA4350, and MAX494 to design analog circuits that run on a single +5 V supply. The OPA4227 and OPA4132 are traditional op amps to be used in high-precision, low-noise applications. The MAX494 is a low-current device. We will begin our discussion of op amps with the ideal op amp. For simple analog circuits using high-quality devices, the ideal model will be sufficient. With an ideal op amp, the output voltage,  $V_{out}$ , is linearly related to the difference between the input voltages,  $V_{out} = K(V_y - V_x)$ , where the gain,  $K$ , is a very large number, as shown in Figure 11.3.

Single Op Amp	OPA227	OPA132	TLC2272	MAX495
Double Op Amp	OPA2227	OPA2132	TLC2274	MAX492
Quad Op Amp	OPA4227	OPA4132	TLC2274	MAX494
Description	High Precision	High-Speed FET	Rail-to-Rail	Rail-to-Rail
$K$ , Open loop gain	160 dB	130 dB	104 dB	108 dB
$R_{cm}$ , Input impedance	$1 \text{ G}\Omega \parallel 3 \text{ pF}$	$10^{13} \Omega \parallel 6 \text{ pF}$	$10^{12} \Omega \parallel 8 \text{ pF}$	
$R_{diff}$ , Input impedance	$10 \text{ M}\Omega \parallel 12 \text{ pF}$	$10^{13} \Omega \parallel 2 \text{ pF}$	$10^{12} \Omega \parallel 8 \text{ pF}$	$2 \text{ M}\Omega$
$V_{os}$ , Offset voltage	0.075 mV	0.5 mV	3 mV	0.5 mV
$I_{os}$ , Offset current	10 nA	50 pA	100 pA	6 nA
$I_b$ , Bias current	10 nA	50 pA	100 pA	60 nA
$e_n$ , Noise density	3 nV/ $\sqrt{\text{Hz}}$	23 nV/ $\sqrt{\text{Hz}}$	50 nV/ $\sqrt{\text{Hz}}$	25 nV/ $\sqrt{\text{Hz}}$
$f_1$ , Gain*bandwidth product	8 MHz	8 MHz	2.18 MHz	500 kHz
$dV/dt$ , Slew rate	2.3 V/ $\mu\text{s}$	20 V/ $\mu\text{s}$	3.6 V/ $\mu\text{s}$	0.2 V/ $\mu\text{s}$
$\pm V_s$ , Voltage supply	$\pm 5$ to $\pm 15$ V	$\pm 2.5$ to $\pm 18$ V	0 to 5 or $\pm 5$ V	2.7 to 6 V
$\pm I_s$ , Supply current	$\pm 3.8$ mA	$\pm 4.8$ mA	3 mA	170 $\mu\text{A}$

**Table 11.3**

Parameters for various op amps used in this chapter.

**Figure 11.3**  
Ideal op amp.



Voltage ranges of the inputs and outputs are bounded by the supply voltages,  $+V_s$  and  $-V_s$ . Op amp circuits found in many traditional analog design textbooks are powered with  $\pm 12$ -V supplies. On the other hand, we will find it convenient to power our embedded systems with a single voltage. More specifically, we will set  $+V_s$  to  $+5$  V and  $-V_s$  to ground. Either way, we assume the input and output voltages will be between  $+V_s$  and  $-V_s$ . A “rail-to-rail” op amp operates in its linear mode for output voltages all the way from  $-V_s$  to  $+V_s$ . We will see later how to create an effective  $\pm 2.5$  V analog supply from the single  $+5$  V digital supply. The input currents into the op amp are very small. Because the input impedance of the op amp is large, we will assume  $I_x$  and  $I_y$  are zero.

If a feedback resistor is placed between the output and the negative terminal of the op amp, then this feedback will select an output such that  $V_x$  is very close to  $V_y$ . In the ideal model, we let  $V_x = V_y$ . One way to justify this behavior is to recall that  $V_{out} = K \cdot (V_y - V_x)$ . Since  $K$  is very large, the only way for  $V_{out}$  to be between  $-V_s$  and  $+V_s$  is for  $V_x$  to be very close to  $V_y$ .

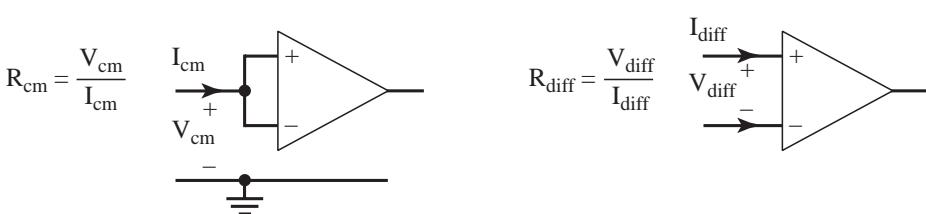
Positive feedback or no feedback drives  $V_{out}$  to equal  $-V_s$  or  $+V_s$ . If a feedback resistor is placed between the output and the positive terminal of the op amp, then this feedback will saturate the output to either the positive or negative supply. With no feedback, the output will also saturate. In both cases the output will saturate to the positive supply if  $V_y > V_x$ , and to the negative supply if  $V_x > V_y$ . We will see later, that positive feedback can be used to create hysteresis.

**Checkpoint 11.1:** What is the open loop gain of a MAX494 in units of V/V?

Although the input impedance of an op amp is large, it is not infinite and some current enters the input terminals. Figure 11.4 illustrates the definition of input impedance. We define the *common-mode input impedance*,  $R_{cm}$ , as the common-mode voltage divided by the common-mode current. We define the *differential input impedance*,  $R_{diff}$ , as the differential voltage divided by the differential current. These parameters vary considerably from op amp to op amp. The CMOS and FET devices have a very large input impedance.

**Figure 11.4**

Definition of op amp input impedance.



The next realistic parameter we will define is *open-loop output impedance*. When the op amp is used without feedback, the op amp output impedance is defined as the open-circuit voltage divided by the short-circuit current. The output impedance is a measure of how much current the op amp can source or sink. The open-loop output impedance of the TLC2274 is  $140\ \Omega$ .

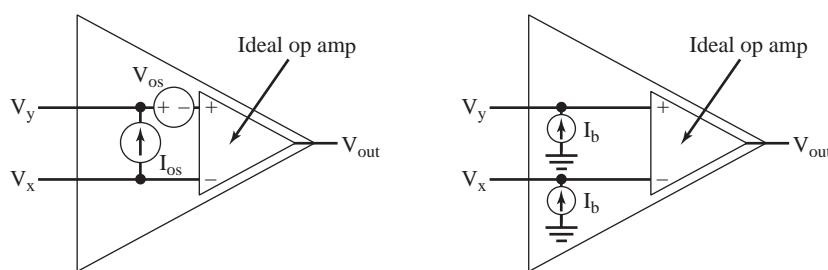
**Checkpoint 11.2:** The MAX494 has an output short-circuit current of  $30\text{ mA}$ . Assuming an output of  $5\text{ V}$ , what is its output impedance?

**Observation:** The input and output impedances of the op amp are not necessarily the same as the input and output impedances of the entire analog circuit.

As illustrated in Figure 11.5, the op amp *offset voltage*  $V_{os}$  is defined as the voltage difference between  $V_y$  and  $V_x$ , which yields an output of zero. The *offset current*  $I_{os}$  is defined as the current difference between  $V_y$  and  $V_x$ . There will be an output error equal to the offset voltage times the gain of the amplifier. Similarly, the offset current creates an offset voltage through the resistors of the circuit. Some op amps provide the capability to add an external potentiometer to nullify the two offset errors. Alternatively, we can reduce the offset error by using a more expensive lower offset op amp, such as the OPA4227. The op amp bias

**Figure 11.5**

Definition of op amp offset voltage, offset current, and bias current.



current  $I_b$  is defined as the common current coming out of both  $V_y$  and  $V_x$ . We can reduce the effect of bias current by selecting resistors in order to equalize the effective impedance to ground from the two input terminals.

**Observation:** The use of a null offset pot increases manufacturing costs and incurs a labor cost to adjust it periodically. Therefore, the overall system cost may be reduced by using more expensive op amps that do not require a null offset pot.

**Observation:** If the gain and offset are small enough not to saturate the output, then the offset error can be corrected in software by adding or subtracting an appropriate constant.

**Checkpoint 11.3:** Consider the situation where a MAX494 is used to create an analog amplifier with gain 100. What will be the output error due to offset voltage?

**Checkpoint 11.4:** Why can't a TLC2274 be used to create an analog amplifier with gain 1000?

The *input voltage noise*,  $V_n$ , arises from the thermal noise generated in the resistive components within the op amp. Due to the white-noise process, the magnitude of the noise is a function of the bandwidth (BW) of the analog circuit. This parameter varies quite a bit from op amp to op amp. To calculate the RMS amplitude of the voltage noise, we need to calculate,  $V_n = e_n \cdot \sqrt{(\text{BW})}$ . To reduce the effect of noise, we can limit the bandwidth of the analog system using an analog low pass filter. The *output voltage noise* will be the input voltage noise multiplied by the gain of the circuit.

There are two approaches to defining the transient response of our analog circuits. In the frequency domain we can specify the frequency and phase response. In the time domain, we can specify the step response. For most simple analog circuits designed with op amps, the frequency response depends on the op amp performance and the analog circuit gain. If the unity-gain op amp frequency response is  $f_1$ , then the frequency response at gain,  $G$ , will be  $f_1/G$ . The *bandwidth* (BW) is defined as the frequency at which the gain ( $V_{out}/V_{in}$ ) drops to 0.707 of the original. The *output slew rate* is the maximum slope that the output can generate. Slew rate is important if the circuit must respond quickly to changes in input (e.g., a sensor detecting discrete events). Alternatively, bandwidth is important if the circuit is responding to a continuously changing input (e.g., audio and video).

**Checkpoint 11.5:** Consider the situation where a MAX494 is used to create an analog amplifier with a gain of 100. What will be the bandwidth of this circuit? Given this bandwidth, what will be the RMS output voltage noise?

When we consider the performance of a linear amplifier, normally we specify the voltage gain, input impedance, and output impedance. These three parameters can be lumped into a single parameter,  $A_{db}$ , called the *power gain*. Let  $V_{in}$ ,  $R_{in}$  be the inputs and  $V_{out}$ ,  $R_{out}$  be the outputs of our amplifier. The input and output powers are  $P_{in} = V_{in}^2/R_{in}$  and  $P_{out} = V_{out}^2/R_{out}$ , respectively. Then the power gain in decibels has voltage gain and impedance components.

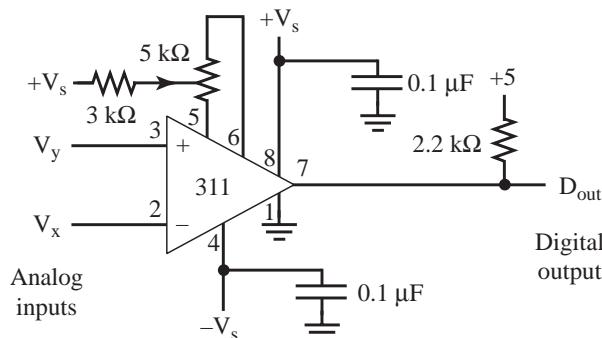
$$A_{db} = 10 \log_{10} \frac{P_{out}}{P_{in}} = 20 \log_{10} \frac{V_{out}}{V_{in}} + 10 \log_{10} \frac{R_{in}}{R_{out}}$$

### 11.2.2 Threshold Detector

Threshold detectors are very important in embedded systems. Recall that all through Chapter 6, a threshold detector was used to convert an external analog signal into a digital signal so that the period, pulse width, or frequency could be measured. An entire class of voltage comparators exist for this purpose. The LM311 is a typical voltage comparator with analog inputs ( $V_x, V_y$ ), analog voltage supplies ( $+V_s, -V_s$ ), digital output ( $D_{out}$ ), and digital supply (pin 1 ground). The LM311 will operate on a wide range of analog supply voltages ( $+V_s$  to  $-V_s$ ) from 0 to +5 V, all the way to -15 to +15 V. When the input  $V_y$  is above the input  $V_x$ , then the digital output  $D_{out}$  becomes high impedance, and the pull-up resistor creates a +5-V output. On the other hand, when  $V_y$  is below  $V_x$ , then  $D_{out}$  saturates to 0.4 V. The LM311 offset voltage of 7.5 mV can be reduced with the null offset pot connected to pins 5, 6. The LM311 offset current is 50 nA, and bias current is 250 nA. The input voltages of the LM311 must be between  $-V_s$  and  $+V_s$ . The LM311 can sink -8 mA output low current. Since the LM311 has an open-collector output, it cannot source any output high current. The LM311 settling time is 200 ns (Figure 11.6).

**Figure 11.6**

A voltage comparator using a LM311.



### 11.2.3 Simple Rules for Linear Op Amp Circuits

A wide range of analog circuits can be designed by following these simple design rules.

- 1. Choose quality components.** It is important to use op amps with good enough parameters. Similarly, we should use low-tolerance resistors and capacitors. On the other hand, once the preliminary prototype has been built and tested, then we could create alternative designs with less expensive components. Because a working prototype exists, we can explore the cost/performance trade-off.
- 2. Negative feedback is required to create a linear mode circuit.** As mentioned earlier, the negative feedback will produce a linear I/O response. In particular, we place a resistor between the negative input terminal and the output.
- 3. Assume no current flows into the op amp inputs.** Since the input impedance of the op amp is large compared to the other resistances in the circuit, we can assume that  $I_x = I_y = 0$ .
- 4. Assume negative feedback equalizes the op amp input voltages.** If the analog circuit is operating in linear mode with negative feedback, then we can assume  $V_x = V_y$ .
- 5. Choose resistor values in the  $1\text{ k}\Omega$  to  $1\text{ M}\Omega$  range.** To have the resistors in the circuit be much larger than the output impedance of the op amp and much smaller than the input impedance of the op amp, we choose resistors in the  $1\text{ k}\Omega$  to  $1\text{ M}\Omega$  range. If we can, it is better to restrict to the  $10\text{ k}\Omega$  to  $100\text{ k}\Omega$  range. If we choose resistors below  $1\text{ k}\Omega$ , then currents will increase. If the currents get too large, the batteries will drain faster and the op amp may not be able to source or sink enough current. As the resistors go above  $1\text{ M}\Omega$ , the white noise increases, and the current errors ( $I_{os}, I_b, I_n$ ) become more significant. In addition, low-tolerance precision resistors are expensive in sizes above  $2\text{ M}\Omega$ .

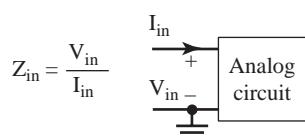
**6. The analog circuit BW depends on the gain and the op amp performance.** Let the unity gain op amp frequency response be  $f_1$  and the analog circuit gain be  $G$ . The frequency response or BW of the analog circuit will be  $f_1/G$ .

**7. Equalize the effective resistance to ground at the two op amp input terminals.** To study the bias currents, consider all other voltage sources as shorts to ground and all other current sources as open circuits. Adjust the resistance values in the circuit so that the impedance from the positive terminal to ground is the same as the impedance from the negative terminal to ground. In this way, the bias currents will create a common-mode voltage, which will not appear at the op amp output because of the common-mode rejection of the op amp (recall the op amp amplifies differential voltage inputs).

**8. The input impedance of the analog circuit is the input voltage divided by the input current.** If the analog circuit has a single input voltage, then the input impedance  $Z_{in}$  is simply the input voltage divided by the input current (Figure 11.7). If the input stage of the analog circuit is a differential amplifier with two input voltages, then we can specify the common-mode input impedance  $Z_{cm}$  and the differential mode input impedance  $Z_{diff}$  (Figure 11.8).

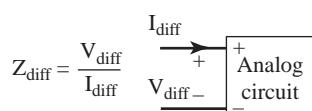
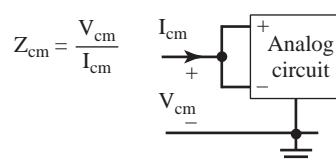
**Figure 11.7**

Definition of input impedance for an analog circuit with a single input.



**Figure 11.8**

Definition of input impedance for an analog circuit with two inputs.



**Observation:** In most cases, the differential mode input impedance of the analog circuit will be the differential mode input impedance of the op amp.

**9. Match input impedances to improve common-mode rejection ratio (CMRR).** If the input stage of the analog circuit is a differential amplifier with two input voltages, a very important performance parameter is called CMRR. It is assumed that the signal of interest is the differential voltage, whereas common-mode voltages are considered noise. The CMRR is defined to be the ratio of the differential gain divided by the common-mode gain. In decibels, it is calculated as

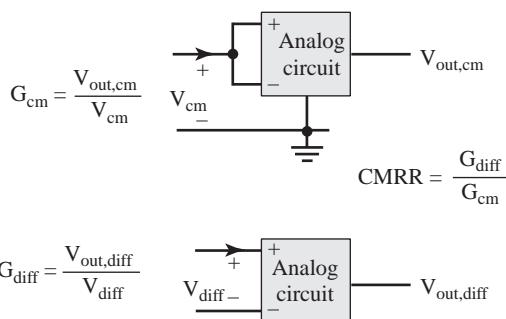
$$\text{CMRR} = 20 \cdot \log_{10} \frac{G_{\text{diff}}}{G_{\text{cm}}}$$

Therefore a differential amplifier with a large CMRR will pass the signal and reject the noise (Figure 11.9).

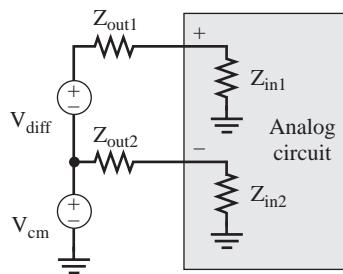
There are two design sets of impedances we must match to achieve a good CMRR. Each amplifier input has a separate input impedance to ground, shown as  $Z_{in1}$  and  $Z_{in2}$  in Figure 11.10. To improve CMRR, we make  $Z_{in1}$  equal to  $Z_{in2}$ . Similarly, signal source has

**Figure 11.9**

Definition of common-mode rejection ratio (CMRR).

**Figure 11.10**

Circuit model for improving the CMRR.



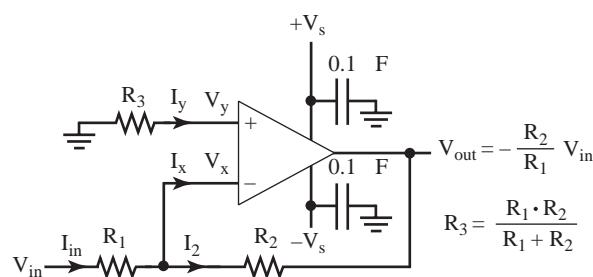
a separate output impedance to ground,  $Z_{out1}$  and  $Z_{out2}$ . Again, we try to make  $Z_{out1}$  equal to  $Z_{out2}$ . On the other hand, if  $Z_{in1}$  does not equal  $Z_{in2}$  or if  $Z_{out1}$  does not equal  $Z_{out2}$  then a common-mode signal (e.g., added noise in the cable) will appear as a differential signal to the analog circuit and thus be present in the output.

### 11.2.4 Linear Mode Op Amp Circuits

We will begin with one of the simplest linear mode analog circuits, which is the inverting amplifier (Figure 11.11). The gain is the  $R_2/R_1$  ratio. Notice that the gain response is independent of  $R_3$ . Thus, we can choose  $R_3$  to be the parallel combination of  $R_1||R_2$  so that the effect of the bias currents is reduced.

**Figure 11.11**

Inverting amplifier.



Because  $I_y$  is zero,  $V_y$  is also zero. Because of negative feedback,  $V_x$  equals  $V_y$ . Thus,  $V_x$  equals zero too. Because  $V_x$  is zero,  $I_{in}$  is  $V_{in}/R_1$  and  $I_2$  is  $-V_{out}/R_2$ . Because  $I_x$  is zero,  $I_{in}$  equals  $I_2$ . Setting  $I_{in}$  equal to  $I_2$  yields

$$V_{out} = -(R_2/R_1) \cdot V_{in}$$

The input impedance ( $Z_{in}$ ) of this circuit (defined as  $V_{in}/I_{in}$ ) is  $R_1$ . If the circuit were built with an OPA227 (which has a gain bandwidth product of 8 MHz), the bandwidth of this circuit would be 8 MHz divided by the gain.

**Common error:** This low-input impedance of  $Z_{in}$  may cause loading on the previous analog stage.

**Observation:** The inverting amplifier input impedance is independent of the op amp input impedance.

The negative feedback will reduce the output impedance of the amplifier,  $Z_{\text{out}}$ , to a value much less than the output impedance of the op amp itself,  $R_{\text{out}}$ . To calculate  $Z_{\text{out}}$ , we first determine the open circuit voltage

$$V_{\text{open}} = -(R_2/R_1) \cdot V_{\text{in}}$$

We next determine the short circuit current,  $I_{\text{short}}$ . This means we consider what would happen if the output were shorted to ground. If the output is shorted, the circuit is no longer in feedback mode, and  $V_x$  will not equal  $V_y$ . In fact,  $V_x$  will be a simple voltage divider from  $V_{\text{in}}$  through  $R_1$  and  $R_2$  to ground,

$$V_x = V_{\text{in}} \cdot R_2/(R_1 + R_2)$$

Because of the large open-loop gain, the ideal output will attempt to become

$$V_o = K \cdot (V_y - V_x) = -K \cdot V_{\text{in}} \cdot R_2/(R_1 + R_2)$$

The short circuit current will be a function of the ideal output voltage, and the output resistance of the op amp,

$$I_{\text{short}} = V_o/R_{\text{out}}$$

The output impedance of the circuit is defined to be the open circuit voltage divided by the short circuit current, which for this inverting amplifier is

$$Z_{\text{out}} = V_{\text{open}}/I_{\text{short}} = R_{\text{out}} \cdot (R_2 + R_1)/(K \cdot R_1)$$

**Observation:** The output impedance of analog circuits using op amps with negative feedback is typically in the  $M\Omega$ 's.

**Example 11.1** Design an analog circuit with a gain of  $-10$ .

**Solution** The transfer relationship will be  $V_{\text{out}} = -10V_{\text{in}}$ . We can use the template in Figure 11.11 and select  $R_1$  and  $R_2$ , such that  $R_2/R_1$  equals 10. The rule of thumb is to select resistors in the range of  $1\text{ k}\Omega$  to  $1\text{ M}\Omega$ . One possible solution is to choose  $R_1$  equal to  $10\text{ k}\Omega$  and  $R_2$  equal to  $100\text{ k}\Omega$ .

**Checkpoint 11.6:** If  $R_1$  is equal to  $10\text{ k}\Omega$  and  $R_2$  is equal to  $100\text{ k}\Omega$ , what value should you choose for  $R_3$  to remove the error due to the bias currents?

**Common error:** Precision reference chips do not provide much output current and should not be used to power other chips.

A *mixed-signal design* includes both analog and digital components. The classic approach to combining analog and digital circuits is to power the analog system with a low noise  $\pm 12\text{ V}$  power supply, maintain separate analog and digital grounds, and connect the analog ground to the digital ground only at the ADC. As mentioned earlier, one of the limitations of the ADC built into a microcontroller is that analog signals extending beyond the  $0$  to  $+5\text{ V}$  range will permanently damage the microcontroller. One approach to allowing signed analog voltages, while still using a single voltage supply and protecting the microcontroller, is to create an analog ground that is at a different potential from the digital ground. For our 9S12 system, which is powered with a  $+5\text{ V}$  supply, we will create an analog ground that is  $2.5\text{ V}$  relative to the digital ground. Thus, analog signals ranging from  $-2.5$  to  $+2.5\text{ V}$  are actually  $0$  to  $+5\text{ V}$  relative to the 9S12 digital ground. The first step to implementing this approach is to use an analog reference chip, like the ones shown in Table 11.4, to create a low-noise  $+2.50\text{ V}$  signal (the analog ground). The second step is to connect power to the analog

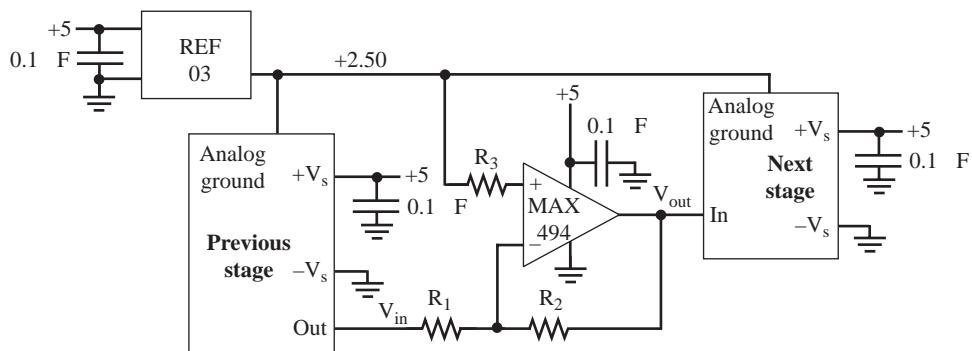
Part	Voltage (V)	$\pm$ Accuracy (mV)
AD1580, AD589, REF1004, MAX6120, LT1034, LM385	1.2	1 to 15
ADR420, ADR520, REF191, MAX6191, LT1790, LM4120	2.048	1 to 10
AD580, REF03, REF43, REF1004, MAX6192, MAX6225, LT1389, LM336	2.5	1 to 75
AD1583, ADR530, ADR423, REF193, MAX6163, LT1461, LM4120	3.0	1.5 to 10
ADR366, REF196, MAX6331, LT1461, LM3411, LM4120	3.3	4 to 10
AD1584, ADR540, ADR292, REF198, MAX6241, LT1790, LM4040	4.096	2 to 8
ADR425, AD586, REF02, REF195, MAX6250, LT1027, LT1236, LM336	5.0	2 to 20
AD581, AD587, AD633, REF01, LT1236, LM4040, LM4050	10.0	5 to 30

**Table 11.4**

Parameters of various precision reference voltage chips.

circuits with  $-V_s$  set to digital ground, and the  $+V_s$  set to the  $+5$  V supply. We will use rail-to-rail op amps that operate on 0 to  $+5$  V power, like the TLC2274, OPA4350, or MAX494. The last step is to replace all connections to analog ground with the low-noise  $+2.50$  V signal. Figure 11.12 shows the inverting amplifier from Figure 11.11, redesigned to operate on a single  $+5$  V supply. The signals  $V_{in}$  and  $V_{out}$  are allowed to vary from 0 to  $+5$  V relative to digital ground, but relative to analog ground, these signals will vary from  $-2.5$  to  $+2.5$  V. Shunt voltage references like the LM4040 and LM4041 also can be used to create constant analog voltages.

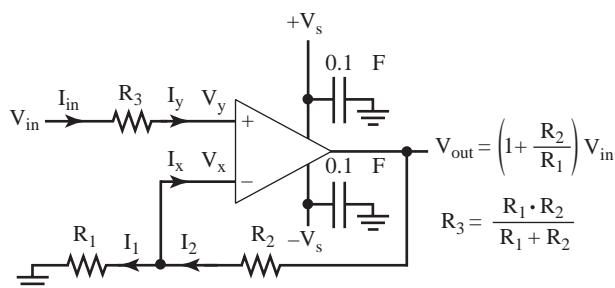
**Figure 11.12**  
Inverting amplifier with an effective  $-2.5$  to  $+2.5$  V analog signal range.



**Observation:** It is important in this scheme to separate the digital and analog grounds, avoiding direct connections between the two grounds.

The second linear mode circuit we will study is the noninverting amplifier, as shown in Figure 11.13. The gain is  $1 + R_2/R_1$ . The noninverting amplifier cannot have a gain less than 1. Just as with the inverting amp, the gain response is independent of  $R_3$ . So, we choose  $R_3$  to be the parallel combination  $R_1 \parallel R_2$  so that the effect of the bias currents is reduced.

**Figure 11.13**  
Noninverting amplifier.



Because  $I_y$  is zero,  $V_y$  equals  $V_{in}$ . Because of negative feedback,  $V_x$  equals  $V_y$ . Thus,  $V_x$  equals  $V_{in}$  too. Calculating currents we get  $I_1$  is  $V_{in}/R_1$  and  $I_2$  is  $(V_{out} - V_{in})/R_2$ . Because  $I_x$  is zero,  $I_1$  equals  $I_2$ . Setting  $I_1$  equal to  $I_2$  yields

$$V_{out} = (1 + R_2/R_1) \cdot V_{in}$$

Using the simple op amp rules,  $I_y$  is zero, hence the input impedance ( $Z_{in}$ ) of this circuit (defined as  $V_{in}/I_{in}$ ) would be infinite. In this situation we specify the amplifier input impedance to be the op amp input impedance. If the circuit were built with an OPA227 (which has a gain bandwidth product of 8 MHz), the bandwidth of this circuit would be 8 MHz divided by the gain.

**Checkpoint 11.7:** If the noninverting amplifier in Figure 11.13 were built with an OPA227, what would be the input impedance of the amplifier?

The calculation of the output impedance of this amp follows the same approach as the inverting amp,

$$Z_{out} = R_{out} \cdot (R_2 + R_1)/(K \cdot R_1)$$

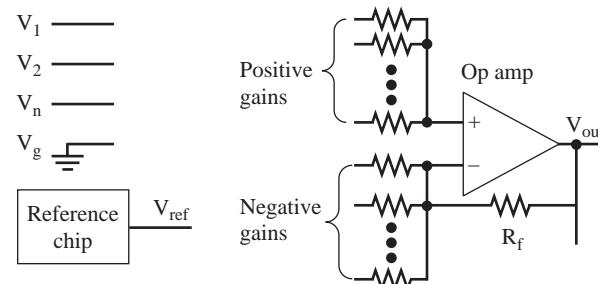
**Checkpoint 11.8:** List the design steps you would take to convert the noninverting amplifier, shown in Figure 11.13, to operate on analog signals varying from  $-2.5$  to  $2.5$  V using a single power supply.

This design example works with any analog circuit in the form of

$$V_{out} = A_1 V_1 + A_2 V_2 + \dots + A_n V_n + B$$

where  $A_1, A_2, \dots, A_n, B$  are constants and  $V_1, V_2, \dots, V_n$  are input voltages. The circuit will be designed with one op amp, beginning with the boiler plate shown in Figure 11.14.

**Figure 11.14**  
Boilerplate circuit model  
for linear circuit design.



The **first step** is to choose a reference voltage from available reference voltage chips, like those shown in Table 11.4. Some of the manufacturers that produce voltage references are Analog Devices, Burr-Brown, Linear Technology, Maxim, and National Semiconductor. The parameters to consider when choosing a voltage reference are voltage, package configuration, accuracy, temperature coefficient, and power. In particular, let  $V_{ref}$  be this reference voltage.

**Common error:** If you use a resistor divider from the power supply to create a voltage constant, then the power supply ripple will be added directly to your analog signal.

The **second step** is to rewrite the design equation in terms of the reference voltage,  $V_{ref}$ . In particular, we make  $A_{ref} = B/V_{ref}$ .

$$V_{out} = A_1 V_1 + A_2 V_2 + \dots + A_n V_n + A_{ref} V_{ref}$$

where  $A_1, A_2, \dots, A_n, A_{ref}$  are constants and  $V_1, V_2, \dots, V_n$  are input voltages.

The **third step** is to add a ground input to the equation. Ground is zero volts ( $V_g = 0$ ), but it is necessary to add this ground so that the sum of all the gains is equal to one.

$$V_{out} = A_1 V_1 + A_2 V_2 + \dots + A_n V_n + A_{ref} V_{ref} + A_g V_g$$

Choose  $A_g$  such that

$$A_1 + A_2 + \dots + A_n + A_{ref} + A_g = 1$$

In other words, let

$$A_g = 1 - (A_1 + A_2 + \dots + A_n + A_{ref})$$

The **fourth step** is to choose a feedback resistor,  $R_f$ , in the range of  $10\text{ k}\Omega$  to  $1\text{ M}\Omega$ . The larger the gains, the larger the value of  $R_f$  must be. Then calculate input resistors to create the desired gains. In particular,

$ A_1  = R_f/R_1$	so $R_1 = R_f/ A_1 $
$ A_2  = R_f/R_2$	so $R_2 = R_f/ A_2 $
$ A_n  = R_f/R_n$	so $R_n = R_f/ A_n $
$ A_{ref}  = R_f/R_{ref}$	so $R_{ref} = R_f/ A_{ref} $
$ A_g  = R_f/R_g$	so $R_g = R_f/ A_g $

**Observation:** We will get a simple solution if we choose the value of  $R_f$  to be a common multiple of the gains  $A_1, A_2, \dots, A_n, A_{ref}$ , and  $A_g$ .

The **last step** is to build the circuit. If the gain is positive, then the input resistor is connected to the positive terminal of the op amp. Conversely, if the gain is negative, then the input resistor is connected to the negative terminal of the op amp. The feedback resistor,  $R_f$ , will always be connected from the negative input to the output.

**Example 11.2** Design an analog circuit with a transfer function of  $V_{out} = 5V_1 - 3V_2 + 2V_3 - 10$ .

**Solution** The **first step** is to choose a reference voltage. The REF02 +5.00 V voltage reference will be used. The **second step** is to rewrite the design equation in terms of the reference voltage.

$$V_{out} = 5V_1 - 3V_2 + 2V_3 - 2V_{ref}$$

The **third step** is to add a ground input to the equation so that the sum of all the gains is equal to one.

$$V_{out} = 5V_1 - 3V_2 + 2V_3 - 2V_{ref} - V_g$$

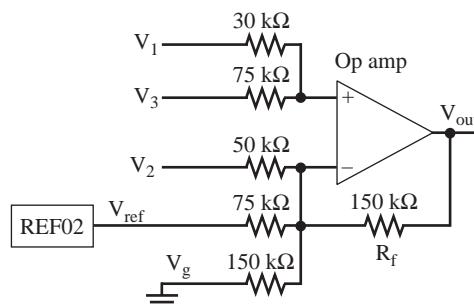
The **fourth step** is to choose a feedback resistor,  $R_f = 150\text{ k}\Omega$ . The value is a common multiple of the gains: 5, 3, 2, 1. Then calculate input resistors to create the desired gains.

$R_1 = R_f/ A_1  = 150\text{ k}\Omega/5 = 30\text{ k}\Omega$
$R_2 = R_f/ A_2  = 150\text{ k}\Omega/3 = 50\text{ k}\Omega$
$R_3 = R_f/ A_3  = 150\text{ k}\Omega/2 = 75\text{ k}\Omega$
$R_{ref} = R_f/ A_{ref}  = 150\text{ k}\Omega/2 = 75\text{ k}\Omega$
$R_g = R_f/ A_g  = 150\text{ k}\Omega/1 = 150\text{ k}\Omega$

The **last step** is to build the circuit, as shown in Figure 11.15. The positive gain inputs are connected to the plus input of the op amp and the negative gain inputs are connected to the minus input of the op amp input.

**Figure 11.15**

A linear op amp circuit.

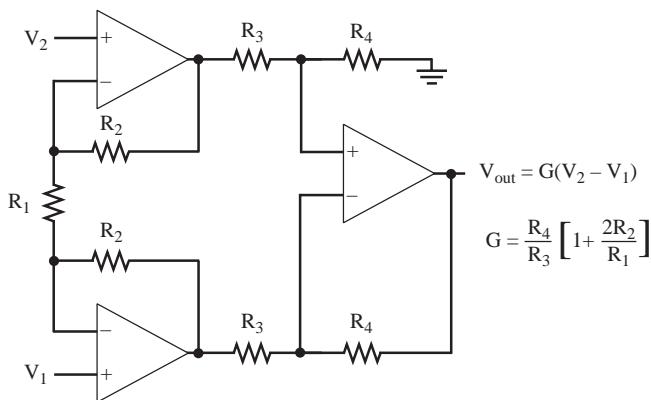


### 11.2.5 Instrumentation Amplifier

The *instrumentation amp* will amplify a differential voltage, shown in Figure 11.16 as  $V_2 - V_1$ . We use instrumentation amplifiers in applications that require a large gain (above 100), a high input impedance, and a good CMRR. One can be built using three high-quality op amps.

**Figure 11.16**

Instrumentation amplifier.



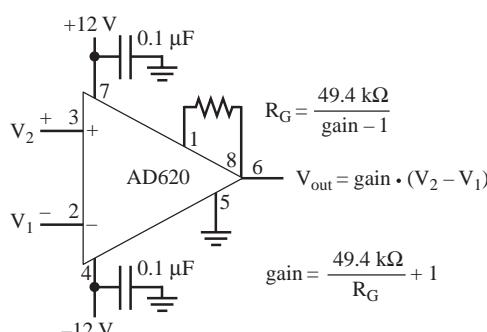
**Observation:** To achieve quality performance with a three-op amp instrumentation amplifier circuit, we must use precision resistors and quality op amps.

**Common error:** If you use a potentiometer in place of one of the  $R_3$  gain resistors in the Figure 11.16 circuit, then fluctuations in the potentiometer resistance that can occur with temperature, vibration, and time will have a strong effect on the amplifier gain.

Because of the range of applications that require instrumentation amplifiers, chip manufacturers have developed a variety of integrated solutions. In many cases we can achieve higher performance at reduced cost by utilizing one of these ICs. The gain is selected by external jumpers or external resistors. The Analog Devices AD620 is a typical low-cost device (\$6) (Figure 11.17). The MAX4460, MAX627, and INA122 are single-supply rail-to-rail instrumentation amps.

**Figure 11.17**

Integrated instrumentation amplifier.



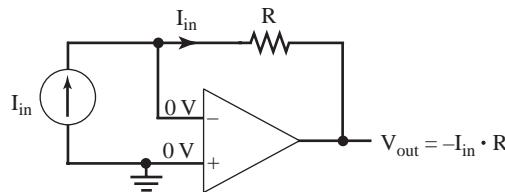
**Common error:** If you use a potentiometer as the  $R_G$  gain resistors in the Figure 11.17 circuit, then fluctuations in the potentiometer resistance that can occur with temperature, vibration, and time will have a strong effect on the amplifier gain.

**Checkpoint 11.9:** How do you build an instrumentation amp with a gain of 11?

### 11.2.6 Current-to-Voltage Circuit

In this circuit, the input is a current,  $I_{in}$ . Because no current enters the op amp terminal, this current will cross the feedback resistor  $R$ . Because of the negative feedback, the two op amp input terminals will be at approximately the same voltage, in this case zero. Therefore, the output voltage  $V_{out}$  will be  $-I_{in} \cdot R$  (Figure 11.18).

**Figure 11.18**  
Current-to-voltage converter.

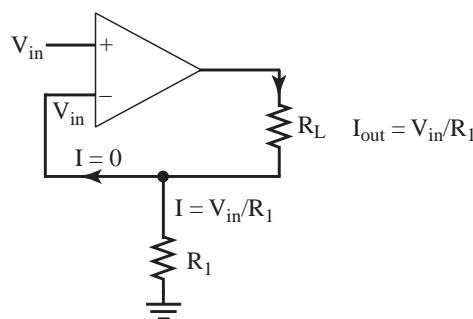


### 11.2.7 Voltage-to-Current Circuit

In this circuit, the input is a voltage,  $V_{in}$ . The resistance  $R_1$  is fixed and known (e.g.,  $10\text{ k}\Omega$ ), but the resistance  $R_L$  is external and unknown. The basic idea is that this circuit will deliver a constant current across  $R_L$ , independent of the value of  $R_L$ . Because of the negative feedback, the two op amp input terminals will be at approximately the same voltage, in this case  $V_{in}$ . The current across  $R_1$  will be  $V_{in}/R_1$ . Since no current enters the op amp terminal, this same current must also pass through  $R_L$  (Figure 11.19).

**Observation:** A precision reference voltage chip together with this voltage-to-current circuit creates a precision reference current system. In the application notes of the reference chips shown in Table 11.4 are many other circuits that create a precision reference current.

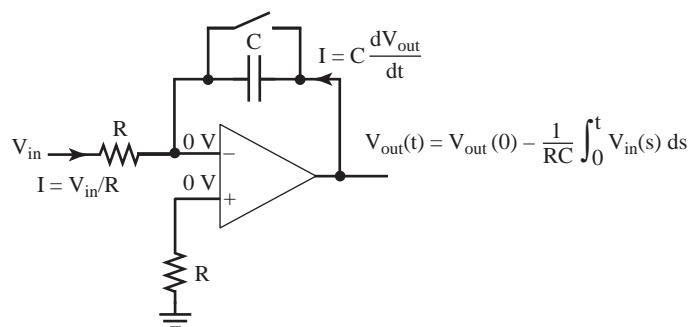
**Figure 11.19**  
Voltage-to-current converter.



### 11.2.8 Integrator Circuit

In this circuit, the input is a voltage,  $V_{in}$ . The basic idea of this circuit is to create a voltage output that is the integral of the voltage input. Often a computer-controlled switch is added to short the voltage across the capacitor, initializing the output voltage to zero. Because of the negative feedback, the two op amp input terminals will be at approximately the same voltage, in this case zero. The current across the input  $R$  will be  $V_{in}/R$ . Since no current enters the op amp terminal, this same current must also pass through the capacitor. The second  $R$  on the positive terminal was added to subtract the bias current of the op amp (Figure 11.20). A low-offset op amp (e.g., OP227) should be used to reduce the error caused by the offset.

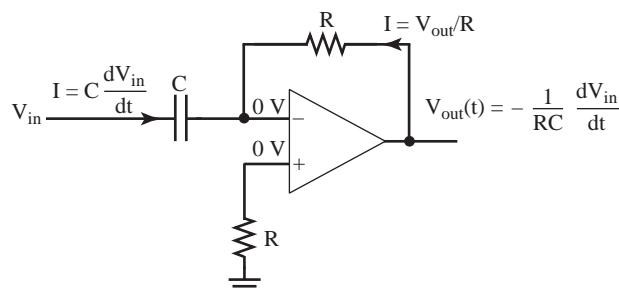
**Figure 11.20**  
Analog integrator circuit.



### 11.2.9 Derivative Circuit

In this circuit, the input is a voltage,  $V_{in}$ . The basic idea of this circuit is to create a voltage output that is the derivative of the voltage input. Because of the negative feedback, the two op amp input terminals will be at approximately the same voltage, in this case zero. The current across the input capacitor is related to the derivative of the input. Since no current enters the op amp terminal, this same current must also pass through the feedback resistor. The second R on the positive terminal was added to subtract the bias current of the op amp (Figure 11.21).

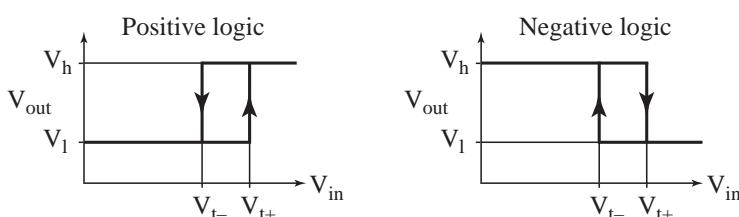
**Figure 11.21**  
Analog derivative circuit.



### 11.2.10 Voltage Comparators with Hysteresis

We can use a voltage comparator to detect events in an analog waveform. The range across the input voltage can vary and is usually determined by the analog supply voltages of the comparator. The output takes on two values, shown as  $V_h$  and  $V_l$  in Figure 11.22. A comparator with hysteresis has two thresholds,  $V_{t+}$  and  $V_{t-}$ . In both the positive- and negative-logic cases the threshold ( $V_{t+}$  or  $V_{t-}$ ) depends on the present value of the output. Hysteresis prevents small noise spikes from creating a false trigger.

**Figure 11.22**  
Input/output response of voltage converters with hysteresis.

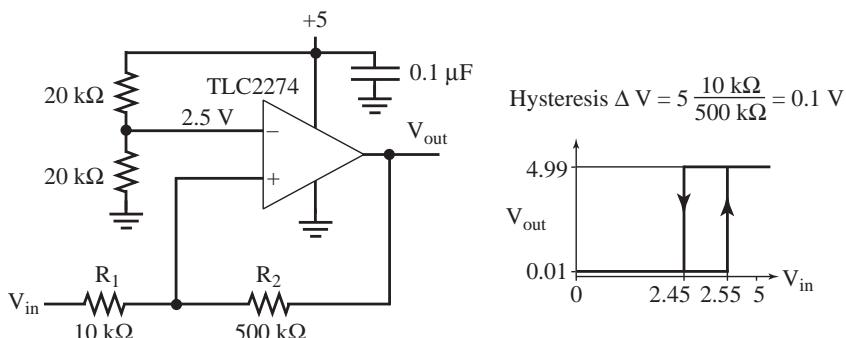


**Performance tip:** To eliminate false triggering, we select a hysteresis level ( $V_{t+} - V_{t-}$ ) greater than the noise level in the signal.

In this next circuit, a TLC2274 rail-to-rail op amp is used to design a voltage comparator (Figure 11.23). Since the output swings from 0.01 to 4.99 V, it can be connected directly to an input pin of the microcomputer. On the other hand, since +5 and 0 are used to power the

**Figure 11.23**

A voltage comparator with hysteresis using a rail-to-rail TLC2274.

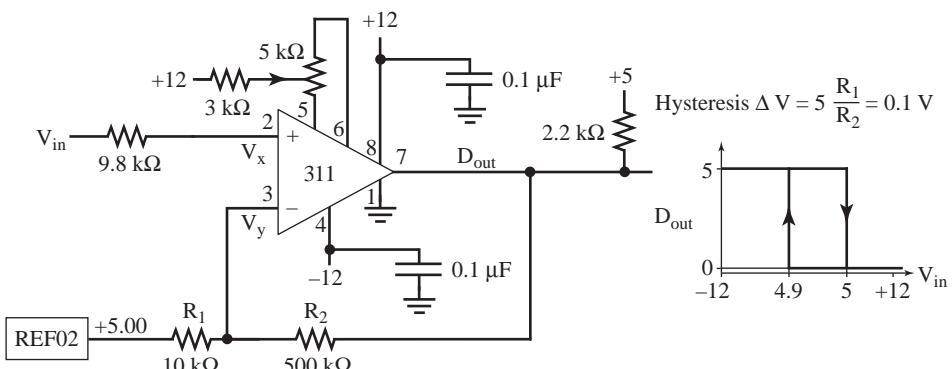


op amp, the analog input must remain in the 0 to +5 V range. The hysteresis level is determined by the  $R_2/R_1$  ratio. If the output is at 0 V, then the input goes above +2.55 before the positive terminal of the op amp reaches 2.5 V. Similarly, if the output is at +5 V, then the input goes below +2.45 before the positive terminal of the op amp falls below 2.5 V. In linear mode circuits, we should not use the supply voltage to create voltage references, but in a saturated mode circuit, power supply ripple will have little effect on the response.

There exists a wide range of integrated voltage comparators (Figure 11.24). They vary in price, speed, voltage offset, output configuration, and power consumption. The positive feedback is added to this LM311 circuit to produce hysteresis. If the output is at 0 V, then  $V_y$  is 4.9 V and the input goes below +4.9 to make the output rise. Similarly, if the output is at +5 V, then  $V_y$  is 5 V and the input goes after 5 V to make the output fall.

**Figure 11.24**

A voltage comparator with hysteresis using a LM311.

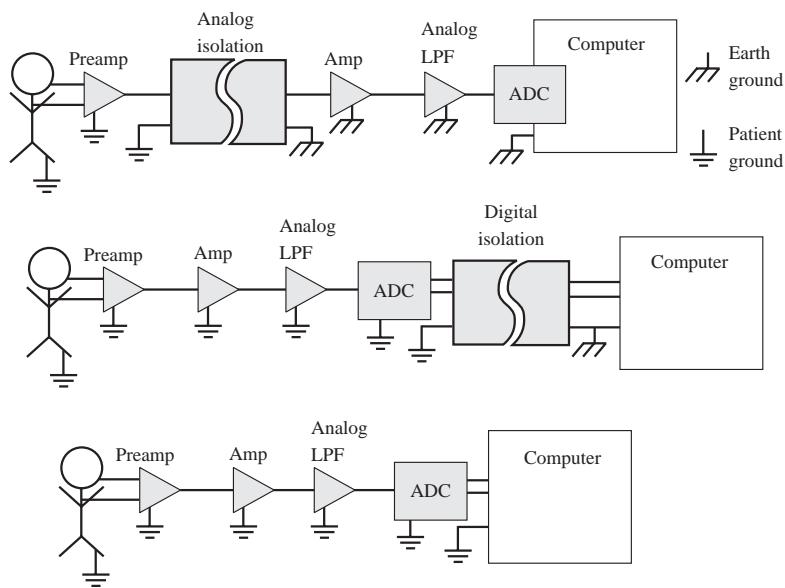


### 11.2.11 Analog Isolation

In some medical and industrial applications we need to design analog instrumentation that is isolated from earth ground. In an industrial setting, isolation is one way to reduce noise pickup from large EM fields produced by heavy machinery. In medical applications we need to protect the patient from potentially dangerous microshocks. Thus, the medical instrument must be isolated. There are three approaches to isolation, as shown in Figure 11.25. In the first approach, shown at the top of Figure 11.25, an analog isolation barrier is created between the preamp and the amp. This was the original approach used in analog instruments before the advent of mixed analog-digital systems. It is expensive and bulky and it introduces a very large transfer error. It is not appropriate for embedded applications that use a microcontroller. In the second approach, we use digital isolation. The 6N139 optical isolator, described previously in Chapter 8, is an effective low-cost mechanism to implement digital isolation. This is the most common approach used for new designs when a hard connection between the data acquisition system and building is required. It is fast, small,

**Figure 11.25**

Analog isolation, digital isolation, and battery-powered systems all provide protection from microshocks.



cheap, and will not introduce errors. The third approach runs the entire system with batteries. This is a very attractive approach due to the availability of high-quality low-power LCD displays and wireless networks, such as Bluetooth, ZigBee and 802.11b.

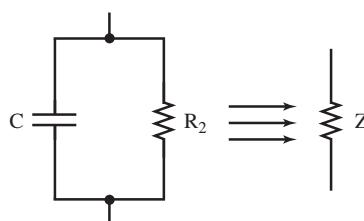
## 11.3 Analog Filters

**11.3.1 Simple Active Filter** We can add capacitors in parallel with the feedback resistor in the inverting amplifier to create a simple *one-pole low-pass filter* (Figure 11.26). The impedance of a resistor  $R_2$  in parallel with the capacitor  $C$  is a function of frequency.

$$Z = \frac{R_2}{1 + j\omega R_2 C}$$

**Figure 11.26**

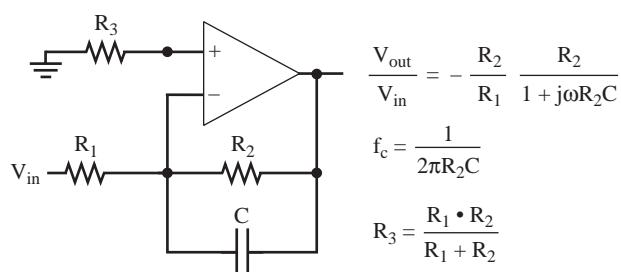
Complex impedance model of a resistor in parallel with a capacitor.



Therefore the gain of the circuit is  $-Z/R_1$ , which exhibits low-pass behavior. The *cutoff frequency* is defined to be the frequency at which the gain drops to 0.707 of its original value. In this simple low-pass filter, the cutoff frequency  $f_c$  is  $1/(2\pi R_2 C)$  (Figure 11.27).

**Figure 11.27**

One-pole low-pass analog filter.

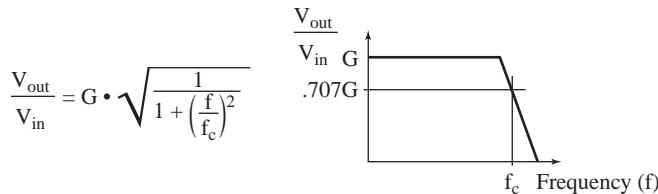


**Checkpoint 11.10:** What is the value of  $j$ ?

We classify this low-pass filter as one pole, because the transfer function has only one pair of poles in the  $s$  plane. One-pole low-pass filters have a gain versus frequency response as shown in Figure 11.28 (G is a constant).

**Figure 11.28**

Frequency response of a one-pole low-pass analog filter.



### 11.3.2 Butterworth Filters

Higher-order analog filters can be designed using multiple capacitors. One of the advantages of the *two-pole Butterworth analog filter* is that as long as the capacitors maintain the 2/1 ratio, the analog circuit will be a Butterworth filter. The design steps for the two-pole Butterworth low-pass filter are as follows (Figure 11.29):

1. Select the cutoff frequency  $f_c$
2. Divide the two capacitors by  $2\pi f_c$  (let  $C_{1A}$ ,  $C_{2A}$  be the new capacitor values)

$$C_{1A} = \frac{141.4 \mu\text{F}}{2\pi f_c} \quad C_{2A} = \frac{70.7 \mu\text{F}}{2\pi f_c}$$

3. Locate two standard-value capacitors (with the 2/1 ratio) with the same order of magnitude as the desired values; let  $C_{1B}$ ,  $C_{2B}$  be these standard value capacitors and let  $x$  be this convenience factor

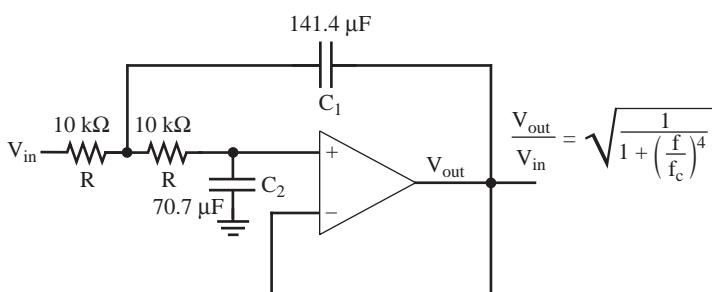
$$C_{1B} = \frac{C_{1A}}{x} \quad C_{2B} = \frac{C_{2A}}{x}$$

4. Adjust the resistors to maintain the cutoff frequency

$$R = 10 \text{ k}\Omega \cdot x$$

**Figure 11.29**

Two-pole Butterworth low-pass analog filter.



The analog filters in this section all require low-leakage, high-accuracy, and low-temperature coefficient capacitors like Teflon, polystyrene, or polypropylene. Lower-quality (lower cost) Mylar or ceramic capacitors could be used if filter accuracy were less important. This design process is implemented as file LPF.xls on the CD.

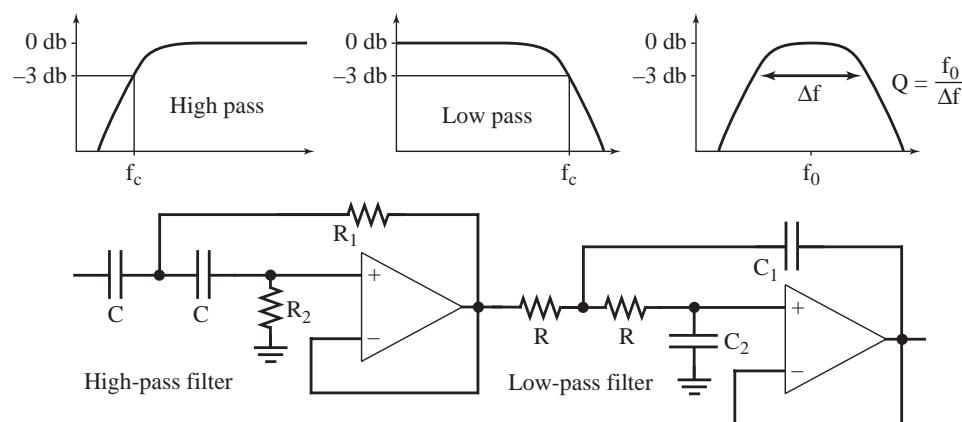
**Performance tip:** If you choose standard-value resistors near the desired values, you will save money and the circuit will still be a Butterworth filter. The only difference is that the cutoff frequency will be slightly off from the original specification.

### 11.3.3 Bandpass and Band-Reject Filters

Low-pass and high-pass filters can be cascaded to build a bandpass filter (Figure 11.30). Normally we put the high-pass filter first so that the low-pass filter can remove high-frequency noise generated in both stages.

**Figure 11.30**

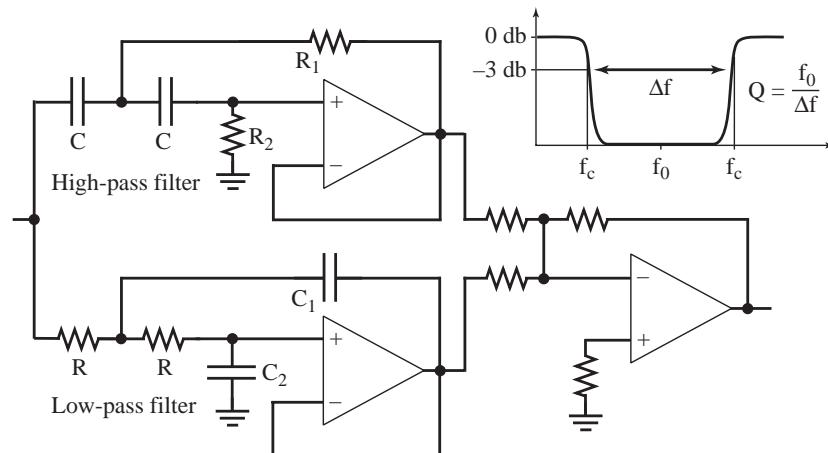
Cascade approach to bandpass analog filter design.



This cascade filter should only be used for low-Q applications. Similarly, a band-reject filter can be constructed by running both filters in parallel and summing the output (Figure 11.31). The parallel band-reject filter also should be used only for low-Q applications. One implementation of a high-Q bandpass (frequency select) filter is called the multiple feedback bandpass filter (Figure 11.32). The Q is defined as the center frequency

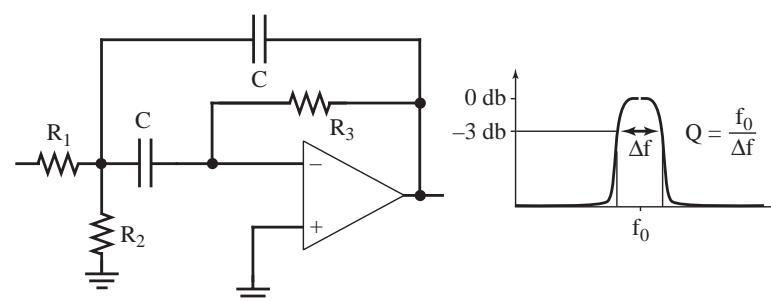
**Figure 11.31**

Parallel approach to band-reject analog filter design.



**Figure 11.32**

Multiple feedback bandpass filter.



divided by the bandpass range,  $f_0/\Delta f$ . The two capacitors have the same value and are typically in the range from 0.001 to 10  $\mu\text{F}$ . The design steps are:

1. Select a convenience capacitance value for the two capacitors
2. Calculate the three resistor values for  $x = 1/(2\pi f_0 C)$

$$R_1 = Q \cdot x \quad R_2 = x/(2Q - 1/Q) \quad R_3 = 2 \cdot Q \cdot x$$

The resistors should be in the range from 5  $\text{k}\Omega$  to 5  $\text{M}\Omega$ . If not, then repeat the design steps with a different capacitance value.

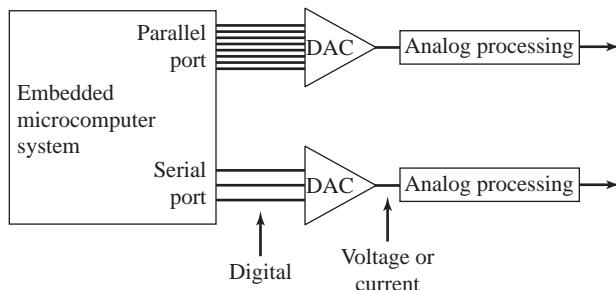
**Performance tip:** Many analog IC manufacturers provide design tools. FilterPro is a free design tool from Texas Instruments you can use to design analog filters ([www.ti.com](http://www.ti.com)).

## 11.4 Digital-to-Analog Converters

### 11.4.1 DAC Parameters

A DAC converts digital signals into analog form (Figure 11.33). A microcomputer output can be connected to a DAC. Many DACs can be interfaced to the SPI synchronous serial port. The DAC output can be current or voltage. Additional analog processing may be required to filter, amplify, or modulate the signal. We will also use DACs to design variable-gain or variable-offset analog circuits.

**Figure 11.33**  
DACs provide analog output for our embedded microcomputer systems.



The DAC *precision* is the number of distinguishable DAC outputs (e.g., 256 alternatives, 8 bits). The DAC *range* is the maximum and minimum DAC output (volts, amperes). The DAC *resolution* is the smallest distinguishable change in output. The units of resolution are in volts or amperes depending on whether the output is voltage or current. The resolution is the change that occurs when the digital input changes by 1.

$$\text{Range (volts)} = \text{Precision (alternatives)} \cdot \text{Resolution (volts)}$$

The DAC *accuracy* is  $(\text{actual} - \text{ideal}) / \text{ideal}$ , where ideal is referred to the National Bureau of Standards (NBS). There are two common 8-bit encoding schemes for a DAC:

$$\text{Unsigned: } V_{\text{out}} = V_{\text{fs}} \left( -\frac{b_7}{2} + \frac{b_6}{4} + \frac{b_5}{8} + \frac{b_4}{16} + \frac{b_3}{32} + \frac{b_2}{64} + \frac{b_1}{128} + \frac{b_0}{256} \right) + V_{\text{os}}$$

$$\text{Signed 2s complement: } V_{\text{out}} = V_{\text{fs}} \left( -\frac{b_7}{2} + \frac{b_6}{4} + \frac{b_5}{8} + \frac{b_4}{16} + \frac{b_3}{32} + \frac{b_2}{64} + \frac{b_1}{128} + \frac{b_0}{256} \right) + V_{\text{os}}$$

where  $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$  are the 8-bit digital inputs,  $V_{\text{fs}}$  is the full-scale voltage (typically +5 or +10), and  $V_{\text{os}}$  is the offset voltage (hopefully 0).

One can choose the full-scale range of the DAC to simplify the use of fixed-point mathematics. For example, if an 8-bit DAC had a full-scale range of 0 to 2.55 V, then the resolution would be exactly 10 mV. This means that if the DAC digital input were  $123_{10}$ , then the DAC output voltage would be 1.23 V.

The following discussion will focus on a 3-bit DAC with a range of 0 to +7 V. This DAC can be used to generate a variable voltage  $V_{out}$ , a variable offset, or a variable gain  $V_{out}/V_{in}$  (Figure 11.34). A DAC gain error is a shift in the slope of the  $V_{out}$  versus  $V_{in}$  static response (Figure 11.35). A DAC offset error is a shift in the  $V_{out}$  versus  $V_{in}$  static response. The DAC transient response (Figure 11.36) has three components:

- Delay phase
- Slewing phase
- Ringing phase

For purposes of linearity, let  $m, n$  be digital inputs, and let  $f(n)$  be the analog output of the DAC. Let  $\Delta$  be the DAC resolution. The DAC is linear if

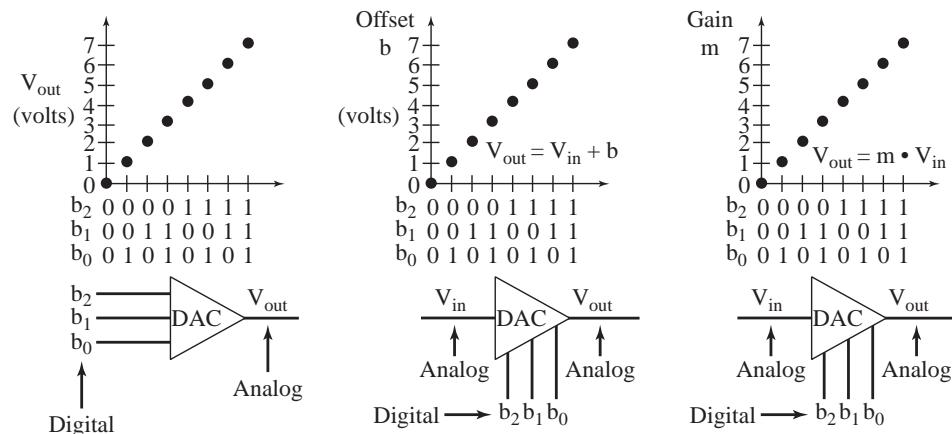
$$f(n+1) - f(n) = f(m+1) - f(m) = \Delta \quad \text{for all } n, m$$

The DAC is monotonic (Figure 11.64) if

$$\text{sign}[f(n+1) - f(n)] = \text{sign}[f(m+1) - f(m)] \quad \text{for all } n, m$$

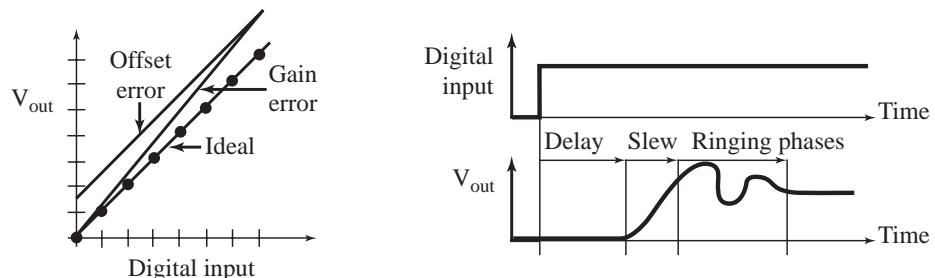
**Figure 11.34**

Three-bit DAC used to generate a variable output, a variable offset, or a variable gain.



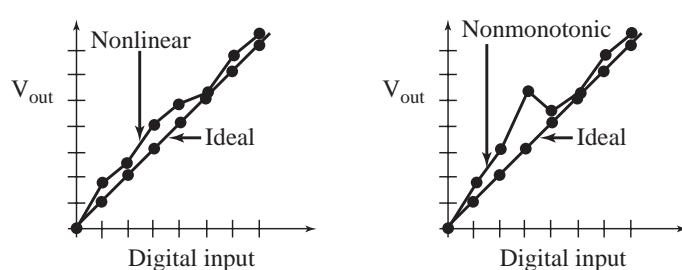
**Figure 11.35**

Static and dynamic performance measures of DACs.



**Figure 11.36**

Nonlinear and nonmonotonic DACs.



Conversely, the DAC is nonlinear if

$$f(n+1) - f(n) \neq f(m+1) - f(m) \quad \text{for some } m, n$$

Practically speaking all DACs are nonlinear, but the worst nonlinearity is nonmonotonicity. Table 11.5 lists sources of error. The DAC is nonmonotonic if

$$\text{sign}[f(n+1) - f(n)] \neq \text{sign}[f(m+1) - f(m)] \quad \text{for some } n, m$$

**Table 11.5**

Sources and solutions to DAC errors.

Errors can be due to	Solutions
Incorrect resistor values	Precision resistors with low tolerances
Drift in resistor values	Precision resistors with good temperature coefficients
White noise	Reduce BW using a low-pass filter, reduce temperature
Op amp errors	Use more expensive devices with low noise and low drift
Interference from external fields	Shielding, ground planes

### 11.4.2 DAC Using a Summing Amplifier

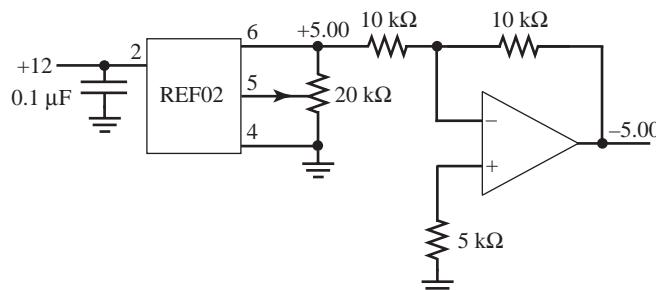
We will need a  $-5.00$ -V reference, which we can create using an inverting op amp and voltage reference (Figure 11.37). We use the summing mode of another op amp to build the DAC. The exponential basis elements of the DAC are created by the resistance values  $25\text{ k}\Omega$ ,  $50\text{ k}\Omega$ , and  $100\text{ k}\Omega$ . This method can only be used to build low-precision DACs, because it is difficult to extend the exponential resistance network beyond 5 or 6 bits. In the DAC circuit of Figure 11.38, the SW02 switch is closed ( $75\text{ }\Omega$ ) when the digital input (IN) is high, and the SW02 switch is open when the digital input is low. The output is  $V_{\text{out}} = 4b_2 + 2b_1 + b_0$ .

The range of the DAC is 0 to  $+7\text{ V}$  with a resolution of 1 V.

$$V_{\text{out}} = \sum_{n=0}^2 b_n 2^n$$

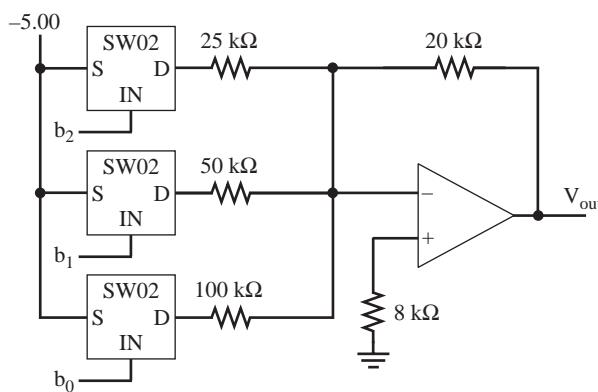
**Figure 11.37**

Negative reference that will be used in the DAC design.



**Figure 11.38**

Three-bit unsigned summing DAC.



The operation of this 3-bit DAC can be summarized using the basis elements 1, 2, 4. The responses of these basis elements are presented in Table 11.6.

**Table 11.6**

Responses of the basis elements for an unsigned 3-bit DAC.

<b>b<sub>2</sub></b>	<b>b<sub>1</sub></b>	<b>b<sub>0</sub></b>	<b>V<sub>out</sub></b>
0	0	1	+1
0	1	0	+2
1	0	0	+4

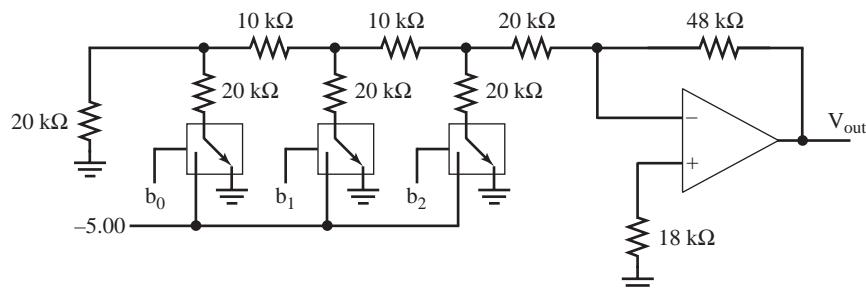
### 11.4.3

#### Three-Bit DAC with an R-2R Ladder

It is not feasible to construct a high-precision DAC using the summing op amp technique for two reasons. First, if one chooses the resistor values from the practical  $10\text{ k}\Omega$  to  $1\text{ M}\Omega$  range, then the maximum precision would be  $1\text{ M}\Omega/10\text{ k}\Omega = 100$ , or about 7 bits. The second problem is that it would be difficult to avoid nonmonotonicity because a small percentage change in the small resistor (e.g., the one causing the largest gain) would overwhelm the effects of the large resistor (e.g., the one causing the smallest gain). To address both these limitations, the R-2R ladder is used (Figure 11.39). It is practical to build resistor packages such that all the Rs are equal and all the 2Rs are two times the Rs. Resistance errors will change all resistors equally. This type of error affects the slope  $V_{fs}$  but not the linearity or the monotonicity.

**Figure 11.39**

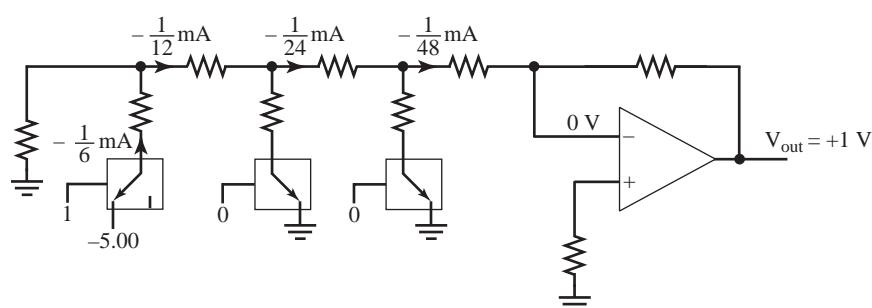
Three-bit unsigned R-2R DAC.



To analyze this circuit we will consider the three basis elements (1, 2, 4). The unsigned basis elements were presented earlier in Table 11.6. If these three cases are demonstrated, then the law of superposition guarantees the other five will work. Notice that these analog switches are “three-pole” and are fundamentally different from the switches in the summing DAC. When one of the digital inputs is true then  $-5.00\text{ V}$  is connected to the R-2R ladder, and when the digital input is false, then the connection is grounded. In each of the three test cases, the current across the active switch is  $-1/6\text{ mA}$ . This current is divided by 2 at each branch point. Current injected away from the op amp will be divided more times. Since each stage divides by 2, the exponential behavior is produced. An actual DAC is implemented with a current switch rather than a voltage switch as shown in Figure 11.40. Nevertheless,

**Figure 11.40**

The DAC output is  $+1\text{ V}$ , then the digital input is 001.



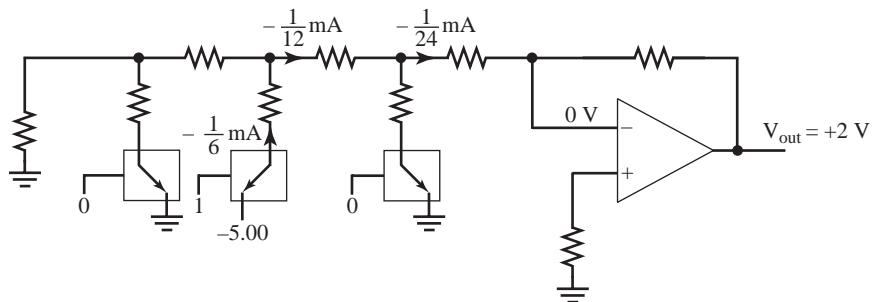
this simple circuit illustrates the operation of the R-2R ladder function. The output voltage is a linear combination of the three digital inputs,  $V_{out} = 4b_2 + 2b_1 + b_0$ . To increase the precision one simply adds more stages to the R-2R ladder. One can purchase 16-bit DACs that utilize the R-2R ladder technique.

When the input is 001,  $-5.00\text{ V}$  is presented to the left. The effective impedance to ground is  $3R$  ( $30\text{ k}\Omega$ ), so the current injected into the R-2R ladder is  $-1/6\text{ mA}$ . The current is divided three times, and  $-1/48\text{ mA}$  goes across the  $48\text{-k}\Omega$  resistor to create the  $+1\text{-V}$  output (Figure 11.40).

When the input is 010,  $-5.00\text{ V}$  is presented in the middle. The effective impedance to ground is again  $3R$  ( $30\text{ k}\Omega$ ), so the current injected into the R-2R ladder is  $-1/6\text{ mA}$ . The current is divided twice, and  $-1/24\text{ mA}$  goes across the  $48\text{-k}\Omega$  resistor to create the  $+2\text{-V}$  output (Figure 11.41).

**Figure 11.41**

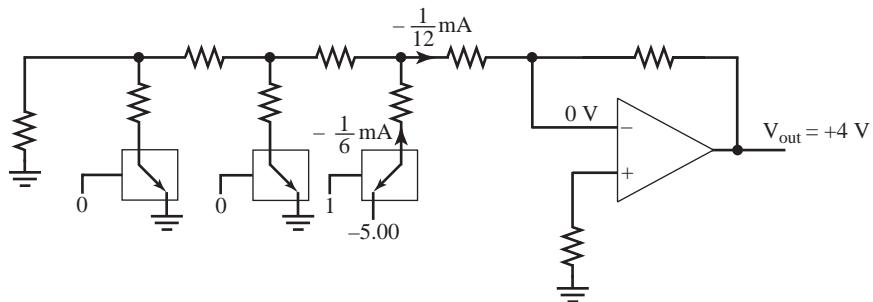
The DAC output is  $+2\text{ V}$ , then the digital input is 010.



When the input is 100,  $-5.00\text{ V}$  is presented on the right. The effective impedance to ground is again  $3R$  ( $30\text{ k}\Omega$ ), so the current injected into the R-2R ladder is  $-1/6\text{ mA}$ . This time the current is divided only once, and  $-1/12\text{ mA}$  goes across the  $48\text{-k}\Omega$  resistor to create the  $+4\text{-V}$  output (Figure 11.42).

**Figure 11.42**

The DAC output is  $+4\text{ V}$ , then the digital input is 100.

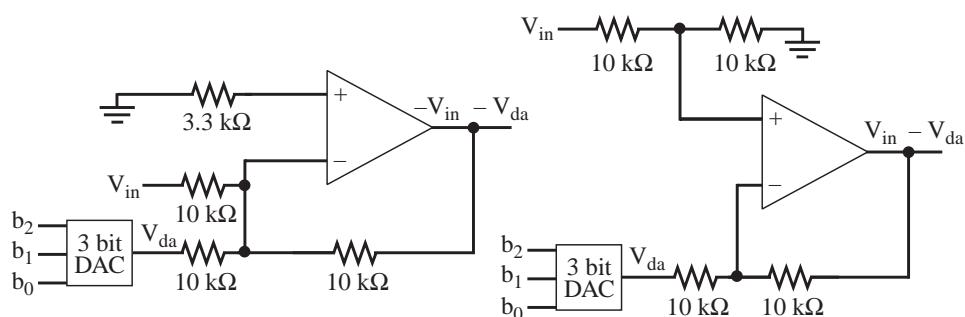


One of the reasons this simple 3-bit DAC is introduced is to illustrate the use of DACs in the design of variable-gain and variable-offset analog amplifiers. In this way, the software has control over the gain and offset of the analog circuit. The variable-offset approach simply connects the DAC output into one of the inputs of an analog adder or analog subtractor, as illustrated in Figure 11.43.

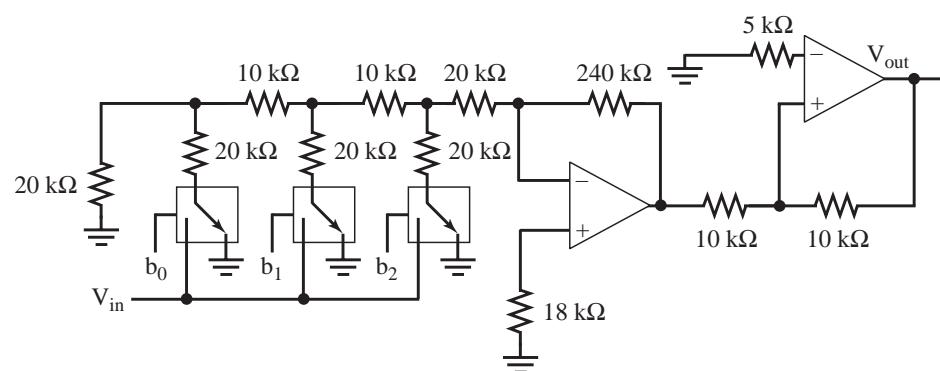
A multiplying DAC is one that allows a voltage input to be connected to the R-2R ladder instead of a voltage reference. If you connect a voltage reference to this pin, the DAC operates in the usual fashion. On the other hand, if this pin is an analog input, then the DAC is a variable-gain amplifier. Using the same 3-bit DAC circuit developed earlier, we can build a variable-gain amplifier. An inverting amplifier is added at the end to create positive-gain amplification. The  $48\text{-k}\Omega$  resistor is replaced with a  $240\text{-k}\Omega$  one so that the gain varies from 0 to  $+7$  (Figure 11.44).

**Figure 11.43**

Variable-offset analog circuit using a 3-bit DAC.

**Figure 11.44**

Variable-gain analog circuit using a 3-bit multiplying DAC.



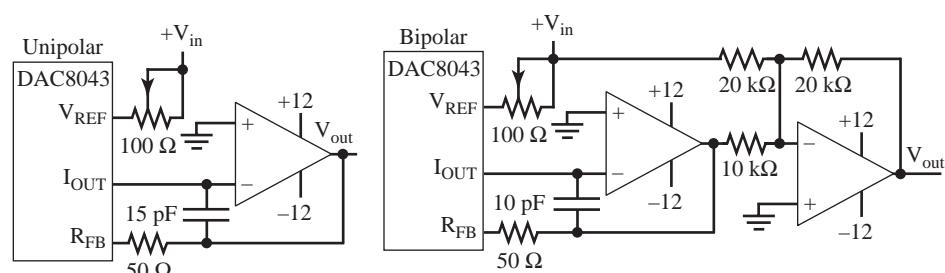
#### 11.4.4

#### Twelve-Bit DAC with a DAC8043

The synchronous serial interface between the computer and the Analog Devices DAC8043 DAC was introduced previously in Section 7.6. Here in this section we will focus on the analog aspects of the circuit. If the DAC is used in the regular digital input/analog output mode, then  $V_{in}$  will be a reference voltage (e.g., +5.00 or +10.00 V). If the DAC is used in variable-gain mode (multiplying DAC), then  $V_{in}$  is the analog input (i.e.,  $-10 \leq V_{in} \leq +10$  V). The optional 100 Ω potentiometer is used to adjust the gain (Figure 11.45).

**Figure 11.45**

Unipolar and bipolar modes of the DAC8043 12-bit DAC.



The  $V_{out}$  in Table 11.7 is calculated for a  $V_{in} = +5.00$ -V reference voltage. The bipolar code is called *offset binary*. To convert to regular 2s complement we simply complement the most significant digital bit. When the unipolar circuit is used in variable-gain mode, it is called two-quadrant because the input ( $V_{in}$ ) can be positive or negative, but the gain is always negative. When the bipolar circuit is used in variable-gain mode, it is called four-quadrant because the input ( $V_{in}$ ) can be positive or negative, and the gain can be positive or negative.

Digital input	Unipolar $V_{out}$ (V)	Bipolar $V_{out}$ (V)	Unipolar gain	Bipolar gain
1111,1111,1111	-4.999	4.998	$-\frac{4095}{4096}$	$+\frac{2047}{2048}$
1000,0000,0001	-2.501	0.002	$-\frac{2049}{4096}$	$+\frac{1}{2048}$
1000,0000,0000	-2.500	0.000	$-\frac{2048}{4096}$	$+\frac{0}{2048}$
0111,1111,1111	-2.499	-0.002	$-\frac{2047}{4096}$	$-\frac{1}{2048}$
0000,0000,0001	-0.001	-4.998	$-\frac{1}{4096}$	
0000,0000,0000	0.000	-5.000	$-\frac{0}{4096}$	$-\frac{2048}{2048}$

**Table 11.7**

The 12-bit DAC8043 DAC can create an analog output or be used as a variable-gain amplifier.

### 11.4.5 DAC Selection

Many manufacturers, like Analog Devices, Burr Brown, Motorola, Sipex, and Maxim, produce DACs. These DACs have a wide range of performance parameters and come in many configurations. The following paragraphs discuss the various issues to consider when selecting a DAC. Although we assume the DAC is used to generate an analog waveform, these considerations will generally apply to most DAC applications.

*Precision/range/resolution.* These three parameters affect the quality of the signal that can be generated by the system. The more bits in the DAC, the finer the control the system has over the waveform it creates. As important as this parameter is, it is one of the more difficult specifications to establish a priori. A simple experimental procedure to address this question is to design a prototype system with a very high precision (e.g., 12, 14, or 16 bits). The software can be modified to use only some of the available precision. For example, the 12-bit DAC8043 hardware developed in Sections 7.6 and 11.4.4 can be reduced to 8 or 10 bits using the functions shown in Program 11.1. The bottom bits are set to zero, instead of shifting, so that the rest of the system will operate without change.

#### Program 11.1

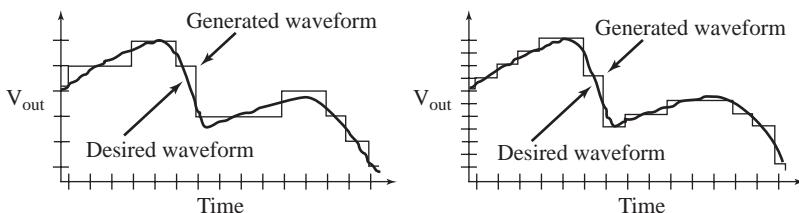
Software used to test how many bits are really needed.

```
void DACout8(unsigned short code){
    DACout(code&0xFFFF); // ignore bottom 4 bits
}
void DACout10(unsigned short code){
    DACout(code&0xFFFC); // ignore bottom 2 bits
}
```

The three versions of the software (e.g., 8-, 10-, 12-bit DAC) are used to see experimentally the effect of DAC precision on the overall system performance. Figure 11.46 illustrates how DAC precision affects the quality of the generated waveform.

**Figure 11.46**

The waveform on the right was created by a DAC with one more bit than the left.



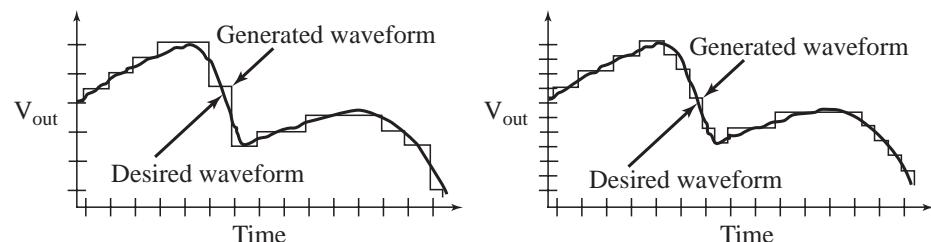
**Channels.** Even though multiple channels could be implemented using multiple DAC chips, it is usually more efficient to design a multiple-channel system using a multiple-channel DAC. Some advantages of using a DAC with more channels than originally conceived are future expansion, automated calibration, and automated testing.

**Configuration.** DACs can have voltage or current outputs. Current-output DACs can be used in a wide spectrum of applications (e.g., adding gain and filtering) but do require external components. DACs can have internal or external references. An internal-reference DAC is easier to use for standard digital input/analog output applications, but the external-reference DAC can be used often in variable-gain applications (multiplying DAC). As we saw with the DAC8043, sometimes the DAC generates a unipolar output, while other times the DAC produces bipolar outputs.

**Speed.** There are a couple of parameters manufacturers use to specify the dynamic behavior of the DAC. The most common is settling time; another is maximum output rate. When operating the DAC in variable-gain mode, we are also interested in the gain/BW product of the analog amplifier. When comparing specifications reported by different manufacturers, it is important to consider the exact situation used to collect the parameter. In other words, one manufacturer may define settling time as the time to reach 0.1% of the final output after a full-scale change in input given a certain load on the output, while another manufacturer may define settling time as the time to reach 1% of the final output after a 1-V change in input under a different load. The speed of the DAC together with the speed of the computer/software will determine the effective frequency components in the generated waveforms. Both the software (rate at which the software outputs new values to the DAC) and the DAC speed must be fast enough for the given application. In other words, if the software outputs new values to the DAC at a rate faster than the DAC can respond, then errors will occur. Figure 11.47 illustrates the effect of DAC output rate on the quality of the generated waveform. We will learn in Chapter 12 that the Nyquist theorem states that to prevent error, the digital data rate must be greater than twice the maximum frequency component of the desired analog waveform.

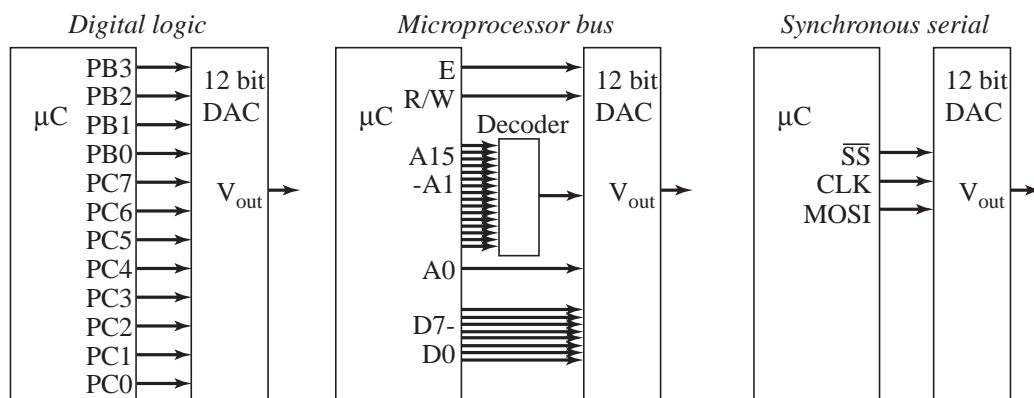
**Figure 11.47**

The waveform on the right was created by a system with twice the output rate than the left.



**Power.** There are three power issues to consider. The first consideration is the type of power required. Some devices require three power voltages (e.g., +5, +12, and -12 V), while many of the newer devices will operate on a single voltage supply (e.g., +2.7, +3.3, +5, or +12 V). If a single supply can be used to power all the digital and analog components, then the overall system costs will be reduced. The second consideration is the amount of power required. Some devices can operate on less than 1 mW and are appropriate for battery-operated systems or for systems where excess heat is a problem. The last consideration is the need for a low-power sleep mode. Some battery-operated systems need the DAC only intermittently. In these applications, we wish to give a shutdown command to the DAC so that it won't draw much current when the DAC is not needed.

**Interface.** Three approaches exist for interfacing the DAC to the computer (Figure 11.48). In a digital logic interface, the individual data bits are connected to a dedicated computer

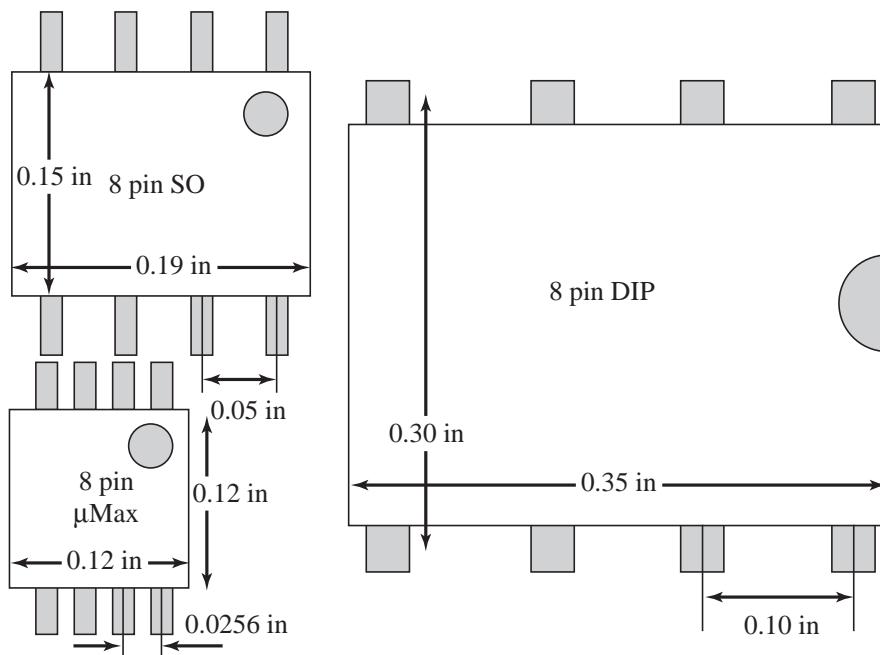
**Figure 11.48**

Three approaches to interfacing a 12-bit DAC to the microcomputer.

output port. For example, a 12-bit DAC requires 12-bit output port bits to interface. The software simply writes to the parallel port(s) to change the DAC output. The second approach is called  $\mu$ P-bus or microprocessor-compatible. These devices are intended to be interfaced onto the address/data bus of an expanded-mode microcomputer. An example of this type of interface can be found in Exercise 9.18. The SPI/DAC8043 interface is an example of the last type of interface. This approach requires the fewest number of I/O pins. Even if the microcomputer does not support the SPI directly, these devices can be interfaced to regular I/O pins via the bit-banging software approach.

**Package.** The standard DIP is convenient for creating and testing an original prototype. On the other hand, surface-mount packages like the SO and  $\mu$ Max require much less board space. Because surface-mount packages do not require holes in the printed circuit board, circuits with these devices are easier/cheaper to manufacture (Figure 11.49).

**Figure 11.49**  
Integrated circuits come  
in a variety of packages.



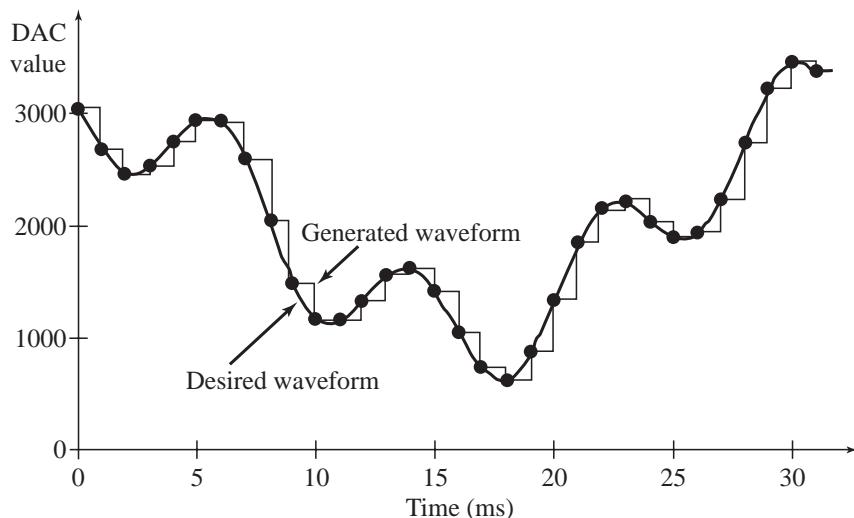
**Cost.** Cost is always a factor in engineering design. Besides the direct costs of the individual components in the DAC interface, other considerations that affect cost include (1) power supply requirements, (2) manufacturing costs, (3) the labor involved in individual calibration if required, and (4) software development costs.

### 11.4.6 DAC Waveform Generation

One application that requires a DAC is waveform generation. In this section, we will discuss various software methods for creating analog waveforms with a DAC. In each case, we will be using the DAC8043 hardware/software interface introduced in Sections 7.6 and 11.4.4. In addition, we will use an output compare interrupt for the timing so that the waveform generation occurs in the background. The rituals for initializing the periodic interrupt are shown in Section 6.2. To get a fair comparison between the various methods, each implementation will generate 32 interrupts per waveform.

In the first approach, we assume there exists a time-to-voltage function, called `wave()`, which we can call to determine the next DAC value to output. For example, the waveform in Figure 11.50 could be generated by Program 11.2. The simplest solution generates an output compare interrupt at a regular rate. The advantage of this approach is that complex waveforms can be encoded with a small amount of data. In this particular example, the entire waveform can be stored as five data points (2048.0, 1000.0, 31.25, -500.0, 125.0). The disadvantage of this technique is that not all waveforms have a simple function, and this software will run slower as compared to the other techniques (Program 11.3).

**Figure 11.50**  
Generated waveform using either the function or the table look-up.



**Program 11.2**  
Periodic interrupt used to create the analog output waveform.

```
unsigned short wave(unsigned short t){
    float result,time;
    time=2*pi*((float)t)/1000.0;
    // integer t in msec into floating point time in seconds
    result=2048.0+1000.0*cos(31.25*time)-500.0*sin(125.0*time);
    return (unsigned short) result; }
```

**Program 11.3**  
Periodic interrupt used to create the analog output waveform.

```
unsigned short Time; // every 1ms
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20;      // ack C5F
    TC5 = TC5+1000;    // Executed every 1 ms
    Time++;
    DAC_out(wave(Time)); }
```

In the second approach, we put the waveform information in a large statically allocated global variable. Every interrupt we fetch a new value out of the data structure and output it to the DAC. In this case the output compare interrupt also occurs at a regular rate. Assume the ritual initializes  $I=0$ . This waveform could be defined as shown in Programs 11.4 and 11.5.

#### Program 11.4

Simple data structure for the waveform.

```
unsigned short I; // incremented every 1ms
const unsigned short wave[32] = {
    3048, 2675, 2472, 2526, 2755, 2957, 2931, 2597,
    2048, 1499, 1165, 1139, 1341, 1570, 1624, 1421,
    1048, 714, 624, 863, 1341, 1846, 2165, 2206, 2048,
    1890, 1931, 2250, 2755, 3233, 3472, 3382};
```

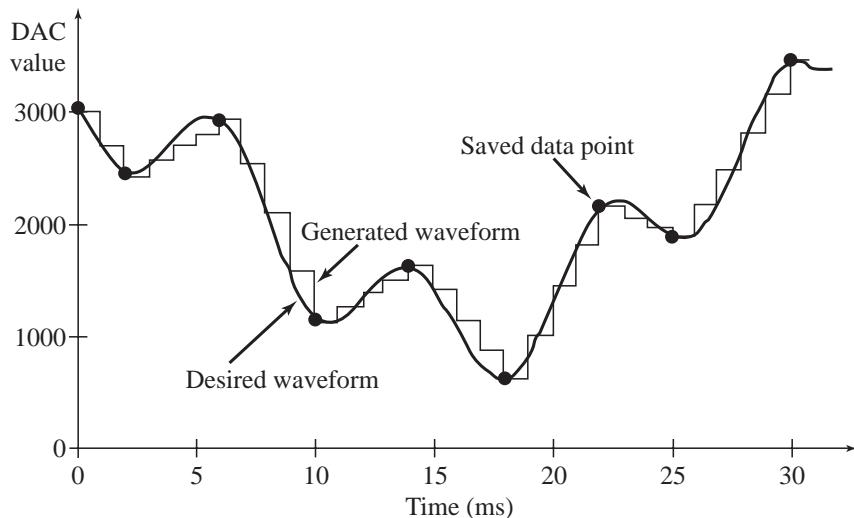
#### Program 11.5

Periodic interrupt used to create the analog output waveform.

```
unsigned short Time; // every 1ms
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20; // ack C5F
    TC5 = TC5+1000; // Executed every 1 ms
    if((++I)==32) I=0;
    DAC_out(wave[I]);}
```

Since the output rate is equal and fixed, these first two methods have the same performance as illustrated in Figure 11.50. The thick line is the desired waveform, and the thinner line is the actual generated curve. If the size of the table gets large, it is possible to store a smaller table in memory and use linear interpolation to recover the data points between the stored samples. Figure 11.51 shows the generated waveform derived from only 9 of the original 32 data points. To simplify the software, the first data point is repeated as the last data point. For each point we will need to save both the DAC value and time length of the current-line segment. For the 9 saved data points we simply output the data, but for the other points, we must perform a linear interpolation to get the value to output to the DAC.

**Figure 11.51**  
Generated waveform using a short table with linear interpolation.



Assume the ritual initializes  $I=J=0$ . Signed 16-bit numbers are used so that the subtractions operate properly. In other words, some of the intermediate calculations can be negative. This waveform could be defined as shown in Programs 11.6 and 11.7.

```

short I; // incremented every 1ms
short J; // index into these two tables
const short t[10] = {0,2,6,10,14,18,22,25,30,32}; // time in msec
const short wave[10]={3048,2472,2931,1165,1624,624,2165,1890,3472,3048}; //last=first

```

**Program 11.6**

Data structure with time and value for the waveform.

```

unsigned short Time; // every 1ms
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20; // ack C5F
    TC5 = TC5+1000; // Executed every 1 ms
    if((++I)==32) {I=0; J=0;}
    if(I==t[J])
        DAC_out(wave[J]);
    else if (I==t[J+1]){
        J++;
        DAC_out(wave[J]);}
    else
        DAC_out(wave[J]+((wave[J+1]-wave[J])*(I-t[J]))/(t[J+1]-t[J]));
}

```

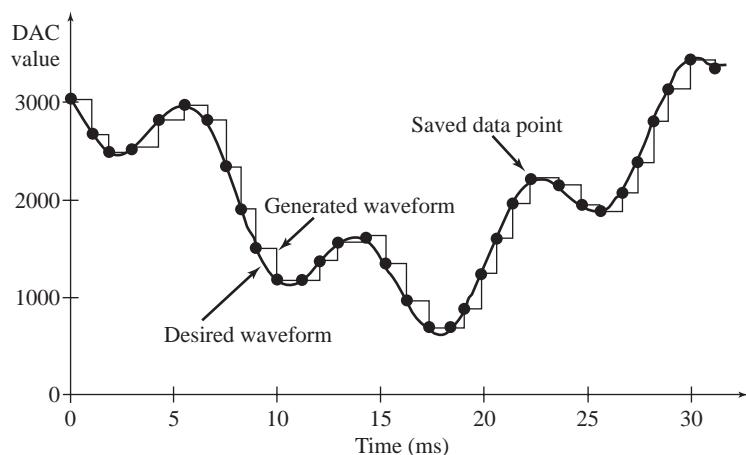
**Program 11.7**

Periodic interrupt used to create the analog output waveform.

The software in the previous techniques changes the DAC at a fixed rate. While this is adequate most of the time, there are some waveforms for which an uneven time between outputs seems appropriate. In our test signal, there are phases of the wave where the signal varies slowly while there are phases where the signal changes. Notice the data points in Figure 11.52 are placed at uneven time intervals to match the various phases of this signal. This generated waveform is still created with 32 points, but placing the points closer together during phases with large slopes improves the overall accuracy.

**Figure 11.52**

Generated waveform using the uneven-time table look-up technique.



The table data structure will encode both the voltage (as a DAC value) and the time. The time parameter is stored as a  $\Delta t$  in cycles to simplify servicing the output compare interrupt. We will assume the TCNT is initialized to count every 500 ns, therefore the maximum  $\Delta t$  that can be generated is 32 ms. Assume the ritual initializes  $I=0$  (Programs 11.8 and 11.9).

```

unsigned short I; // incremented every sample
const unsigned short wave[32]= {
    3048, 2675, 2472, 2526, 2817, 2981, 2800, 2337, 1901, 1499, 1165, 1341, 1570, 1597, 1337, 952,
    662, 654, 863, 1210, 1605, 1950, 2202, 2141, 1955, 1876, 2057, 2366, 2755, 3129, 3442, 3382} ;
const unsigned short dt[32]= { // time increment in 500 ns cycles
    2000, 2000, 2000, 2500, 2500, 2000, 2000, 1500, 1500, 2000, 4000, 2000, 2500, 2000, 2000, 2000,
    2000, 1500, 1500, 1500, 1500, 2000, 2500, 2000, 2000, 2000, 1500, 1500, 2000, 2500, 2000} ;

```

**Program 11.8**

Data structure with delta time and value for the waveform.

**Program 11.9**

Periodic interrupt used to create the analog output waveform.

```

unsigned short Time; // every 1ms
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20; // ack C5F
    if((++I)==32) I=0;
    TC5 = T C5+dt[I]; // variable rate
    DAC_out(wave[I]);}

```

### 11.4.7 PWM DAC

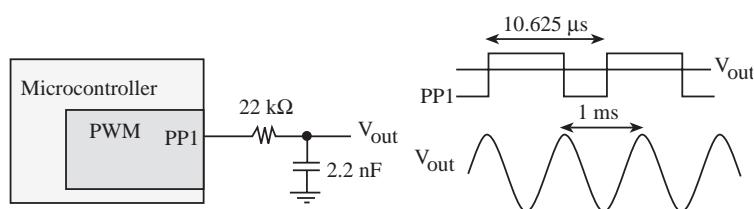
We can use the PWM module to create a DAC. The PWM digital signal is fed through an analog low-pass filter to create a DC voltage linearly related to the duty cycle of the PWM wave. The precision of the DAC will be the precision of the counter used to create the PWM. In particular, the precision will be the number we put into the PWMPER register of the 9S12. For example, if we use the software in Program 6.15, then the DAC precision will be 250 alternatives or about eight bits. The frequency of the DAC is the rate at which the DAC output can be changed and depends on the frequency of the PWM output. In particular, we can change the PWM duty cycle only once per cycle. Using the software in Program 6.15, we can update the DAC 100 times a second. However, if we change the PWM clock to the A clock with no scaling, the PWM clock will be the E clock. If the E clock is 24 MHz, then the PWM frequency will be 96 kHz, and the software can change the duty cycle every 10.4  $\mu$ s. It is usually best for the PWM signal frequency to be 10 to 100 times higher than the desired bandwidth of analog signals to be produced. Generally, the higher the PWM frequency, the lower the order of filter required, and the easier it is to build a suitable filter. Let  $f_{max}$  be the highest frequency component we wish to create in the analog output, and let  $f_{PWM}$  be the PWM frequency. We want the analog LPF to reject  $f_{PWM}$  and pass  $f_{max}$ .

**Example 11.3** Design a system that outputs a 1-kHz sine wave using a PWM DAC.

**Solution** We will build an 8-bit PWM DAC and output 32 points for each cycle in the wave. We can create the 8-bit 94-kHz PWM using PWMPERO equal to 255 and an A clock of 24 MHz (see Figure 11.53 and Program 11.10). The LPF must pass 1 kHz and reject 94 kHz. We choose the LPF cutoff at 3 kHz, which is between  $f_{PWM}$  and  $f_{max}$ . A passive LPF can be made with a resistor and capacitor,  $f_c = 1/(2\pi RC)$ . An R equal to 22 k $\Omega$  and C equal to 2.2 nF will create a LPF at 3.3 kHz. If we need low output impedance, we could replace the passive filter with an active filter like Figure 11.29.

**Figure 11.53**

PWM DAC used to create a 1-kHz sine wave.



**Program 11.10**

Software for PWM DAC creating a 1-kHz sine wave.

```

void PWMDAC_Init(void){
    DDRP |= 0x02;          // PP1 is output
    PWMPOL |= 0x02;         // PWM channel high then low
    PWMCLK &= ~0x02;        // clock A for PCLK1,
    PWMPRCLK &= ~0x07;       // A clock is 24 MHz
    PWMCAE &= ~0x02;        // left aligned mode
    PWMCTL &= ~0x1C;        // do not concatenate
    PWMPER1 = 255;          // period1, 94 kHz
    PWMDTY1 = 0;             // duty cycle1, off
    PWME |= 0x02;            // enable channel 1
}
// set duty cycle, data to PP1
void PWMDAC_Out(unsigned char data){
    PWMDTY1 = data;        // 0 to 255
}
const unsigned char SinWave[32] = {
    120,140,158,176,191,203,212,218,220,218,212,
    203,191,176,158,140,120,100,82,64,49,37,
    28,22,20,22,28,37,49,64,82,100
};
interrupt 11 void TOC3handler(void){
static unsigned char n;
    TFLG1 = 0x08;           // acknowledge OC3
    TC3 = TC3+750;          // 31.25 us
    PWMDAC_Out(SinWave[n]);
    if(n<31) n=n+1;
    else n = 0;
}
void OC3_Init (){
    TSCR1 = 0x80;           // Enable TCNT, 24MHz
    TSCR2 = 0x00;           // divide by 1
    PACTL = 0;               // timer prescale used for TCNT
    TIOS |= 0x08;           // activate TC3 as output compare
    TIE |= 0x08;             // arm OC3
    TC3 = TCNT+50;          // first interrupt right away
}
void main(void) {
    PLL_Init();              // running at 24MHz
    OC3_Init ();             // 32 kHz periodic interrupt
    PWMDAC_Init();           // 8-bit DAC on PP1
    EnableInterrupts;
    for(;;) {
    }
}

```

## 11.5 Analog-to-Digital Converters

### 11.5.1 ADC Parameters

An ADC converts an analog signal into digital form. The input signal is usually an analog voltage ( $V_{in}$ ), and the output is a binary number. The ADC *precision* is the number of distinguishable ADC inputs (e.g., 256 alternatives, 8 bits). The ADC *range* is the maximum and minimum ADC input (volts, amperes). The ADC *resolution* is the smallest distinguishable change in input (volts, amperes). The *resolution* is the change in input that causes the digital output to change by 1.

$$\text{Range (volts)} = \text{Precision (alternatives)} \cdot \text{Resolution (volts)}$$

Normally we don't specify accuracy for just the ADC, but rather we give the *accuracy* of the entire instrument (including transducer, analog circuit, ADC, and software). Therefore,

accuracy will be described later in Chapter 12 (specifically, Section 12.1.1), part of the systems approach to data acquisition systems. An ADC is *monotonic* if it has no missing codes. This means that if the analog signal is a slow rising voltage, then the digital output will hit all values one at a time. The ADC is *linear* if the resolution is constant through the range. The ADC *speed* is the time to convert, called  $t_c$ . The ADC cost is a function of the number and price of internal components. There are four common encoding schemes for an ADC. Tables 11.8 and 11.9 show the four encoding schemes for an 8-bit ADC.

**Table 11.8**  
Unipolar codes for an 8-bit ADC.

Unipolar codes	Straight binary	Complementary binary
+5.00	1111, 1111	0000, 0000
+2.50	1000, 0000	0111, 1111
+0.02	0000, 0001	1111, 1110
+0.00	0000, 0000	1111, 1111

**Table 11.9**  
Bipolar codes for an 8-bit ADC.

Bipolar codes	Offset binary	2s Complement binary
+5.00	1111, 1111	0111, 1111
+2.50	1100, 0000	0100, 0000
+0.04	1000, 0001	0000, 0001
+0.00	1000, 0000	0000, 0000
-2.50	0100, 0000	1100, 0000
-5.00	0000, 0000	1000, 0000

To convert between straight binary and complementary binary, we simply complement (change 0 to 1, change 1 to 0) all the bits. To convert between offset binary and 2s complement, we complement just the most significant bit. The exclusive-OR instruction can be used to complement a single bit.

Just like the DAC, one can choose the full-scale range to simplify the use of fixed-point mathematics. For example, if a 12-bit ADC had a full-scale range of 0 to 4.095 V, then the resolution would be exactly 1 mV. This means that if the ADC input voltage were 1.234 V, then the result would be  $1234_{10}$ .

The *total harmonic distortion* (THD) of a signal is a measure of the harmonic distortion present and is defined as the ratio of the sum of the powers of all harmonic components to the power of the fundamental frequency. Basically, it is a measure of all the noise processes in an ADC and usually is given in dB full scale. A similar parameter is *signal-to-noise and distortion ratio* (SINAD), which is measured by placing a pure sine wave at the input of the ADC (signal) and measuring the ADC output (signal plus noise). We can compare precision in bits to signal-to-noise ratio in dB using the relation  $\text{dB} = 20 \log_{10}(2^{-n})$ . For example, the 12-bit MAX1247 ADC has a SINAD of  $-72$  dB. Notice that  $20 \log_{10}(2^{-12})$  is  $-72$  dB.

## 11.5.2 Two-Bit Flash ADC

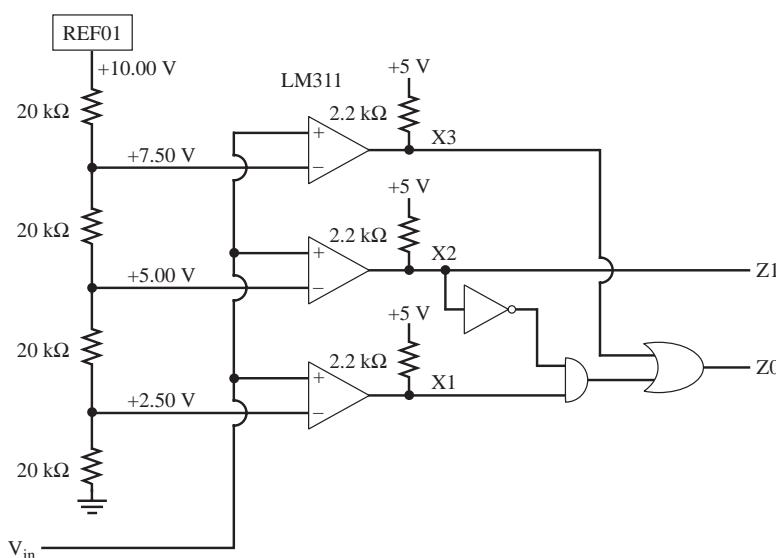
The flash ADC can be used for high-speed, low-precision conversions. Four equal-value resistors are used to create reference voltages 7.50, 5.00, and 2.50. The LM311 voltage comparators perform the ADC conversion. The digital circuit produces the binary code ( $Z_1, Z_0$ ). The conversion speed is limited by the comparator delay, which can be as fast as 100 ns (Table 11.10, Figure 11.54).

**Table 11.10**  
Output results for a 2-bit flash ADC.

$V_{in}$	X3	X2	X1	Z1	Z0
$2.5 > V_{in}$	0	0	0	0	0
$5.0 > V_{in}$	2.5	0	0	1	0
$7.5 > V_{in}$	5.0	0	1	1	1
$V_{in}$	7.5	1	1	1	1

**Figure 11.54**

Two-bit flash ADC.



Some flash converters include two more comparators. The underflow comparator is used to determine if  $V_{in} < 0$ . The overflow comparator is used to determine if  $V_{in} > +10\text{ V}$ .

A 3-bit flash ADC would place eight equal-value resistors in series to generate the reference voltages 1.25, 2.50, 3.75, 5.00, 6.25, 7.50, and 8.75. Seven comparators would perform the ADC conversion. A 74LS148 8-to-3 encoder could be used to convert the X1, X2, X3, X4, X5, X6, X7 comparator outputs into the 3-bit binary code Z2, Z1, Z0. With a flash ADC there is an exponential relationship between the ADC precision and the number of components. In other words, an 8-bit flash requires 256 resistors, 255 comparators, and a 255 to 8-bit digital encoder.

**Observation:** The cost of a flash ADC relates linearly with its precision in alternatives.

There are two approaches to produce a signed flash ADC. The direct approach would be to place the equal resistors from a +10.00-V reference to a -10.00-V reference. The middle of the series resistors ("zero" reference) should be grounded. The digital circuit would then be modified to produce the desired digital code.

The other method would be to add analog preprocessing to convert the signed input to an unsigned range. This unsigned voltage is then converted with an unsigned ADC. The digital code for this approach would be offset binary. This approach will work to convert any unsigned ADC into one that operates on bipolar inputs.

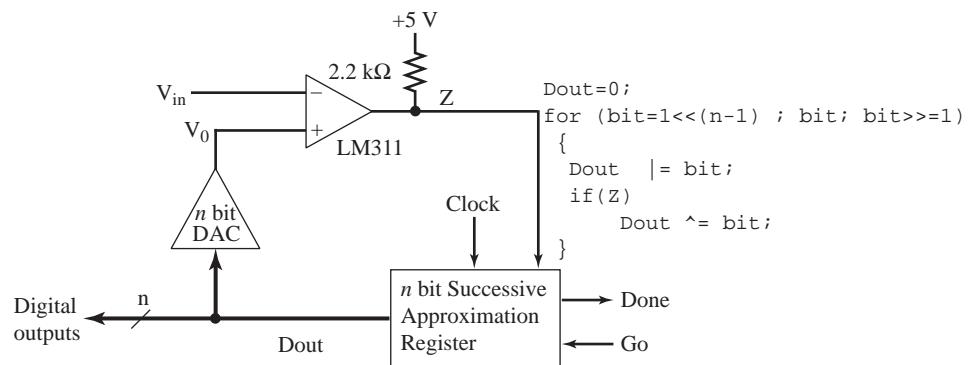
### 11.5.3 Successive Approximation ADC

The most pervasive method for ADC conversion is the successive approximation technique. An ADC with a precision of  $n$  is clocked  $n$  times. At each clock another bit is determined, starting with the most significant bit. For each clock, the successive approximation hardware issues a new "guess" ( $V_0$ ) by setting the bit under test to a 1. If  $V_0$  is now higher than the unknown input  $V_{in}$ , then the bit under test is cleared. If  $V_0$  is less than  $V_{in}$ , then the bit under test remains 1. The C program in Figure 11.55 illustrates the algorithm. In this program,  $n$  is the precision in bits,  $bit$  is an unsigned integer that specifies the bit under test (e.g., for an 8-bit ADC,  $bit$  goes 128, 64, 32, 16, . . . , 1),  $Dout$  is the ADC digital output, and  $z$  is the binary input that is true if  $V_0$  is greater than  $V_{in}$ .

Most successive approximation ADCs use a current-output DAC and a current comparator, instead of the voltage DAC and voltage comparator shown above. In this case, each guess is converted to a current by the DAC. The input voltage is also converted to a current (notice the low-input impedance of this ADC). The current comparator determines if the guess is high or low. If the guess is low, the bit under test is left set to 1. If the guess is high, the bit under test is cleared to 0. This procedure starts with the most significant bit and stops with bit 0. An  $n$ -bit ADC requires exactly  $n$  cycles to convert, independent of the input voltage level.

**Figure 11.55**

Successive approximation ADC.



**Observation:** The speed of a successive approximation ADC relates linearly with its precision in bits.

### 11.5.4 Sixteen-Bit Dual Slope ADC

The dual slope technique can be used to construct an ADC with precision ranging from 16 to 20 bits (Figure 11.56). This 16-bit ADC will have a range of 0 to +10 V and a conversion time of 130 ms. There are three phases for the conversion. The three digital signals P0, P1, and P2 each control a SPST BiFET analog switch. Only one switch is activated at a time. Let the times  $t_0$ ,  $t_1$ , and  $t_2$  refer to the time at the end of each phase (Figure 11.57). During phase 0, P0 is active, which shorts the capacitor C. This initialization phase is used to make the output  $V_{out}$  equal to zero. Thus,

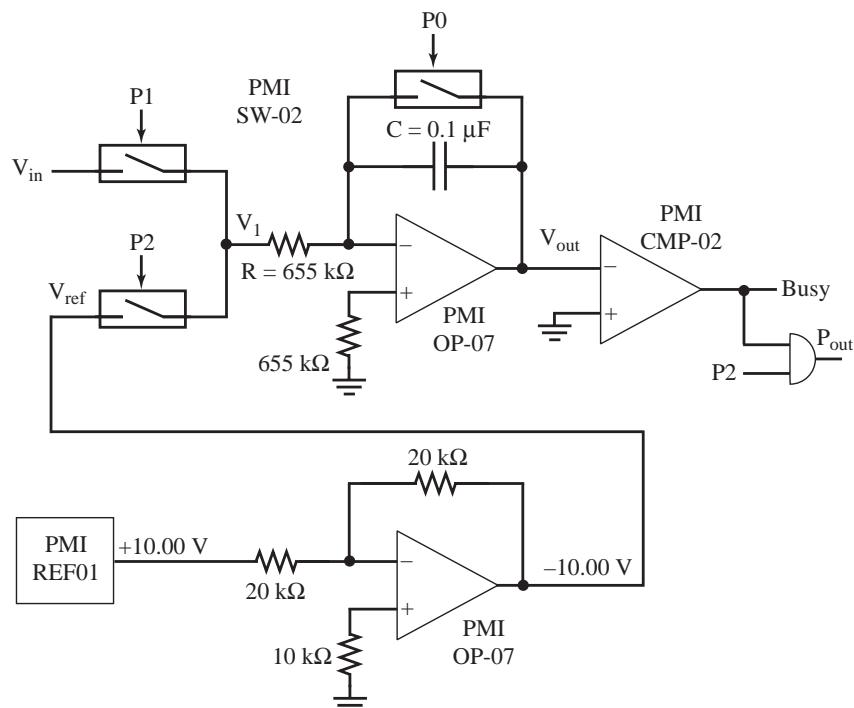
$$V_{out}(t_0) = 0$$

Phase 0 must be long enough to completely discharge the capacitor. During phase 1, P1 is active, which connects the unknown input voltage  $V_{in}$  to the integrator. The length of phase 1 is exactly 65,535  $\mu$ s. We will call this length of phase 1 a time reference,

$$t_{ref} = t_1 - t_0 = 65,535 \mu s$$

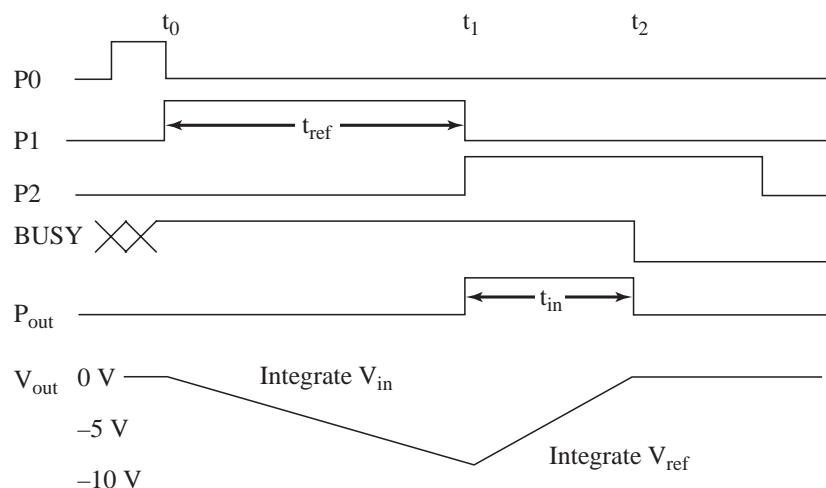
**Figure 11.56**

Hardware required to implement the dual slope ADC conversion technique.



**Figure 11.57**

Digital and analog waveforms during a dual slope ADC conversion.



Since the input voltage is a constant and  $V_{out}(t_0)$  is known, the integration can be simplified:

$$V_{out}(t_1) = V_{out}(t_0) - \frac{1}{RC} \int_{t_0}^{t_1} V_{in}(s)ds = -\frac{1}{RC} V_{in} t_{ref}$$

The largest negative voltage will occur when  $V_{in}$  equals +10 V. The values of R, C, and  $t_{ref}$  were chosen to prevent the integrating op amp from saturating. Output compare can be used to generate  $t_{ref}$ , and input capture can be used to measure  $t_{in}$ . During phase 2, P2 is active, which connects the reference voltage  $V_{ref}$  to the integrator. The negative voltage reference is created with the REF01 voltage reference and an inverting amplifier. During this phase  $V_{out}$  is integrated back to zero. The phase ends when  $V_{out} = 0$ . Thus,

$$V_{out}(t_2) = 0$$

$t_{in}$  is the time spent in phase 2,  $t_{in} = t_2 - t_1$ . The pulse width of  $P_{out}$ ,  $t_{in}$ , is measured with a precision timer. Again, since  $V_{ref}$  is constant, the integration can be simplified.

$$V_{out}(t_2) = V_{out}(t_1) - \frac{1}{RC} \int_{t_1}^{t_2} V_{ref}(s)ds = V_{out}(t_1) - \frac{1}{RC} V_{ref} t_{in} = 0$$

Substituting from above the value of  $V_{out}(t_1)$

$$\frac{1}{RC} V_{in} t_{ref} - \frac{1}{RC} V_{ref} t_{in} = 0$$

The next algebraic step is critical to explain why this method can be used to produce high-precision ADCs. The RCs cancel!

$$V_{in} t_{ref} - V_{ref} t_{in} = 0$$

We can calculate the unknown voltage  $V_{in}$  from the voltage reference  $V_{ref}$ , the time reference  $t_{ref}$ , and the time measurement  $t_{in}$ .

$$V_{in} = \frac{V_{ref}}{t_{ref}} t_{in}$$

The 16-bit precision requires that the time  $t_{in}$  be measured with a resolution of 1  $\mu$ s and a range of 0 to 65,535  $\mu$ s. The operation can be summarized in Table 11.11.

## 11.5.5 Sigma Delta ADC

The sigma delta ADC is used in many audio applications (Figure 11.58). It is a cost-effective approach to 16-bit 44-kHz sampling (CD quality). Sigma delta converters have a DAC, a

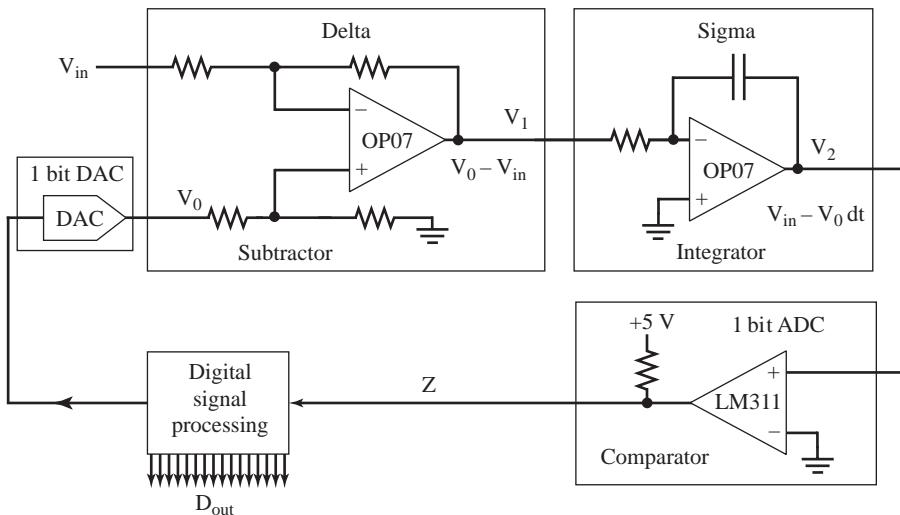
**Table 11.11**

Output results of a dual slope ADC.

$V_{in}$ (V)	$V_{out}(t_1)$ (V)	$t_{in}$ ( $\mu s$ )
0.00000	0.00000	0
0.00015	-0.00015	1
0.00031	-0.00031	2
1.00000	-1.00000	6,554
5.00000	-5.00000	32,768
9.99985	-9.99985	65,535

**Figure 11.58**

Block diagram of the sigma delta ADC conversion technique.



comparator, and digital processing similar to the successive approximation technique. While successive approximation converters have DACs with the same precision as the ADC, sigma delta converters use 1-bit DAC (which is simply a digital signal itself). The digital signal processing will run at a clock frequency faster than the overall ADC system (oversampled). It uses complex signal processing to drive the output voltage  $V_0$  to equal the unknown input  $V_{in}$  in a time-averaged sense. The “delta” part of the sigma delta converter is the subtractor, where  $V_1 = V_0 - V_{in}$ . Next comes the “sigma” part that implements an analog integration. If  $V_0$  equals the unknown input  $V_{in}$  in a time-averaged sense, then  $V_2$  will be zero. The comparator tests the  $V_2$  signal. If  $V_2$  is positive, then  $V_0$  is made smaller. If  $V_2$  is negative, then  $V_0$  is made larger. This DAC-subtractor-integrator-comparator-digital loop is executed at a rate much faster than the eventual digital output rate.

A very simple software algorithm, shown in Program 11.11, is implemented as a periodic interrupt. This algorithm is much too simple to be appropriate in an actual converter, but it does illustrate the sigma delta approach. For an 8-bit conversion, the internal clock rate (interrupt rate) is 256 times the output rate of the 8-bit samples. We assume that the input voltage  $V_{in}$  is between 0 and +5 V. In this simple solution, the DAC is set to 1 ( $V_0 = +5$ ) if  $Z$  is 0 ( $V_2 < 0$ ). Conversely, the DAC is set to 0 ( $V_0 = 0$ ) if  $Z$  is 1 ( $V_2 > 0$ ). Each time the DAC is set to 1, the counter, SUM, is incremented. At the end of 256 passes, the value SUM is recorded as the ADC sample. Since there are 256 passes through the loop, the result will vary from 0 to 255. During each interrupt one pass through the sigma delta algorithm is executed. Similar to the other software ADC examples,  $z()$  is a function that returns the value of the comparator output. Let  $dacout()$  be a function that takes a Boolean input parameter and sets the 1-bit DAC. Assume the ritual will clear all three global variables.

## 11.5.6 ADC Interface

Similar to the DAC interface, three approaches exist for interfacing the ADC to the computer (Figure 11.59). In a digital logic interface, the individual data bits are connected to dedicated computer I/O ports. Usually the software toggles the **Start** ADC signal to

```

unsigned char DOUT; // 8-bit sample
unsigned char SUM; // number of times Z=0 and V0=1
unsigned char CNT; // 8-bit counter
void interrupt 13 TOC5handler(void){
    TFLG1=0x20; // ack C5F
    TC5=TC5+rate; // interrupt 256 times faster than the ADC output rate
    if(Z())
        // check input
        DACout(0); // too high, set DAC output, V0=0
    else {
        DACout(1); // too low, set DAC output, V0=+5v
        SUM++;
    }
    if(++CNT==0){
        DOUT=SUM;
        SUM=0;
        // get ready for the next
    }
}

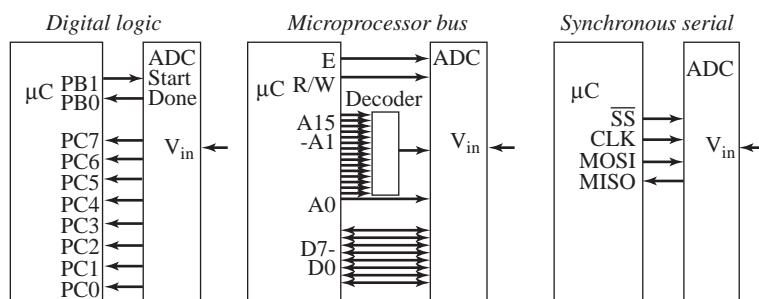
```

### Program 11.11

Software implementation of a sigma delta ADC.

**Figure 11.59**

Three approaches to interfacing an ADC to the microcomputer.



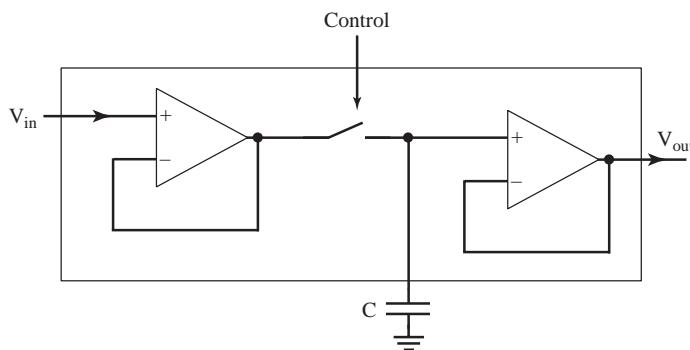
begin an ADC conversion. The software can determine when the ADC is finished by reading the **Done** signal. For example, an 8-bit ADC requires one output port bit and 9-bit input port bits to interface. The software toggles the **Start** signal, waits for the **Done** to be true, then reads the ADC result. The second approach is called microprocessor-bus or microprocessor-compatible. These devices are intended to be interfaced onto the address/data bus of an expanded mode microcomputer. A microcomputer-bus ADC can be interfaced to a MC68HC812A4 without additional external logic. Discussion of this type of interface can be found in Chapter 9. The software interfaces to this type of ADC in a manner similar to the internal ADCs (discussed later in Section 11.10). The SPI/MAX1247 interface (discussed in Section 7.6) is an example of the last type of interface. This approach requires the fewest number of I/O pins. Even if the microcomputer does not support the SPI directly, these devices can be interfaced to regular I/O pins via the bit-banging software approach.

## 11.6 Sample and Hold

A sample and hold (S/H) is an analog latch (Figure 11.60). An alternative name for this analog component is track and hold. The purpose of the S/H is to hold the ADC analog input constant during the conversion. The digital input, **control**, determines the S/H mode. The S/H is in *sample mode*, where V<sub>out</sub> equals V<sub>in</sub>, when the switch is closed. The S/H is in *hold mode*, where V<sub>out</sub> is fixed, when the switch is open. The *acquisition time* is the time for the output to equal the input after the control is switched from hold to sample. This is the time to charge the capacitor C. The *aperture time* is the time for the output to stabilize after the

**Figure 11.60**

The S/H has a digital input, an analog input, and an analog output.



control is switched from sample to hold. This is the time to open the switch that is usually quite fast. The *droop rate* is the output voltage slope ( $dV_{out}/dt$ ) when **control** equals hold. Normally the gain K should be 1 and the offset  $V_{off}$  should be 0. The gain and offset error specify how close the  $V_{out}$  is to the desired  $V_{in}$  when **control** equals sample,

$$V_{out} = KV_{in} + V_{off}$$

where K should be one and  $V_{off}$  should be zero.

To choose the external capacitor C:

1. One should use a polystyrene capacitor because of its high insulation resistance and low dielectric absorption
2. A larger value of C will decrease (improve) the droop rate. If the droop current is  $I_{DR}$ , then the droop rate will be

$$\frac{dV_{out}}{dt} = \frac{I_{DR}}{C}$$

3. A smaller C will decrease (improve) the acquisition time

The system will require a sample and hold if the input signal could change during an ADC conversion. There will be a time during which the ADC samples the input voltage,  $t_{samp}$ . On the 9S12, this sampling time is about 2  $\mu$ s. Let the maximum slew rate of the input signal be  $dV_{in}/dt$ . If the slew rate times the sampling time is larger than the ADC resolution, we should add a sample and hold module to keep the analog input stable during conversion.

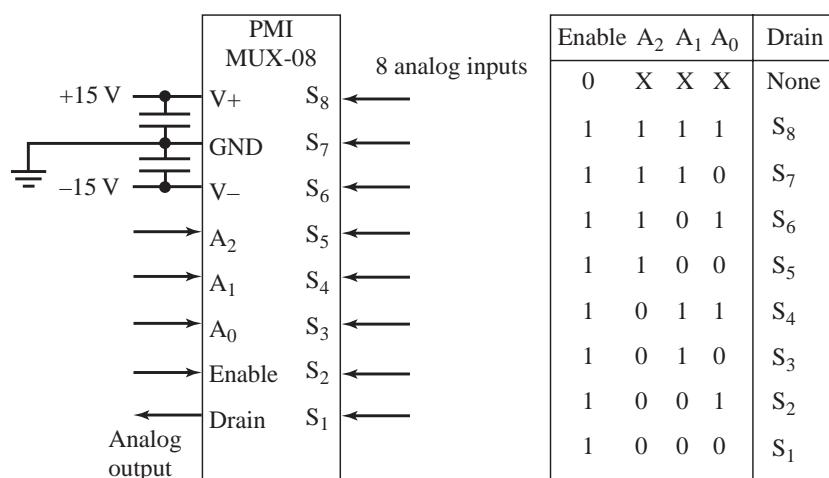
## 11.7 BiFET Analog Multiplexer

The analog multiplexer allows multiple analog signals to be sequentially connected to a single ADC. Figure 11.61 illustrates the operation of a PMI MUX-08. This chip has four digital inputs, **Enable**,  $A_2$ ,  $A_1$ , and  $A_0$ . It also has eight analog inputs ( $S_1 - S_8$ ) and one analog output, **Drain**. When the **Enable** is true, the three address bits determine which analog input is connected to the analog output.

$R_{ON}$  is the on resistance of the switch (200 to 500  $\Omega$  for the PMI MUX-08). The “OFF” isolation,  $ISO_{OFF}$ , is a measure of how well the output is disconnected from its inputs when the device is shut off (60 dB for the PMI MUX-08). To measure  $ISO_{OFF}$ , we place a 500-kHz, +5-V sine wave on channel 8. We make all channels off (Enable = 0) and measure the signal at the output with a  $R_L = 1 R\Omega$ ,  $C_L = 10 pF$ .  $ISO_{OFF}$  is the gain  $V_{out}/V_{in}$  when all channels are off. The crosstalk (CT) is a measure of how much the input on one channel spills over into the output of another channel (70 dB for the PMI MUX-08). To measure CT, we place a 500-kHz, +5-V sine wave on channel 8. We activate channel 4 (Enable = 1,  $A_2A_1A_0 = 011$ ) and measure the signal at the output with a  $R_L = 1 M\Omega$ ,  $C_L = 10 pF$ . CT is the gain from  $V_{in}$ (channel 8) to  $V_{out}$  when channel 4 is on. The settling time  $t_S$  is the time it takes a 10-V step to reach within 0.02% of the final output (2.5  $\mu$ s for the PMI MUX-08).

**Figure 11.61**

An analog multiplexer allows the computer to select one of eight analog signals.



A device similar to the multiplexer is the analog switch. The PMI SW02 is a SPST BiFET analog switch with

$R_{on}$	= 100 $\Omega$ max
$ISO_{off}$	= 58 dB typical
CT	= 70 dB typical
$t_{on}$	= 400 ns max
$t_{off}$	= 300 ns max

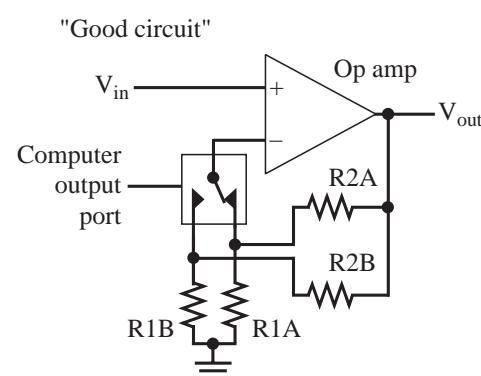
Analog switches and/or multiplexers can be used to make variable-gain analog amplifiers. Because of the large (bad) on-resistance (ranges from 85 to 500  $\Omega$ ), it is important to route the output signal from the switch input into a large input impedance. In this way, there will be a minimal voltage drop across the switch. The two circuits in Figures 11.62 and 11.63 show a good and bad approach to using the bilateral switch to implement a microprocessor-controlled analog amplifier gain. In this first circuit, the gain is either  $1 + R_{2B}/R_{1B}$  or  $1 + R_{2A}/R_{1A}$ , independent of the bilateral switch resistance. Because of the large input impedance into the op amp input terminal, the voltage drop across the switch will be zero.

The problem with the circuit in Figure 11.63 is that the resistance of the switch affects the gain of the circuit. In this second circuit, the gain is either  $1 + (R_{on} + R_{2B})/R_{1B}$  or  $1 + (R_{on} + R_{2A})/R_{1A}$ , where  $R_{on}$  is the bilateral switch resistance. Since  $R_{on}$  can drift with time and temperature, the gain of this circuit will vary as well.

The analog multiplexer in the circuit of Figure 11.64 is used to select the gain from 1, 2, 5, 10, 20, 50, 100, 200.

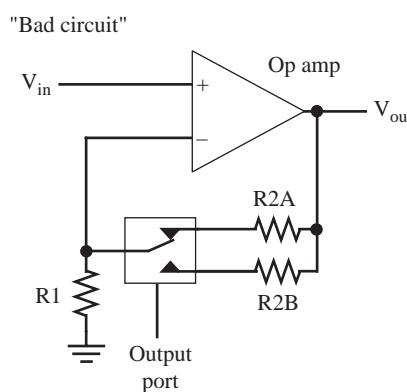
**Figure 11.62**

The proper way to use a bilateral switch to make a variable-gain amplifier.

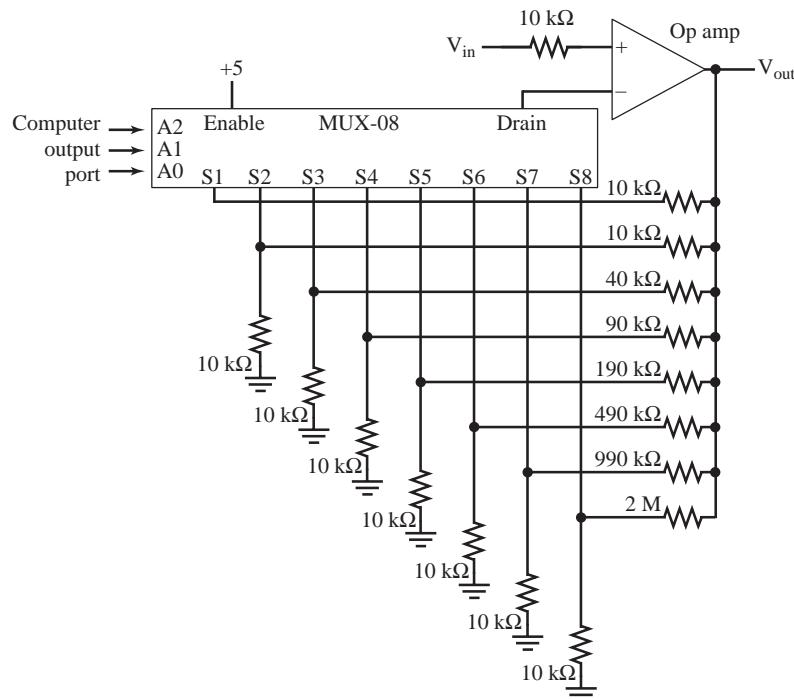


**Figure 11.63**

The improper way to use a bilateral switch to make a variable-gain amplifier.

**Figure 11.64**

The use of an analog multiplexer to make a variable-gain amplifier.



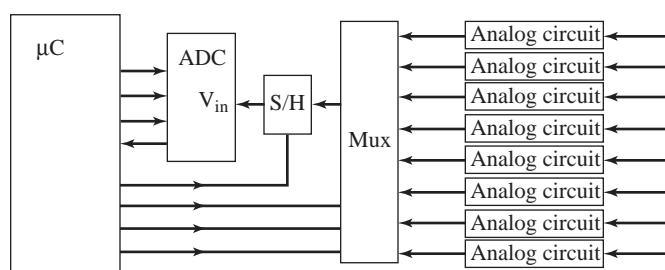
## 11.8 ADC System

### 11.8.1 ADC Block Diagram

The block diagram in Figure 11.65 connects the devices introduced in this chapter to build a data acquisition system. The next chapter will discuss the overall design process, but here in this section we discuss the low-level interfacing issues. Recall that the Maxim MAX1147 ADC integrates the multiplexer, S/H, and ADC into a single package.

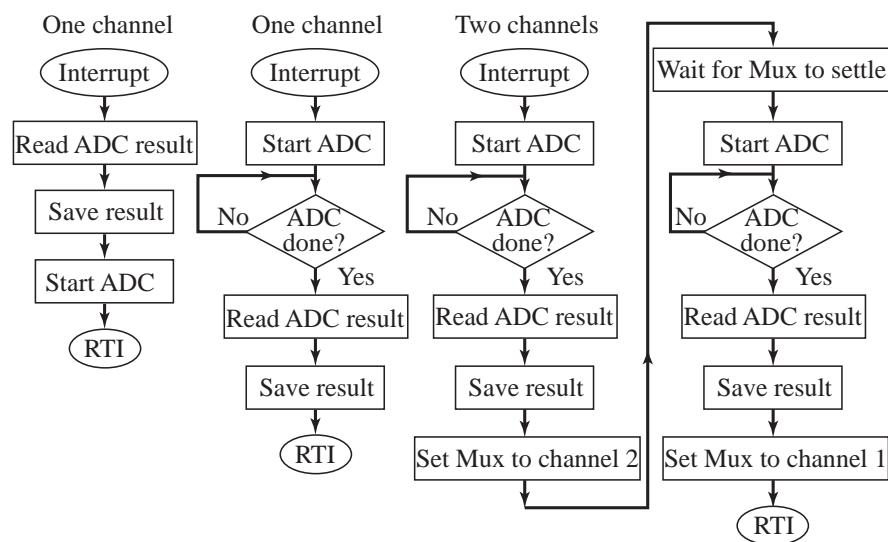
**Figure 11.65**

Together, the analog circuit, multiplexer, S/H, and ADC convert external signals to digital form.



**Figure 11.66**

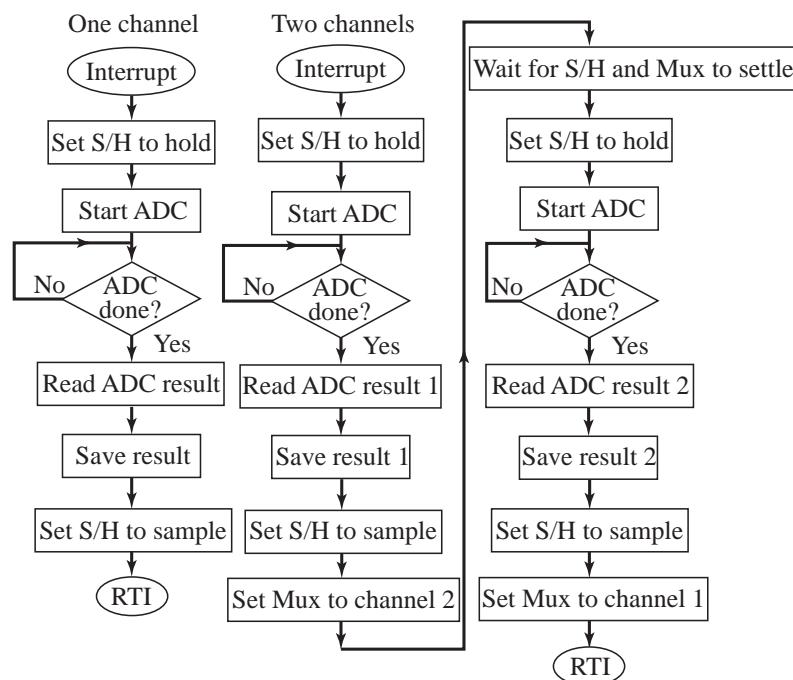
Flowcharts for ADC interrupt software when no S/H is needed.



The flowcharts in Figures 11.66 and 11.67 show the tasks performed in the ISRs. It is assumed that the periodic interrupt occurs at the desired sampling rate. In Figure 11.66, the system does not include a S/H. When only a single channel is being sampled, the multiplexer address is set once in the ritual. There are two approaches to sampling a single channel. If the ADC is started at the end of the interrupt handler, the interrupt software is faster and simpler. Unfortunately, this simple method introduces a delay in the data. In other words, data collected during one interrupt actually represent the signal at the time of the previous interrupt. Multiplexers are usually fast enough that normal software delays from

**Figure 11.67**

Flowcharts for ADC interrupt software when a S/H is needed.



one instruction to the next are sufficient to allow the multiplexer to settle, so no explicit software function needs be added for the “Wait for Mux to settle” step.

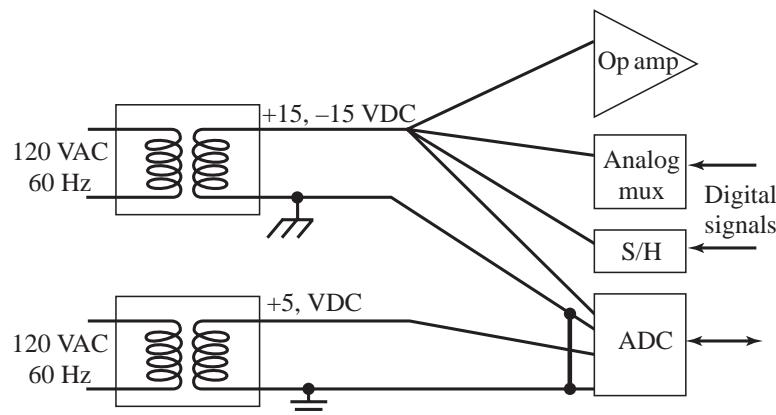
In Figure 11.67, the system does include a S/H. Just like the last example, if only a single channel is being sampled, the multiplexer address is set once in the ritual. Sample and hold modules are usually so slow that normal software delays from one instruction to the next are not sufficient for the S/H to settle, so some explicit software delay may be needed for the “Wait for S/H and Mux to settle” step.

## 11.8.2 Power and Grounding for the ADC System

The analog and digital grounds should be connected only at the ADC (Figure 11.68). Many ADC data sheets discuss layout techniques to minimize noise. A “star” pattern is used to decouple the devices. If the ground lines are in series, then a large current passing from an outer device could affect the ground voltage of an inner device. This can be a significant problem for high-speed devices that sink large transient currents during transitions. Because the S/H module requires careful circuit board layout for the capacitor, most manufacturers suggest PC board layout patterns for their S/H modules.

**Figure 11.68**

The analog and digital grounds are separate, connecting only at the ADC module.



## 11.8.3 Input Protection for CMOS Analog Inputs

It is necessary to protect CMOS inputs from large currents and out of range voltages. The microcontroller is particularly vulnerable to negative voltages. One of the best methods to protect the ADC is to use rail-to-rail analog circuits, powered by the same supply voltage as the microcontroller. For most designs, this approach will limit all analog signals to the range of 0 to the supply voltage.

**Common error:** It is a mistake to neglect the possibility of a broken transducer or disconnected cable.

**Performance tip:** It is good engineering practice to design input protection based on reasonable expected failure modes. If you develop a collection of broken transducers and cables (open and short), you could use these devices to design and test your systems.

## 11.9 Power

### 11.9.1 Regulators

One of the important aspects of a system is power. Many embedded systems are powered with an AC adapter. Other names for this device that takes 120 VAC in and outputs an unregulated DC voltage include power adapter, power block, wall wart, wall cube, and power brick. A 9-V, 500-mA unregulated AC adapter means the voltage is above 9 V for all currents less than 500 mA. However, the actual voltage can vary considerably. For example, the voltage might range from 13 V at no current down to 9.5V at 500 mA. Therefore,

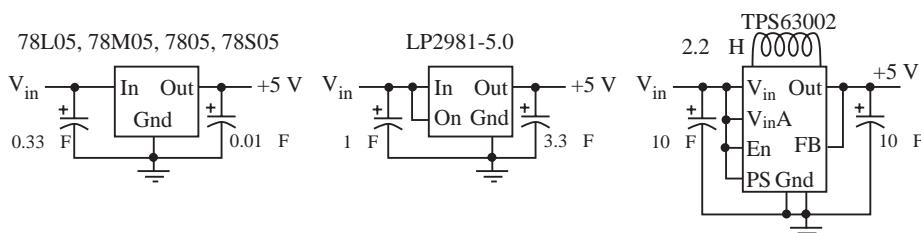
a *regulator* will be used to provide a constant voltage to power the system. There are many considerations when choosing a regulator. A *linear regulator*, like the 78x05 series and the LP2981, use transistors, a reference diode, and feedback to create a constant output voltage (Figure 11.69). Linear regulators need an input voltage,  $V_{in}$ , larger than the output voltage,  $V_{out}$ . If the current is  $I_{out}$ , then power is lost in the regulator and dissipated as heat in the amount of  $(V_{in} - V_{out}) \cdot I_{out}$ . Linear regulators have a *voltage dropout* which specifies how much higher  $V_{in}$  needs to be above  $V_{out}$  in order for it to work. The dropout voltage for the 78L05 is 2 V, meaning  $V_{in}$  must be larger than 7 V to operate. Conversely, the dropout voltage for the LP2981 is only 0.2 V. For example, a 3.7-V Li-ion battery with a LP2981-3.3 regulator could be used to power a 3.3 V system.

A *buck-boost* regulator uses an inductor and a switching circuit to create the constant output voltage. The TPS63002 buck-boost regulator will create a +5 V output as long as the input is between 1.8 and 5.5 V, making it suitable for battery-powered applications. Buck-boost regulators are rated for their efficiency and have a sizeable 100-kHz switching noise in the power line. Since linear regulators have lower noise, they are more suitable for analog electronics.

Regulators can be *fixed output* (e.g., 3.3 or 5 V for a 9S12) or *adjustable*. Adjustable regulators use two resistors to set the output voltage level. Many regulator families have both fixed output and adjustable versions, such as LM1086, LM1117, LT1761, and TPS6300x. Adjustable regulators are convenient if the system needs to have a user-controlled switch to decide whether the system is powered at 3.3 or 5 V. The *maximum current* of the regulator needs to exceed the current requirements of the system. For example, the maximum currents for the 78L05, 78M05, 7805, and 78S05 are 100 mA, 500 mA, 1 A, and 2 A, respectively. The *line regulation* parameter specifies how the output voltage will vary as a function of input voltage. The *load regulation* parameter specifies how the output voltage will vary as a function of the current to the system.

**Figure 11.69**

Regulators can be used to create a constant voltage to power the system.



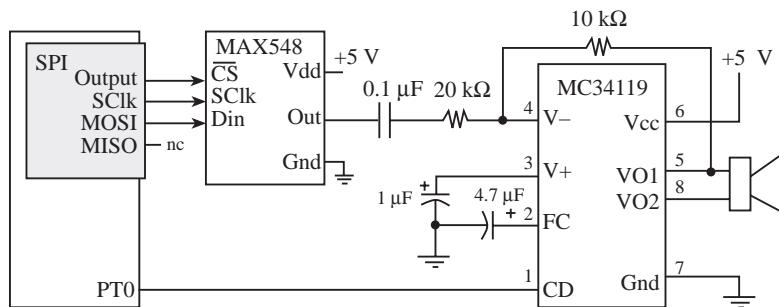
**Observation:** Regulators are very different from each other, so it is essential to read the data sheet and follow all the design recommendations concerning external components, heat sinks, and layout.

### 11.9.2 Low Power Design

To save energy, our parents taught us to “turn off the light when you leave the room.” We can use this same approach to conserve energy in our analog circuits. There are many ways to place analog circuits in low-power mode. Some analog circuits have a low-power mode that the software can select. For example, the MAX548 8-bit DAC requires 550  $\mu$ A for normal operation, but the software can place it into shut-down mode, reducing the supply current to 0.3  $\mu$ A. Some analog circuits have a digital input that the microcontroller can control placing the circuit in active or low-power mode. For example, the MC34119 audio amplifier has a **CD** pin (see Figure 11.70). When this pin is low, the amplifier operates normally with a supply current of 3 mA. However, when the **CD** pin is above 2 V, the supply current drops down to 65  $\mu$ A. So, when the software wishes to output sound, it sends a command to the MAX548 to turn on (bit 4 = 0) and makes PT0 equal to 0. Conversely, when the software wishes to save power, it sends a shut-down command to the MAX548 (bit 4 = 1) and makes PT0 high.

**Figure 11.70**

Audio amplifier that can be placed into low-power mode.

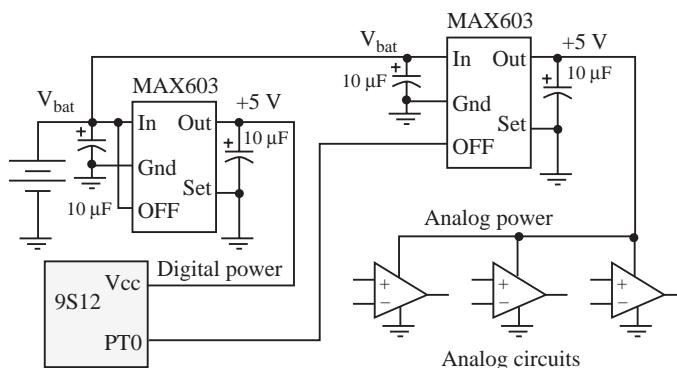


The most effective way to place an analog circuit in a low-power state is to actually shut off its power. Some regulators have a digital signal the microcontroller can control to apply or remove power to the analog circuit. For example, when the OFF pin of the MAX603 regulator is high, the voltage output is regulated to +5 V, as shown in Figure 11.71. Conversely, when the OFF pin is low, the regulator goes into shut-down mode, and no current is supplied to the analog circuit. When the software wishes to turn off power to the analog circuit, it makes PT0 equal to 0. Conversely, when the software wishes to enable the analog circuit, it makes PT0 high. The microcontroller itself always will be powered. However, most microcontrollers (including the 9S12) can put themselves into a low-power state.

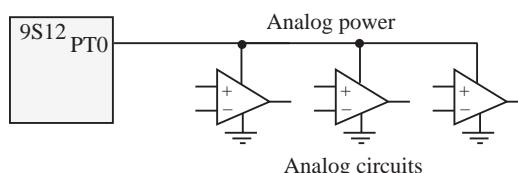
We can save power by designing with low-power components. Many analog circuits require a small amount of current, even when active. The MAX494 requires only 200 μA per amplifier. If there are ten op amps in the circuit, the total supply current will be 2 mA. For currents less than 10 mA, we can use the output port itself to power the analog circuit, as shown in Figure 11.72. To activate the analog circuit, the microcontroller makes PT0 high. To turn the power to the analog circuit off, the microcontroller makes PT0 low.

**Figure 11.71**

Power to analog circuits can be controlled by switching on/off the regulator.

**Figure 11.72**

Power to analog circuits can be delivered from a port output pin.



As we mentioned in Chapter 1, considerable power can be saved by reducing the supply voltage. The 9S12C32 will operate at 3.3 V, requiring less than half the power for an equivalent +5 V system. Power can be saved by turning off modules (like the timer, ADC, SCI, and SPI) when not in use. Whenever possible, slowing down the E clock with the PLL will save power. Many microcontrollers (including the 9S12) can put themselves into a low-power sleep mode to be awakened by a timer or external event.

**11.9.3 Battery Power** A *battery* is a source of energy that can be used in an embedded system to make the system portable. Another application of batteries is to supply power to a mission-critical system when the regular AC power is lost temporarily. Typically, a battery has three parts. The anode is the negative terminal of the battery, the cathode is the positive terminal, and the electrolyte is a liquid solution that accepts, stores, and releases energy. These three components can be constructed from many different materials and configured in an almost endless array of sizes and shapes. The type, size, and shape of the materials play a major role in determining the battery performance. A *primary battery* is used once and discarded, and a *secondary battery* can be recharged and reused.

There are many parameters to consider when selecting a battery. *Nominal voltage* is the typical voltage of the battery when fully charged. Some batteries maintain a fairly constant voltage output while energy is being discharged. However, other batteries will drop the voltage steadily during usage. Physical parameters of the battery (such as *volume*, *weight*, and *shape*) often play a significant role in the overall appeal of an embedded system. *Peak current* is the maximum current the battery can deliver. Shelf-life, operating temperature, and storage temperature are other parameters to consider when choosing a battery. *Memory effect* is an observable condition in some rechargeable batteries that causes them to hold less charge over time. The *energy storage* of a battery is typically defined in amp-hours, because the voltage is assumed constant. The standard units of energy are watt-hours (1 W-hr is 3600 J). One can estimate the operation time of a battery-powered embedded system by dividing the energy storage by the required current to run the system. The *power budget* embodies this concept. Let  $E$  be the battery specification in amp-hours and  $t_{\text{life}}$  be the desired lifetime of the product; then we can estimate the average current our system is allowed to draw:

$$\text{Average Current} \leq E/t_{\text{life}}$$

**Checkpoint 11.11** A medical device implanted inside the body has a 500 mA-hour battery that must last five years. What is your power budget?

*Heavy-duty* batteries were first made with zinc carbon in the mid-1800s, but now are made with zinc chloride. They are a low-cost, low-performance battery but are not appropriate for most embedded applications. An *alkaline* battery is made with alkaline manganese. Alkaline batteries are appropriate for situations that require long shelf-life but size and weight are not important. There are two kinds of *lead-acid* batteries. Flooded lead-acid vent inflammable gasses and require additional water to maintain the proper specific gravity of the acid. Valve-regulated lead-acid (VRLA, also called sealed lead batteries) have about a two-to-one advantage over the flooded-type battery in specific energy and energy density. In the VRLA cell, the vent for the gas space incorporates a pressure relief valve to minimize the gas loss and to prevent direct contact between the headspace and outside air. Lead-acid batteries can be used for backing up power on systems that require large currents. Lead-acid batteries have a maximum storage time of six months at temperatures between 20 and 30°C, after which they require a freshening charge. Zinc chloride, alkaline, and lead-acid batteries all have voltages that drop as energy is drained from them. In these systems, the voltage can be monitored as a measure of the energy left in the battery. However, embedded systems that use these types of battery will require a voltage regulator to maintain a constant voltage for the electronics. For example, a +5 V 9S12DP512 will operate with a power-supply voltage from 4.5 to 5.25 V.

*Nickel-cadmium* (NiCad) and *nickel-metal hydride* (NiMH) are low-cost rechargeable batteries that used to be popular for embedded systems. NiMH batteries have about twice the storage capacity as NiCad. Certain NiCd batteries gradually lose their maximum energy capacity if they are repeatedly recharged after being only partially discharged. Most NiMH batteries do not suffer from a memory effect. The NiMH batteries operate between 10 to

55°C and have a projected life of seven and a half years at 30°C. You should cycle new NiMH batteries three to five times to achieve peak performance. Cycling or conditioning a NiMH battery is performed by completely discharging it then completely recharging it. At room temperature, NiMH batteries will self-discharge in 30 to 60 days without usage, depending on environmental conditions. In general, you can expect NiMH batteries to last up to 500 recharges.

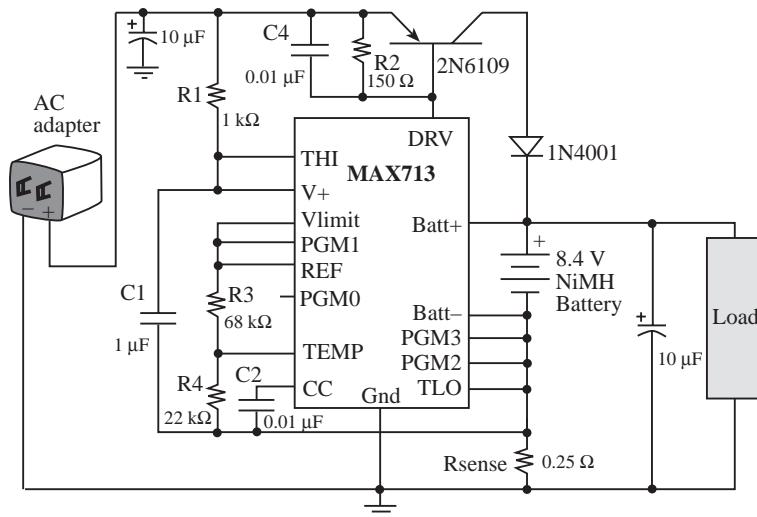
The search for a lighter battery that uses metallic lithium as its anode was driven by the fact that lithium is the lightest and the most electropositive of metals. The specific energy of lithium metal (1727 Ah/lb) is greater than lead (118 Ah/lb) and cadmium (218 Ah/lb). There are a whole range of batteries based on lithium, both single use (used in cameras) and rechargeable. The most common rechargeable type is called *lithium-ion* (Li-ion). When energy is being discharged, the lithium ion moves from the anode to the cathode. During charging, the lithium ion moves from the cathode to the anode. Because of their excellent energy-to-weight and energy-to-size ratios, lithium-ion rechargeable batteries are commonly employed in portable embedded systems. Table 11.12 shows energy storage for typical AA-sized batteries (50 mm tall by 14 mm diameter).

**Table 11.12**  
Energy storage for different battery types.

Battery	Voltage (V)	Energy (mAh)	Type
Alkaline	1.5	2000	Primary
Lithium	1.5	3000	Primary
NiCad	1.2	1200	Secondary
NiMH	1.2	1800	Secondary
Li-ion	3.6	1900	Secondary

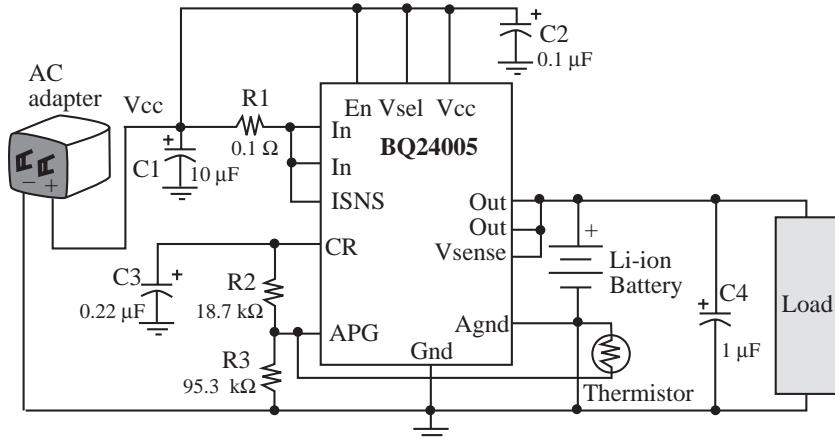
To make the system more convenient, battery-powered embedded systems will include a built-in recharger. The system can run and charge while plugged in. However, it will also run off the battery when AC power is not available. There are many battery charging circuits available. Figure 11.73 shows a charging circuit for a NiMH battery. R1 depends on the smallest  $V_{in}$  from the wall cube. If  $V_{in}$  is larger than 10 V, the set  $R1 = (10 - 5V)/5 \text{ mA} = 1 \text{ k}\Omega$ . R3 and R4 are part of a thermal shut-down safety circuit, which is not implemented in this simple version. The Rsense resistor sets the fast programming current:  $I_{fast} = 2.5 \text{ V}/\text{Rsense}$ , which is 1 A for this circuit. The PGM0, PGM1, PGM2, and PGM3 pins specify the

**Figure 11.73**  
Charging circuit for an 8.4-V NiMH battery with a 1 A charging current.



charging protocol and the number of NiMH cells, which is seven cells or 8.4 V for this circuit. Figure 11.74 shows a charging circuit for a Li-ion battery with a 1.2 A charging current.

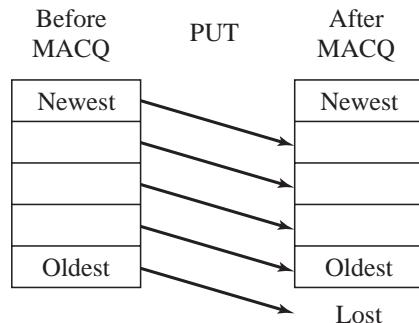
**Figure 11.74**  
Charging circuit for a  
7.2-V Li-ion battery with  
a 1.2 A charging  
current.



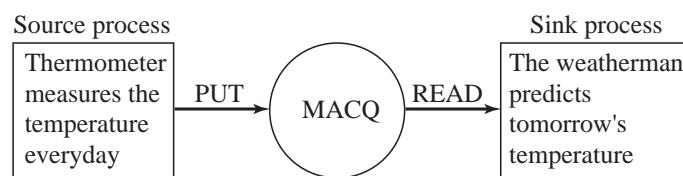
## 11.10 Multiple-Access Circular Queue

A *multiple-access circular queue* (MACQ) is used for data acquisition and control systems (Figure 11.75). A MACQ is a fixed-length, order-preserving data structure. The source process (ADC sampling software) places information into the MACQ. Once initialized, the MACQ is always full. The oldest data are discarded when the newest data are **PUT** into a MACQ. The sink process can read any of the data from the MACQ. The **READ** function is nondestructive. This means that the MACQ is not changed by the READ operation. For example, consider the problem of weather forecasting (Figure 11.76). The weatherman measures the temperature every day at 12 noon, and PUTs the temperature into the MACQ.

**Figure 11.75**  
When data are put into  
a MACQ, the oldest  
data are lost.



**Figure 11.76**  
Application of the  
multiple-access circular  
queue.



To predict tomorrow's temperature, she looks at the trend over the last three days. Let  $T(0)$  be today's temperature,  $T(1)$  be yesterday's temperature, etc. Tomorrow's predicted temperature might be

$$U = \frac{170 \cdot T(0) + 60 \cdot T(1) + 36 \cdot T(2)}{256}$$

The MACQ is useful for implementing digital filters and linear control systems. One common application of the MACQ is the real-time calculation of derivative. Assume the function `Adin()` returns the current sample in millivolts. Also assume the function `ClkHandler()` is a periodic interrupt executed every 1 ms.  $x(n)$  will refer to the current sample, and  $x(n - 1)$  will be the sample 1 ms ago ( $\Delta t = 1$  ms). There are a couple of ways to implement the discrete time derivative. The simple approach is

$$d(n) = \frac{x(n) - x(n - 1)}{\Delta t}$$

In practice, this first-order equation is quite susceptible to noise. In most practical control systems, the derivative is calculated using a higher-order equation like

$$d(n) = \frac{x(n) + 3x(n - 1) - 3x(n - 2) - x(n - 3)}{\Delta t}$$

The C implementation of this discrete derivative (Program 11.12) uses a MACQ. Since  $\Delta t$  is 1 ms, we simply consider the derivative to have units millivolts per millisecond and not actually execute the divide by  $\Delta t$  operation.

<pre> x rmb 8    ;MACQ (mV) d rmb 2    ;derivative(V/s)  TC5handler     ldd TC5     addd #1000     std TC5      ;every 1 ms     movb #\$20,TFLG1 ;ack C5F     movw x+4,x+6    ;shift MACQ     movw x+2,x+4     movw x,x+2     ldaa #\$80     jsr ADC_In ;current data     std x     ldd x+2  ;x[1]     subd x+4  ;x[1]-x[2]     pshd     asld      ;2*(x[1]-x[2])     addd 2,s+ ;3*(x[1]-x[2])     addd x    ;x[0]+3*(x[1]-x[2])     addd x+6  ;derivative in V/s     std d     rti </pre>	<pre> unsigned short x[4]; // MACQ (mV) unsigned short d; // derivative(V/s) void interrupt 13 TC5handler(void){     TC5 = TC5+1000; // every 1 ms     TFLG1 = 0x20; // ack C5F     x[3] = x[2]; // shift data     x[2] = x[1]; // units of mV     x[1] = x[0];     x[0] = ADC_In(0x80); // current     d = x[0]+3*x[1]-3*x[2]-x[3]; } </pre>
--	---

### Program 11.12

Software implementation of first derivative using a MACQ.

When the MACQ holds many data points, it can be implemented using a pointer or index to the newest data. In this way, the data need not be shifted each time a new sample is added. The disadvantage of this approach is that address calculation is required during the READ access.

## 11.11 Internal ADCs

The Freescale 9S12 microcomputer is available in versions that have built-in ADCs. We will begin by discussing the particular I/O registers used to interface analog signals to the individual microcomputer. The common features include:

- Eight-channel operation
- 8-bit or 10-bit resolution
- Successive approximation conversion technique
- A clock and charge pump is used to create higher voltages needed for the ADC
- Two operation modes: single sequence of conversions, then stop, and continuous conversion
- Two channel selection modes: multiple conversions of a single channel (e.g., channel 1,1,1,1), and one conversion each on a group of channels (e.g., channels 0,1,2,3)
- External  $V_{RH}$ ,  $V_{RL}$  analog high/low references

### 11.11.1 9S12 ADC System

Most of the 9S12 microcontrollers have built-in ADC converters. This section specifically covers the MC9S12C32, but the other 9S12 devices operate in a very similar manner. The I/O registers used for the ADC are listed in Table 11.13. The ADC on the MC9S12C32 can be operated in 8-bit mode or 10-bit mode. The 8 pins of Port AD can be individually defined to be analog input, digital output, or digital input. We set the corresponding bit in the ATDDIEN register to be 1 for digital or 0 for analog input. If a pin is digital, then the corresponding bit in the DDRAD register specifies input(0) or output(1). The ADC digital output can be right- or left-justified within the 16-bit result register, and it can be in a signed or unsigned format. When the ADC is triggered, it performs a sequence of conversions, with the sequence length being any number from 1 to 8 conversions. When performing a sequence, it can convert the same channel multiple times or it can convert different channels during the sequence. We can trigger ADC conversions in three ways. The first way is to use an explicit software trigger

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name								
\$0082	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIG	ASCIE	ASCIF	ATDCTL2								
\$0083	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0	ATDCTL3								
\$0084	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0	ATDCTL4								
\$0085	DJM	DSGN	SCAN	MULT	0	CC	CB	CA	ATDCTL5								
\$0086	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0	ATDSTAT0								
\$008B	CCF7	CCF6	CCF5	CCF4	CCF3	CCF2	CCF1	CCF0	ATDSTAT1								
\$008D	Bit 7	6	5	4	3	2	1	Bit 0	ATDDIEN								
\$0270	PTAD7	PTAD6	PTAD5	PTAD4	PTAD3	PTAD2	PTAD1	PTAD0	PTAD								
\$0272	DDRAD7	DDRAD6	DDRAD5	DDRAD4	DDRAD3	DDRAD2	DDRAD1	DDRAD0	DDRAD								
Address	msb								lsb	Name							
\$0090	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR0
\$0092	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR1
\$0094	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR2
\$0096	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR3
\$0098	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR4
\$009A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR5
\$009C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR6
\$009E	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR7

**Table 11.13**

MC9S12C32 registers used for analog-to-digital conversion.

(write to ATDCTL5); when the conversions are complete, the SCF flag is set. The example in Programs 11.13 employs the explicit software trigger to start an ADC conversion. The second way to trigger the ADC is continuous mode. In this mode, the ADC sequence is repeated over and over continuously. The third way is to connect an external trigger to the digital input on PAD7. With an external trigger, we can use busy-wait synchronization (gadfly) on the SCF flag, or arm interrupts (ASCIE = 1) on the ASCIF flag. The results of the ADC conversions can be found in the ATDDR0 to ATDDR7 result registers, where the register number refers to the sequence number. In other words, ATDDR0 contains the result of the first conversion in the sequence, ATDDR1 contains the result of the second conversion, . . . , and ATDDR7 contains the result of the eighth conversion.

```

ADC_Init    movb  #$80,ATDCTL2 ;power up
            movb  #$08,ATDCTL3 ;1 sample
            movb  #$05,ATDCTL4 ;10-bit
            rts
;In:  RegA has channel, $80 to $87
;Out: RegD has ADC result
ADC_In     staa  ATDCTL5 ;Start ADC
loop       brcclr ATDSTAT1,#$01,loop
            ldd   ATDDR0  ;first result
            rts

```

```

void ADC_Init(void){
    ATDCTL2 = 0x80; // enable ADC
    ATDCTL3 = 0x08; // sequence length=1
    ATDCTL4 = 0x05; // 10-bit, divide by 12
} // 2 MHz ADC clock
unsigned short ADC_In(unsigned char chan){
    ATDCTL5 = chan; // start, 0x80 to 0x87
    while((ATDSTAT1&0x01)==0){}; // CCF0
    return ATDDR0; // 10-bit result
}

```

### Program 11.13

Software to sample data using the ADC.

The ATDCTL2 contains bits that activate the ADC module. The 6812 ADC system is enabled by setting **ADPU** equal to 1. The ADC will request an interrupt on the completion of a conversion sequence if the arm bit **ASCIE** is set. **ASCIF** is the ATD Sequence Complete Interrupt Flag. If **ASCIE** = 1, the **ASCIF** flag equals the **SCF** flag, else **ASCIF** reads zero. Write operations to ATDCTL2 have no effect on **ASCIF**. **ETRIGE** is the External Trigger Mode Enable bit. This bit enables an external trigger using the digital input from Port AD bit 7. The external trigger allows us to synchronize sample and ATD conversion processes with external events. If external triggering is enabled, then the type of trigger is defined in the **ETRIGLE** and **ETRIGP** bits as specified in Table 11.14.

**Table 11.14**  
External trigger modes  
for the MC9S12C32  
ADC.

ETRIGLE	ETRIGP	External trigger mode
0	0	Falling edge of PAD7 starts a conversion sequence
0	1	Rising edge of PAD7 starts a conversion sequence
1	0	Perform ADC conversions when PAD7 is low
1	1	Perform ADC conversions when PAD7 is high

The ATDCTL3 and ATDCTL4 contain bits that specify the ADC mode. **S8C**, **S4C**, **S2C**, **S1C** control the number of conversions per sequence. Let **n** be the 4-bit number specified by these bits. For values of **n** from 1 to 7, **n** specifies the sequence length. For values of **n** equal to 0 or 8–15, the sequence length is 8. At reset, the default sequence length is 4 (0100), maintaining software continuity to the HC12 family. This book will not discuss FIFO mode or freeze mode. **SRES8** is the ADC Resolution Select bit. This bit selects the resolution of ADC conversion as either 8 (**SRES8** = 1) or 10 bits (**SRES8** = 0). The ADC converter has an accuracy of 10 bits; however, if low resolution is acceptable, selecting 8-bit resolution will reduce the conversion time.

The time to perform an ADC conversion is determined by the E clock and the ATDCTL4 register. The ATDCTL4 register selects the sample period and the PRS-Clock prescaler. **SMP1, SMP0** are the Sample Time Select bits. These two bits select the length of the second phase of the sample time in units of ATD conversion clock periods, as listed in Table 11.15. The sample time consists of two phases. The first phase is two ATD conversion clock periods long and transfers the sample quickly (via the buffer amplifier) onto the ADC machine's storage node. The second phase attaches the external analog signal directly to the storage node for final charging and high accuracy. Table 11.15 lists the lengths available for the second sample phase. Let **m** be the 5-bit number formed by bits **PRS4-0**. Let **f<sub>E</sub>** be the frequency of the E clock. The ATD conversion clock frequency is calculated as follows:

$$\text{ATD clock frequency} = 1/2 f_E / (m + 1)$$

The default (after reset) prescaler value is 5, which results in a default ATD conversion clock frequency that is the E clock divided by 12. For example, if the E clock is 24 MHz, and **m** is 5, then the ATD clock is 2 MHz. The choice of these parameters involves a trade-off between accuracy and speed. Freescale recommends that the ADC clock frequency be restricted to the 500 kHz to 2 MHz range. For analog signals with white noise, we can essentially add an analog low-pass filter by increasing the ADC sample time. To increase conversion speed, we should select a fast clock and short sample period. The last factor to consider is the slewing rate of the input signal. For signals with a high slope,  $dV/dt$ , we need to select a faster conversion time (i.e., shorter sample time). More discussion about slewing rate will be made in the next chapter when considering the need for a sample-and-hold analog latch. For a 24 MHz E clock, the possible **m** prescales range from 5 (ADC clock = 2 MHz) to 23 (ADC clock = 500 kHz). Other choices are not recommended. The ADC conversion time is equal to  $2(m + 1)(s + n)/f_E$ , where **s** is the total sample time (Table 11.15) and **n** is the number of ADC bits (e.g., 10).

**Table 11.15**  
Sampling time for the  
9S12 ADC.

<b>SMP1</b>	<b>SMP0</b>	<b>First sample phase</b>	<b>Second sample phase</b>	<b>Total sample time</b>
0	0	2 ADC clock periods	2 ADC clock periods	4 ADC clock periods
0	1	2 ADC clock periods	4 ADC clock periods	6 ADC clock periods
1	0	2 ADC clock periods	8 ADC clock periods	10 ADC clock periods
1	1	2 ADC clock periods	16 ADC clock periods	18 ADC clock periods

Writing to the ATDCTL5 register will start an ADC conversion. To begin continuous conversions, we write to the ATDCTL5 with **SCAN** = 1. On the other hand, if we write to ATDCTL5 with **SCAN** = 0, only one sequence occurs. **CC, CB, CA** select the analog input channel(s), whose signals are sampled and converted to digital codes. Because the result registers (16 bits) are wider than the ADC digital code (8 or 10 bits), we must choose where in the result register to put the digital code. **DJM** is the Result Register Data Justification bit, where 1 means right-justified and 0 means left-justified data in the result registers. **DSGN** selects between signed and unsigned format. We set DSGN to 1 for signed data representation and we set it to 0 for unsigned data representation. Table 11.16 describes the four possible 10-bit data formations for the MC9S12C32. When **MULT** is 0, the ATD sequence controller samples only from the specified analog input channel for an entire conversion sequence. When **MULT** is 1, the ATD sequence controller samples across channels. The number of channels sampled is determined by the sequence length value (**S8C, S4C, S2C, S1C**). The first analog channel examined is determined by channel selection code (**CC, CB, CA** control bits); subsequent channels sampled in the sequence are determined by incrementing the channel selection code.

The status register contains a status bit, **SCF**, which we use to poll for the ADC conversion completion. The SCF flag is cleared by writing data into the ADCTL5 (i.e., starting a new conversion). The SCF flag can also be cleared by writing a 1 to it. The CC2,CC1,CC0 bits are the sequence counter as the ADC steps through a conversion sequence. The CCFn bits are individual flags for each of the conversions.

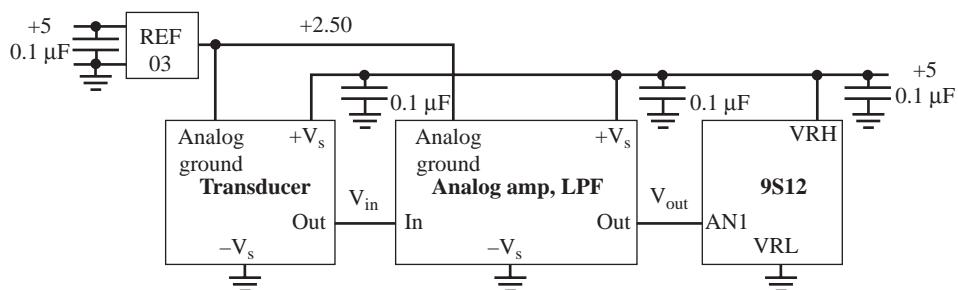
**Performance tip:** If we are interested in a single conversion, we could start the ADC, wait for CCF0, then read the result in ATDDR0.

### 11.11.2 ADC Software

Figure 11.77 shows an analog signal that is connected to analog input channel 1. Using one +5 V supply, this system implements a -2.5 to +2.5 V analog range using a reference chip and rail-to-rail op amps. Because the analog ground is 2.5 V above digital ground, the range of voltages on the analog channel will be 0 to +5 V relative to the 9S12 ground.

**Figure 11.77**

An analog signal is connected to a pin of the internal ADC.



The MC9S12C32 subroutine, **ADC\_In**, will return a 10-bit value representing the analog input. Table 11.16 shows the 8-bit format available. When the 9S12 inserts 8- or 10-bit data into the 16-bit result register, it will pad the extra bits with 0s, but Table 11.16 is shown with bits 15-10 having been sign-extended from bit 9. The C code to perform this sign extension is

```
Result = ATDDR0; // 10-bit signed, right-justified
if(Result&0x200) Result = Result|0xF8; // sign extend if negative
```

Analog input (V)	8-bit unsigned digital output	10-bit unsigned right-justified	10-bit unsigned left-justified	10-bit signed right-justified	10-bit signed left-justified
0.000	\$00 0	\$0000 0	\$0000 0	\$FE00 -512	\$8000 -32768
0.005	\$00 0	\$0001 1	\$0040 64	\$FE01 -511	\$8040 -32704
0.020	\$01 1	\$0004 4	\$0100 256	\$FE04 -508	\$8100 -32512
2.500	\$80 128	\$0200 512	\$8000 32768	\$0000 0	\$0000 0
3.750	\$C0 192	\$0300 768	\$C000 49152	\$0100 256	\$4000 16384
5.000	\$FF 255	\$03FF 1023	\$FFC0 65472	\$01FF 511	\$7FC0 32704

**Table 11.16**

Binary formats used by the Freescale internal ADCs.

For the 9S12 running with an 8-bit precision, the analog input range is 0 to +5 V, and the analog input resolution is 5V/256, which is about 20 mV. For the 9S12 running with a 10-bit precision, the analog input range is 0 to +5 V, and the analog input resolution is 5V/1024, which is about 5 mV. In Program 11.13, the 9S12 ADC will perform multiple conversions

on the same analog signal, but the ADC\_In function returns only the first conversion. The format of the input parameter is

```
bit 7 DJM 1=right , 0=left justified
bit 6 DSGN 1=signed, 0=unsigned
bit 5 SCAN 0=single sequence
bit 4 MULT 0=single channel
bits 2-0 channel number 0 to 7
```

**Checkpoint 11.12:** If the input voltage is 1.0 V, what value, in 10-bit unsigned right-justified mode, will the 9S12 ADC\_In return?

## 11.12 Exercises

**11.1** For each term, give a definition in 32 words or less.

- |                                |                         |
|--------------------------------|-------------------------|
| a) Thermal noise               | h) Output impedance     |
| b) CLC or $\pi$ filter         | i) Shunt reference      |
| c) Voltage comparator          | j) Hysteresis           |
| d) Differential amplifier      | k) Analog isolation     |
| e) Low-pass filter             | l) Complementary binary |
| f) Rail-to-rail                | m) Analog multiplexor   |
| g) Common mode rejection ratio | n) Sample and hold      |

**11.2** For each op amp parameter, give a definition in 32 words or less.

- |                           |                                      |
|---------------------------|--------------------------------------|
| a) Open loop gain         | f) Common-mode input impedance       |
| b) Gain bandwidth product | g) Differential-mode input impedance |
| c) Offset voltage         | h) Slew rate                         |
| d) Offset current         | i) Noise density                     |
| e) Bias current           | j) Supply current                    |

**11.3** For each DAC parameter, give a definition in 32 words or less.

- |               |                   |
|---------------|-------------------|
| a) Precision  | e) Linearity      |
| b) Range      | f) Settling time  |
| c) Resolution | g) Supply current |
| d) Monotonic  | h) Slew rate      |

**11.4** For each ADC parameter, give a definition in 32 words or less.

- |                              |                     |
|------------------------------|---------------------|
| a) Precision                 | e) No missing codes |
| b) Range                     | f) Conversion time  |
| c) Resolution                | g) Supply current   |
| d) Total harmonic distortion | h) Bandwidth        |

**11.5** For each pair of terms, explain the similarities and differences in 32 words or less.

- |                                       |   |
|---------------------------------------|---|
| a) Carbon versus metal film resistor  | e) Op amp versus instrumentation amp          |
| b) Ceramic versus tantalum capacitor  | f) One-pole versus two-pole filter            |
| c) Linear versus buck-boost regulator | g) Li-ion versus NiMH battery                 |
| d) Regular versus rail-to-rail op amp | h) Negative feedback versus positive feedback |

**11.6** Describe each ADC type in 32 words or less.

- |                |                             |
|----------------|-----------------------------|
| a) Flash       | c) Successive approximation |
| b) Sigma delta | d) Dual slope               |

**11.7** A 9S12 with *an 8-bit DAC* is used to create a software-controlled analog output. The 8-bit signed DAC (using offset binary format) is connected to the 9S12 Port B and has a  $-5\text{ V}$  to  $+5\text{ V}$  range as shown in the following table. N is the Port B output (DAC input), and V is the DAC output

N = 8-bit digital input	V = analog output (mvolts)
0 \$00 %00000000	-5000
64 \$40 %01000000	-2500
128 \$80 %10000000	0

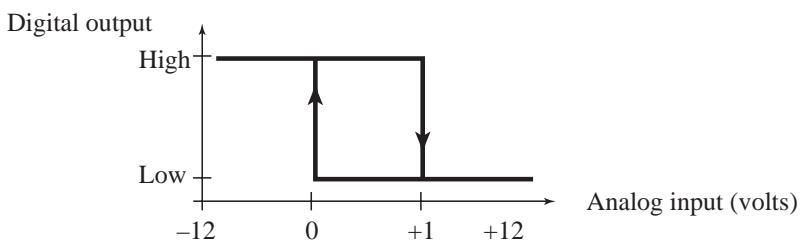
129 \$81 %10000001	÷ 39
192 \$C0 %11000000	+ 2500
255 \$FF %11111111	+ 4961

- a) Derive the linear relationship between N and V. Show both the equation that expresses V in terms of N and the equation that expresses N in terms of V.
- b) What is the *DAC precision*? State the units.
- c) What is the *DAC resolution*? State the units.
- d) Write a 9S12 function, DAout, which takes the desired voltage (in millivolts) and outputs the appropriate value to Port B. The 16-bit signed input parameter is passed by reference on the stack. Here are a couple of typical calling sequences:

ldx #data pointer to value pshx by reference jsr DAout pulx discard parameter data fdb 2314 means 2.314 volts	ldy #neg2 pshy by reference jsr DAout puly neg2 fdb -2000 means -2 volts
---	--

- 11.8 If a 10-bit ADC has a range of 0 to +10 V, what is the resolution?
- 11.9 Give three equivalent ways to specify the precision of a 12-bit ADC.
- 11.10 If an 8-bit ADC has a range of 5 V, what will be the output for an input voltage of 1 V, assuming straight binary encoding?
- 11.11 If an 8-bit ADC takes inputs ranging from -2.5 V to +2.5 V, what will be the output corresponding to +1 V, assuming offset binary encoding?
- D11.12 Design an ADC with the transfer function shown in Figure 11.78. Label all chip numbers (but not pin numbers). Specify the +12, -12, and +5 power supply connections, resistor values, and capacitor values. Offset null potentiometers are not required.

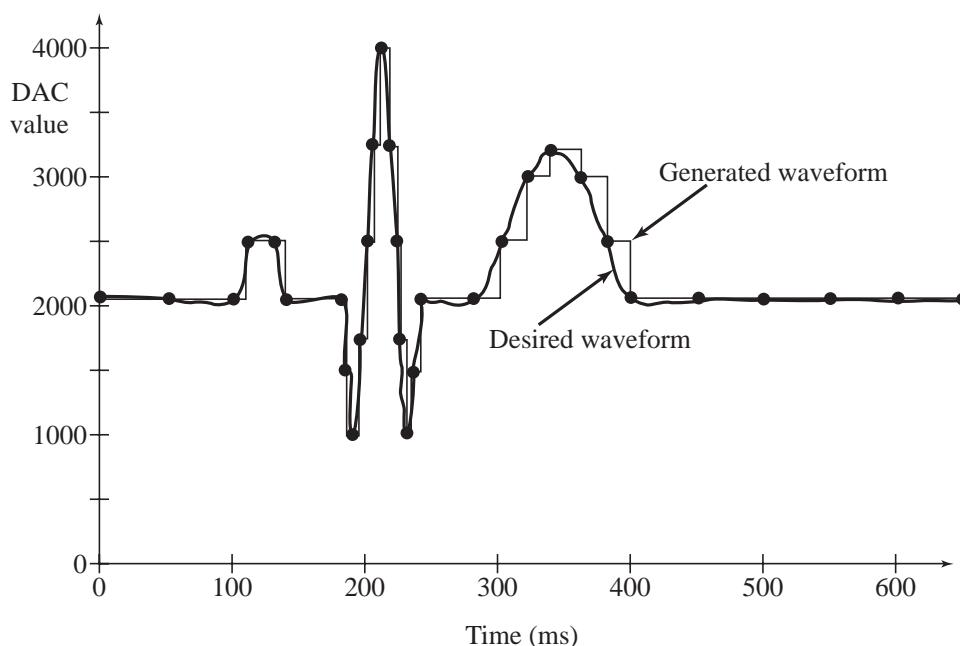
**Figure 11.78**  
Digital output versus analog input for Exercise D11.12.



- D11.13 Design a variable-gain analog amplifier. The analog input is  $V_{in}$ , the 3-bit digital inputs (connected to a microcomputer output digital output) are  $B_2, B_1, B_0$ , and the analog output is  $V_{out}$ . Exactly one of the digital inputs will be 1, and the gain should be

$B_2, B_1, B_0$	Gain ( $V_{out}/V_{in}$ )
001	1
010	10
100	100

- D11.14 In the EKG waveform shown in Figure 11.79, there are long periods of constant value together with short periods of rapidly changing values. Notice that the data points in this figure are placed at uneven time intervals to match the various phases of this signal. Implement software that generates this waveform using uneven time output compare interrupts. In particular, create ROM-based tables that store the nine or ten points and use the DAC8043 12-bit DAC to create the periodic waveform.



**Figure 11.79**  
EKG output waveform for Exercise D11.14.

**D11.15** A computer with *an 8-bit DAC* is used to create a software-controlled analog output. The 8-bit signed DAC (using offset binary format) has a  $-5\text{ V}$  to  $+5\text{ V}$  range as shown in the table. N is the output (DAC input) and V is the DAC output

N = 8-bit digital input	V = analog output (mvolts)
0 \$00 %00000000	-5000
64 \$40 %01000000	-2500
128 \$80 %10000000	0
129 \$81 %10000001	+39
192 \$C0 %11000000	+2500
255 \$FF %11111111	+4961

- a) Derive the linear relationship between N and V. Show both the equation that expresses V in terms of N, and the equation that expresses N in terms of V.
- b) What is the **DAC precision**? State the units.
- c) What is the **DAC resolution**? State the units.

**D11.16** Design an analog circuit with the following specifications:

Two single-ended inputs (not differential)

Any input impedance is OK

Transfer function  $V_{\text{out}} = 5 \cdot V_1 - 3 \cdot V_2 + 5$

You are limited to one OPA227 op amp and one reference chip (you choose it). Give chip numbers but not pin numbers. Specify all resistor values. You will use  $+12$  and  $-12$  V analog supply voltages.

**D11.17** Design an instrumentation amp, using an AD627, with the following transfer function:

$$V_{\text{out}} = 500 \cdot (V_2 - V_1)$$

**D11.18** Design a two-pole Butterworth analog low-pass filter with a cutoff of 250 Hz.

**D11.19** Design an analog circuit with the following transfer function  $V_{\text{out}} = 5 - 2 \cdot V_{\text{in}}$ . The input is a single voltage (not differential). The input range is 0 to 2.5 V and the output range is 0 to 5 V. Use an analog reference and one rail-to-rail op amp. Show your work, and label all chip numbers and resistor values. You do not have to show pin numbers.

**D11.20** The input,  $V_{\text{in}}$ , is differential not single-ended. Design an analog circuit with a transfer function of  $V_{\text{out}} = 50 \cdot V_{\text{in}} + 2.5$  powered by a single +5 V supply. You may use any of the analog chips in this chapter. For example, you may use the REF03 2.50 V reference chip. The input range is -0.05 V to +0.05 V, and the output range is 0 to +5 V. Label all chips, resistors, and capacitors as needed.

**D11.21** The input,  $V_{\text{in}}$ , is single-ended not differential. Design an analog circuit with a transfer function of  $V_{\text{out}} = 50 \cdot V_{\text{in}} - 5$  using one rail-to-rail op amp powered by a single +5 V supply. You may use one REF03 2.50 V reference chip. The input range is 0.1 to 0.2 V, and the output range is 0 to +5 V.

**D11.22** Design an analog circuit with the following specifications:

Two single-ended inputs (not differential)

Any input impedance is okay

Transfer function  $V_{\text{out}} = 3 \cdot V_{1-4} \cdot V_2 + 5$

You are limited to one OPA227 op amp and one reference chip (you choose it). Give chip numbers but not pin numbers. Specify all resistor values. You will use +12 and -12 V analog supply voltages.

**D11.23** Design a two-pole Butterworth analog low-pass filter with a cutoff of 10 Hz.

## 11.13 Lab Assignments

**Lab 11.1** The overall objective of this lab is to design a variable-gain amplifier, where the gain is controlled by a digital output of the microcontroller. The gain settings are 10, 20, 50, and 100. As preparation, design two separate circuits using different approaches. Evaluate the bandwidth, noise, error due to offset voltage, and cost. Build and experimentally measure bandwidth, noise, and offset error.

**Lab 11.2** Most digital music devices rely on high-speed DAC converters to create the analog waveforms required to produce high-quality sound. In this lab you will create a very simple sound-generation system that illustrates this application of the DAC. Your goal is to play your favorite song. For the first step, you will interface a 74HC595 serial in/parallel out shift register to the SPI port. Please refer to the 74HC595 data sheets for the synchronous serial protocol. The second step is to create a DAC from the 8-bit digital output of the 74HC595. You are free to design your DAC with a precision anywhere from 5 to 8 bits. You will convert the binary bits (digital) to an analog output current using a simple resistor network. The third step is to convert the DAC analog output to speaker current using an audio amplifier, such as the MC34119. It doesn't matter what range the DAC is, as long as there is an approximately linear relationship between the digital data and the speaker current. To do this you will have to run the analog circuit in its linear range. Be careful not to saturate the analog circuit. The performance score is based not on loudness, but on sound quality. On the other hand, sound quality will be a function of the number of DAC bits, the linearity of the analog circuit, and the periodic output rate. If an analog signal is noisy, you can add filter capacitors. It is important to add a  $0.1 \mu\text{F}$  bypass capacitor on the power connection of the 74HC595 to prevent output glitches during serial input transmissions. The fourth step is to design a low-level device driver for the DAC. A single 8-bit SPI frame is all that is required to set the DAC output. The fifth step is to design a data structure to store the sound waveform. You are free to design your own format, as long as it uses a formal data structure (i.e., struct). Compressed data occupies less storage, but requires runtime calculation. On the other hand, a complete list of points will be simpler to process, but requires more storage than is available on the microcontroller. The sixth step is to organize the music software into a device driver. Although you will be playing only one song, the song data itself will be stored in the main program, and the device driver will perform all the I/O and interrupts to make it happen. You will need public functions `Rewind`, `Play`, and `Stop`, which perform

operations as does a cassette tape player. The `Play` function has an input parameter that defines the song to play. A background thread implemented with output compare will fetch data from your music structure and send them to the DAC. The last step is to write a main program that inputs from three binary switches and performs the three public functions.

**Lab 11.3** The overall objective of this lab is to design the music player described in Lab 11.2 with a two-channel DAC chip, and design two audio amplifiers. The system will implement two-channel stereo sound.

# 12 Data Acquisition Systems

## Chapter 12 objectives are to

- ❖ Define performance criteria to evaluate our overall data acquisition system
- ❖ Introduce specifications necessary to select the proper transducer for our data acquisition system
- ❖ Describe some typical transducers used in embedded systems
- ❖ Develop a methodology for designing data acquisition systems
- ❖ Analyze the sources of noise and suggest methods to reduce their effect
- ❖ Illustrate concepts of this chapter with case studies

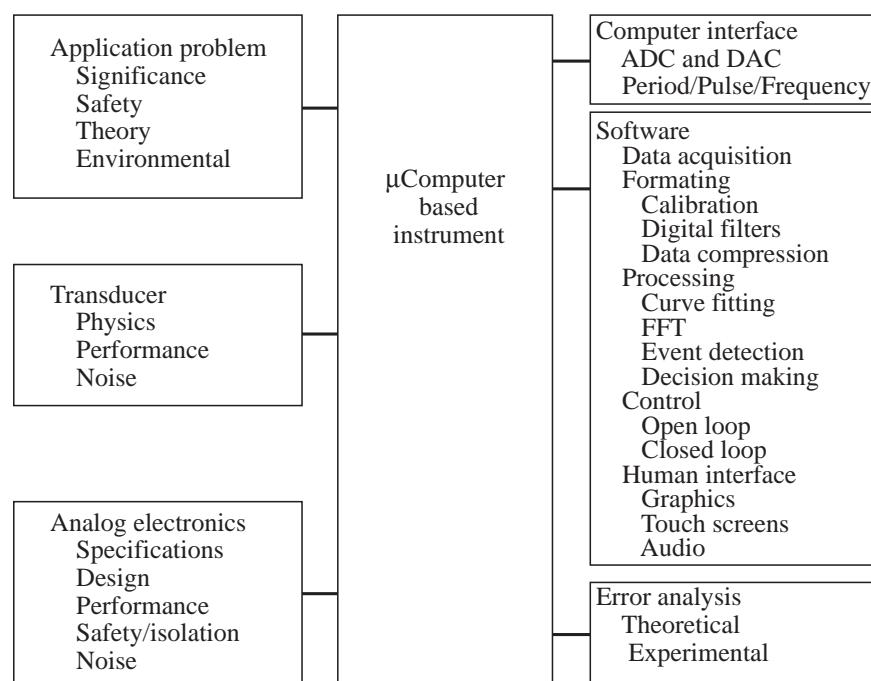
**E**mbedded systems are different from general-purpose computers in the sense that embedded systems have a dedicated purpose. As part of this purpose, many embedded systems are required to collect information about the environment. A system that collects information is called a data acquisition system (DAS). In this chapter, we will use the two terms DAS and instrument interchangeably. Previous chapters presented the basic building blocks to acquire data into the computer, and in this chapter we will combine these blocks into DASs. Sometimes the acquisition of data is the fundamental purpose of the system, such as with a voltmeter, a thermometer, a tachometer, an accelerometer, an altimeter, a manometer, a barometer, an anemometer, an audio recorder, or a camera. At other times, the acquisition of data is an integral part of a larger system such as a control system or communication system. Control systems and communication systems will be discussed in Chapters 13 and 14.

### 12.1 Introduction

Figure 12.1 illustrates the integrated approach to instrument design. In this section, we begin with the clear understanding of the problem. We can use the definitions in this section to clarify the design parameters as well as to report the performance specifications. Next, in Section 12.2, we will define the parameters and discuss the selection of a suitable transducer. In Section 12.3, we put together the analog and digital components, introduced in Chapter 11, to build DASs. The use of period/pulse/frequency as a means of collecting information was developed in Chapter 6. The design of digital filters will be developed later in Chapter 15. A closed-loop control system, as we will see in Chapter 13, includes a DAS. Noise can never be eliminated, but we will study techniques in Section 12.4 to reduce its effect on our system. The integrated approach to design will be illustrated using the case studies in Section 12.5.

**Figure 12.1**

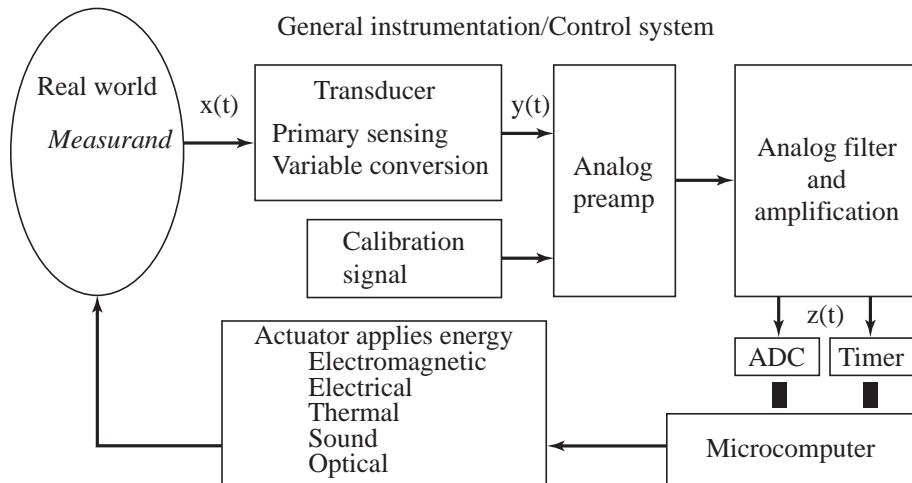
Individual components are integrated into a DAS.



The *measurand* is the physical quantity, property, or condition that the instrument measures. The measurand can be inherent to the object (like position, size, mass, or color), located on the surface of the object (like the human EKG or surface temperature), located within the object (e.g., fluid pressure or internal temperature), or separated from the object (like emitted radiation). In general, a *transducer* converts one energy type into another. In the context of this book, the transducer converts the measurand into an electric signal that can be processed by the microcomputer-based instrument. Typically, a transducer has a primary sensing element and a variable conversion element. The *primary sensing element* interfaces directly to the object and converts the measurand into a more convenient energy form. The output of the *variable conversion element* is an electric signal that we hope depends on the measurand. For example, the primary sensing element of a pressure transducer is the diaphragm, which converts pressure into a displacement of a plunger. The variable conversion element is a strain gage that converts the plunger displacement into a change in electric resistance. If the strain gage is placed in a bridge circuit, the voltage output is directly proportional to the pressure. Some transducers perform a direct conversion without having a separate primary sensing element and variable conversion element. The instrumentation contains *signal processing*, which manipulates the transducer signal output to select, enhance, or translate the signal to perform the desired function, usually in the presence of disturbing factors. The signal processing can be divided into stages. The *analog signal processing* consists of instrumentation electronics, isolation amplifiers, amplifiers, analog filters, and analog calculations. The first analog processing involves calibration signals and preamplification. Calibration is necessary to produce accurate results. An example of a calibration signal is the reference junction of a thermocouple. The second stage of the analog signal processing includes filtering and range conversion. The analog signal range should match the ADC analog input range. Examples of analog calculations include RMS calculation, integration, differentiation, peak detection, threshold detection, PLLs, AM and FM modulation/demodulation, and the arithmetic calculations of addition, subtraction, multiplication, division, and square root. When period, pulse

width, or frequency measurement is used, we typically use an analog comparator to create a digital logic signal to measure. Whereas Figure 12.1 outlined design components, Figure 12.2 shows the data flow graph for a data acquisition system or control system. The control system uses an actuator to drive a parameter in the real world to a desired value, while the DAS has no actuator because it simply measures the parameter in a nonintrusive manner.

**Figure 12.2**  
Data flow graph of a DAS.



The *data conversion element* performs the conversion between the analog and digital domains. This part of the instrument includes hardware and software computer interfaces, ADC, DAC, S/H, analog multiplexer, and calibration references. The *ADC* converts the analog signal into a digital number. In Chapter 6, we saw that the period, pulse width, and frequency measurement approach provides a low-cost high-precision alternative to the traditional ADC. The *digital signal processing* includes data acquisition (sampling the signal at a fixed rate), data formatting (scaling, calibration), data processing (filtering, curve fitting, fast Fourier transform, event detection, decision making, analysis), and control algorithms (open or closed loop). The *human interface* includes the input and output that is available to the human operator. The advantage of computer-based instrumentation is that sophisticated but easy to use and understand devices are possible. The *inputs* to the instrument can be audio (voice), visual (light pens, cameras), or tactile (keyboards, touch screens, buttons, switches, joysticks, roller balls). The *outputs* from the instrument can be numeric displays, cathode ray tube screens, graphs, buzzers, bells, lights, and voice. If the system can deliver energy to the real world, then it is classified as a control system. Control systems will be developed in Chapter 13. In this chapter, we focus on data acquisition. Table 12.1 lists the symbols used in the equations of this chapter.

The rest of this section defines performance criteria we can use to characterize our instrument. Whenever reporting specifications of our instrument, it is important to give the definitions of each parameter, the magnitudes of each parameter, and the experimental conditions under which the parameter was measured, because engineers and scientists apply a wide range of interpretations for these terms.

### 12.1.1 Accuracy

The *instrument accuracy* is the absolute error referenced to the National Institute of Standards and Technology (NIST) of the entire system including transducer, electronics, and software. Let  $x_{mi}$  be the values as measured by the instrument, and let  $x_{ti}$  be the true values

**Table 12.1**  
Nomenclature.

Symbol	Signal (units)
<b>a</b>	Thermistor radius (cm)
<b>b</b>	Offset of the analog signal processing (V)
<b>C</b>	Electric capacitance (F)
<b>D</b>	Thermistor dissipation constant (mW/°C)
<b>e<sub>n</sub></b>	Op amp noise voltage (V)
<b>f<sub>s</sub></b>	ADC sampling rate (Hz)
<b>f<sub>c</sub></b>	Analog system cutoff frequency (where gain = 0.707) (Hz)
<b>f<sub>max</sub>, f<sub>min</sub></b>	Range of frequencies of interest (Hz)
<b>G</b>	Gain of the analog signal processing (V/V)
<b>I</b>	Electric current (A)
<b>k</b>	Boltzmann's constant = $1.38 \times 10^{-23}$ J/K
<b>K</b>	Thermal conductivity [mW/(cm·°C)]
<b>m</b>	Number of multiplexer signals
<b>n</b>	Index used in digital signal processing
<b>n</b>	Number of ADC bits = $\log_2 n_z$
<b>n<sub>x</sub></b>	Precision of <b>x</b> (alternatives)
<b>n<sub>y</sub></b>	Precision of <b>y</b> (alternatives)
<b>n<sub>z</sub></b>	Precision of <b>z</b> , the ADC analog input voltage (alternatives)
<b>P</b>	Electric power (W)
<b>r<sub>x</sub></b>	Range of <b>x</b> = $\max_x - \min_x$ (units of <b>x</b> )
<b>r<sub>y</sub></b>	Range of <b>y</b> = $\max_y - \min_y$ (units of <b>y</b> )
<b>r<sub>z</sub></b>	Range of <b>z</b> = $\max_z - \min_z$ (V)
<b>R</b>	Electric resistance ( $\Omega$ )
<b>R<sub>0</sub></b>	Thermistor constant ( $\Omega$ ) $R = R_0 e^{-T}$
<b>t<sub>ap</sub></b>	S/H aperture time (s)
<b>t<sub>aq</sub></b>	S/H acquisition time (s)
<b>t<sub>c</sub></b>	ADC conversion time (s)
<b>t<sub>mux</sub></b>	Settling time of the multiplexer (s)
<b>T</b>	Temperature (°C or K)
<b>V</b>	Voltage (V)
<b>V<sub>y</sub></b>	Transducer output (typically V)
<b>V<sub>z</sub></b>	ADC input voltage (V)
<b>V<sub>J</sub></b>	Johnson noise (alternatively thermal or white noise) = $\sqrt{4 k T R \Delta f}$
<b>x</b>	Real-world signal (cm, mmHg, °C, etc.)
<b>x(n)</b>	ADC output sequence, sampled at <b>f<sub>s</sub></b>
<b>y</b>	Transducer output (typically V)
<b>y(n)</b>	Digital filter output sequence
<b>z</b>	ADC input voltage (V)
<b>Z</b>	Transducer time constant (s)
<b>Δ<sub>x</sub></b>	Electric impedance ( $\Omega$ )
<b>Δ<sub>y</sub></b>	Fractional temperature sensitivity (1°C)
<b>Δ<sub>z</sub></b>	Thermistor constant (K) $R = R_0 e^{-T}$
<b>Δ<sub>x</sub></b>	Resolution of <b>x</b> (units of <b>x</b> )
<b>Δ<sub>y</sub></b>	Resolution of <b>y</b> (units of <b>y</b> )
<b>Δ<sub>z</sub></b>	Resolution of <b>z</b> , the ADC analog input voltage (V)

from NIST references. In some applications, the signal of interest is a relative quantity (like temperature or distance between objects). For relative signals, accuracy can be appropriately defined many ways:

$$\text{Average accuracy (with units of } x) = \frac{1}{n} \sum_{i=1}^n |x_{ti} - x_{mi}|$$

$$\text{Maximum error (with units of } x) = \max |x_{ti} - x_{mi}|$$

$$\text{Standard error (with units of } x) = \frac{1}{n - 1} \sum_{i=1}^n [x_{ti} - \bar{x}_{mi}]^2$$

In other applications, the signal of interest is an absolute quantity. For these situations, we can specify errors as a percentage of reading or as a percentage of full scale:

$$\text{Average accuracy of reading (\%)} = \frac{100}{n} \sum_{i=1}^n \frac{|x_{ti} - \bar{x}_{mi}|}{x_{ti}}$$

$$\text{Average accuracy or full scale (\%)} = \frac{100}{n} \sum_{i=1}^n \frac{|x_{ti} - \bar{x}_{mi}|}{x_{tmax}}$$

$$\text{Maximum accuracy of reading (\%)} = 100 \max \frac{|x_{ti} - \bar{x}_{mi}|}{x_{ti}}$$

$$\text{Maximum accuracy or full scale (\%)} = 100 \max \frac{|x_{ti} - \bar{x}_{mi}|}{x_{tmax}}$$

**Observation:** The definitions of the terms accuracy, resolution, and precision vary considerably in the technical literature. It is good practice to include both the definitions of your terms and their values in your technical communication.

Since the Celsius and Fahrenheit temperature scales have arbitrary zeros (e.g., 0°C is the freezing point of water), it is inappropriate to specify temperature error as a percentage of reading or as a percentage of full scale when Celsius and Fahrenheit scales are used. When specifying temperature error, we should use average accuracy, maximum error, or standard error that have units of Celsius or Fahrenheit degrees.

### 12.1.2 Resolution

The *instrument resolution* is the smallest input signal difference,  $\Delta$ , that can be detected by the entire system including transducer, electronics, and software. The resolution of the system is sometimes limited by noise processes in the transducer itself (e.g., thermal imaging) and sometimes limited by noise processes in the electronics (e.g., thermistors, resistance temperature devices [RTDs], and thermocouples).

The *spatial resolution* (or *spatial frequency response*) of the transducer is the smallest distance between two independent measurements. The size and mechanical properties of the transducer determine its spatial resolution. When measuring temperature, a metal probe will disturb the existing medium temperature field more than a glass probe. Hence, a glass probe has a smaller spatial resolution than a metal probe of the same size. Noninvasive imaging systems exhibit excellent spatial resolution because the instrument does not disturb the medium that is being measured. The spatial resolution of an imaging system is the medium surface area from which the radiation originates that is eventually focused onto the detector during the imaging of a single pixel, the so-called instantaneous field of view (IFOV). When force is measured, pressure, or flow, the spatial resolution is the effective area over which the measurement is obtained. Another way to illustrate spatial resolution is to attempt to collect a two- or three-dimensional image of the measurand. The spatial resolution is the distance between points in our image.

### 12.1.3 Precision

*Precision* is the number of distinguishable alternatives,  $n_x$ , from which the given result is selected. Precision can be expressed in alternatives, bits or decimal digits. Consider a thermometer instrument with a temperature range of 0 to 100°C. The system displays the output using three digits (e.g., 12.3°C). In addition, the system can resolve each temperature  $T$  from the temperature  $T + 0.1^\circ\text{C}$ . This system has 1001 distinguishable outputs and hence has a precision of 1001 alternatives, or about 10 bits. For a linear system, there

is a simple relationship between range ( $r_x$ ), resolution ( $\Delta_x$ ), and precision ( $n_x$ ). Range is equal to resolution times precision

$$r_x(100^\circ\text{C}) = \Delta_x(0.1^\circ\text{C}) \cdot n_x(1001 \text{ alternatives})$$

where “range” is the maximum minus minimum temperature and precision is specified in terms of number of alternatives. We will develop later in Section 12.3.3 a more complex relationship for nonlinear systems. Table 12.2 illustrates the relationship between alternatives and decimal digits.

**Table 12.2**

Definition of decimal digits.

Alternatives	Decimal digits
1000	3
2000	3½
4000	3¾
10000	4

**Observation:** A good rule of thumb to remember is  $2^{10n} \approx 10^{3n}$ .

**Checkpoint 12.1:** How are precision and resolution related?

**Checkpoint 12.2:** List three major factors that can limit resolution?

### 12.1.4 Reproducibility or Repeatability

*Reproducibility* (or repeatability) is a parameter that specifies whether the instrument has equal outputs, given identical inputs, over some period of time. This parameter can be expressed as the full range or standard deviation of output results given a fixed input, where the number of samples and time interval between samples are specified. One of the largest sources of this type of error comes from transducer drift. *Statistical control* is a similar parameter based on a probabilistic model that also defines the errors due to noise. The parameter includes the noise model (e.g., normal, chi-squared, uniform, salt and pepper<sup>1</sup>) and the parameters of the model (e.g., average, standard deviation).

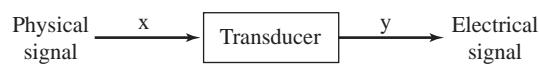
## 12.2 Transducers

In this section, we will start with quantitative performance measures for the transducer. Next, specific transducers will be introduced. Rather than give an exhaustive list of all available transducers, the intent in this section is to illustrate the range of possibilities and to provide specific devices to use in the design sections later in the chapter.

### 12.2.1 Static Transducer Specifications

The input or measurand is  $x$ . The output is  $y$ . A transducer converts  $x$  into  $y$  (Figure 12.3). In this subsection, we assume that the input parameter  $x$  is constant or static.

The input  $x$  and the output  $y$  can be either absolute or differential. An absolute signal represents a parameter that exists in a single place at a single time. A differential signal is



**Figure 12.3**

Transducers in this book convert a physical signal into an electric signal.

derived from the difference between two signals that exist at different places or at different times. Voltage is indeed defined as a potential difference, but when the voltage is referred to ground, we consider it an absolute quantity. On the other hand, if the signal is represented by the voltage difference between two points neither of which is ground, then we consider the signal as differential. Table 12.3 illustrates four types of transducers.

<sup>1</sup>An example of salt and pepper noise is the white and black spots on a poor Xerox copy.

Type	Input → output	Example
Absolute→absolute	$x \rightarrow y$	Thermistor converts an absolute temperature to a resistance
Relative→absolute	$\Delta x \rightarrow y$	Mass balance converts a mass difference to an angle
Absolute→relative	$x \rightarrow \Delta y$	Strain gauge converts a force to a resistance difference
Relative→relative	$\Delta x \rightarrow \Delta y$	Thermocouple converts a temperature difference to voltage difference

**Table 12.3**

Four types of transducers.

### 12.2.1.1 Transducer Linearity

Let  $x_i, y_i$  be the I/O signals of the transducer, as shown in Figure 12.4. The *linearity* is a measure of the straightness of the static calibration curve. Let  $y_i = f(x_i)$  be the transfer function of the transducer. A linear transducer satisfies

$$f(ax_1 + bx_2) = af(x_1) + bf(x_2)$$

for any arbitrary choice of the constants  $a$  and  $b$ . Let  $y_i = mx_i + b$  be the best-fit line through the transducer data. Linearity (or deviation from it) as a figure of merit can be expressed as percentage of reading or percentage of full scale. Let  $y_{\max}$  be the largest transducer output.

$$\text{Average linearity of reading (\%)} = \frac{100}{n} \sum_{i=1}^n \frac{|y_i - mx_i - b|}{y_i}$$

$$\text{Average linearity of full scale (\%)} = \frac{100}{n} \sum_{i=1}^n \frac{|y_i - mx_i - b|}{y_{\max}}$$

Figure 12.4 illustrates the concept of linearity. In particular, we should not assume this thermocouple to be linear over the 0 to 200°C temperature range because doing so would result in significant temperature errors.

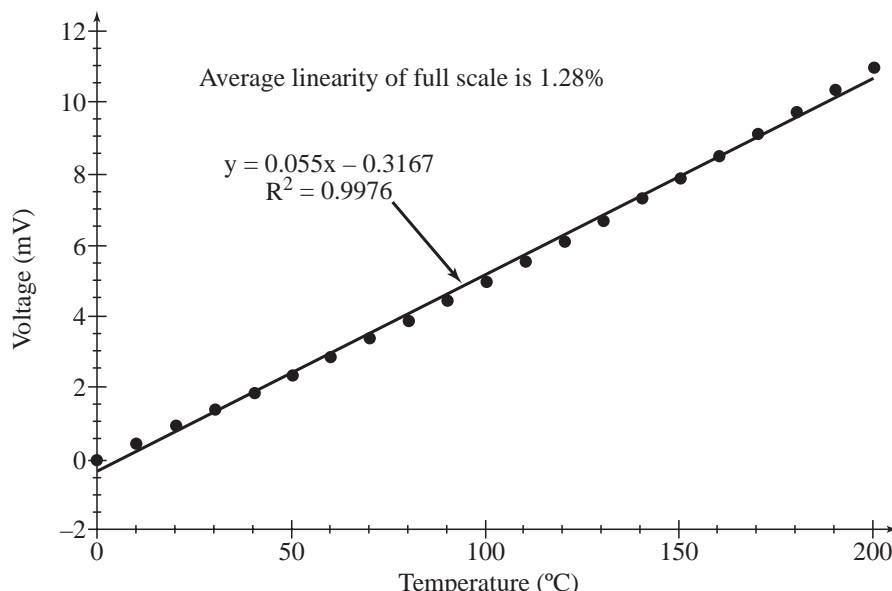
### 12.2.1.2 Transducer Sensitivity

Two definitions for sensitivity are used for temperature transducers. The *static sensitivity* is

$$m = \frac{\Delta y}{\Delta x}$$

**Figure 12.4**

Input/output response of a thermocouple transducer.



If the transducer is linear, then the static sensitivity is the slope  $m$  of the straight line through the static calibration curve that gives the minimum mean squared error. If  $x_i$  and  $y_i$  represent measured I/O responses of the transducer, then the least squares fit to  $y_i = mx_i + b$  is

$$m = \frac{n \sum_{i=1}^n x_i y_i - \sum_{i=1}^n x_i \sum_{i=1}^n y_i}{\sum_{i=1}^n x_i^2 - \left( \sum_{i=1}^n x_i \right)^2} \quad \text{and} \quad b = \frac{\sum_{i=1}^n y_i \sum_{i=1}^n x_i^2 - \sum_{i=1}^n x_i y_i \sum_{i=1}^n x_i}{\sum_{i=1}^n x_i^2 - \left( \sum_{i=1}^n x_i \right)^2}$$

Thermistors can be manufactured to have a resistance value at 25°C ranging from 4 Ω to 20 MΩ. Because the interface electronics can just as easily convert any resistance into a voltage, a 20 MΩ thermistor is not more sensitive than a 30 Ω thermistor. In this situation, it makes more sense to define *fractional sensitivity* as

$$\alpha = \frac{1}{R} \cdot \frac{\partial R}{\partial T} \quad (1/\text{°C}) \quad \text{or} \quad \alpha = \frac{1}{y} \cdot \frac{\partial y}{\partial x}$$

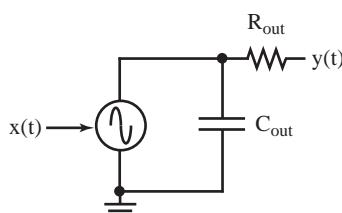
**Checkpoint 12.3:** Why is it better for the transducer to have a higher sensitivity?

**12.2.1.3 Specificity** Unfortunately, transducers are often sensitive to factors other than the signal of interest. Environmental issues involve how the transducer interacts with its external surroundings (e.g., temperature, humidity, pressure, motion, acceleration, vibration, shock, radiation fields, electric fields, magnetic fields). *Specificity* is a measure of relative sensitivity of the transducer to the desired signal compared to the sensitivity of the transducers to these other unwanted influences. A transducer with a good specificity will respond only to the signal of interest and be independent of these disturbing factors. On the other hand, a transducer with a poor specificity will respond to the signal of interest as well as to some of these disturbing factors. If all of these disturbing factors are grouped together as noise, then the *signal-to-noise (S/N) ratio* is a quantitative measure of the specificity of the transducer.

**12.2.1.4 Transducer Impedance** The *input range* is the allowed range of input,  $x$ . The input impedance is the phasor equivalent of the steady-state sinusoidal effort (voltage, force, chemical concentration, pressure) input variable divided by the phasor equivalent of steady-state flow (current, velocity, molecular flux flow) input variable. The output signal strength of the transducer can be specified by the output resistance  $R_{\text{out}}$  and output capacitance  $C_{\text{out}}$  (Figure 12.5).

**Figure 12.5**

Output model of a transducer.



The input impedance of a thermal sensor is a measure of the thermal perturbation that occurs due to the presence of the probe itself in the medium. For example, a thermocouple needle inserted into a laser-irradiated medium will affect the medium temperature because heat will conduct down the stainless steel shaft. A thermocouple has a low input impedance (which is bad) because the transducer itself loads (reduces) the medium temperature. On the other hand, an infrared detector measures surface medium temperature without physical contact. Infrared detectors therefore have a very high input impedance (which is good) because the presence of the transducer has no effect on the temperature to be measured. In the case of temperature sensors, the driving force for heat transfer is the

temperature difference  $\Delta T$ . The resulting heat flow  $q$  can be expressed using Fourier's law of thermal conduction:

$$q = KA \frac{\partial T}{\partial x} < K 4\pi a^2 \frac{\Delta T}{a}$$

where  $K$  is the probe thermal conductivity,  $A$  is the probe surface area, and  $a$  is the radius of a spherical transducer. The steady-state input impedance of a spherical temperature probe can thus be approximated by

$$Z = \frac{\Delta T}{q} \approx \frac{1}{4\pi a K}$$

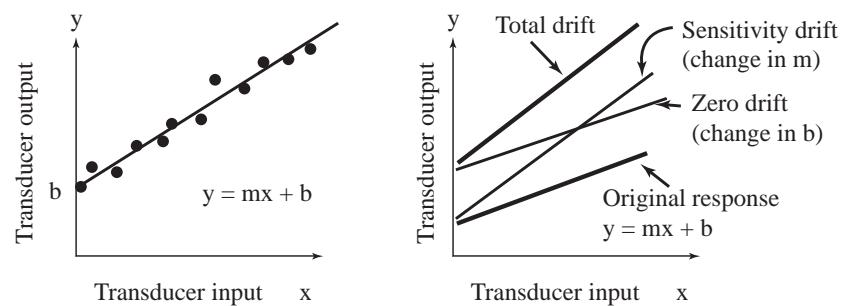
Again, the approximation in the above equation assumes a spherical transducer. Similar discussions can be constructed for the sinusoidal input impedance. Since most thermal events can be classified as step events rather than as sinusoidally varying events, most researchers prefer the use of a time constant to describe the transient behavior of temperature transducers.

Some transducers are completely passive (e.g., thermocouples, EKG electrodes), and others are active, requiring external power (e.g., ultrasonic crystals, strain gages, microphones, thermistors). Electric isolation is a critical factor in medical instrumentation. Some transducers are inherently isolated (e.g., thermistors, thermocouples, microphones), while others are not isolated (e.g., EKG electrodes, pacemakers, blood pressure catheters). Minimization of errors is important for all instruments. The sensitivity to disturbing factors (electric fields, magnetic fields, radiation, vibration, shock, acceleration, temperature, humidity) must be determined before a device can be used.

#### 12.2.1.5 Transducer Drift

The *zero drift* is the change in the static sensitivity curve intercept  $b$  as a function of time or other factor (see Figure 12.6). The *sensitivity drift* is the change in the static sensitivity curve slope  $m$  as a function of time or some other factor. These drift factors determine how often the transducer must be calibrated. For example, thermistors have a drift much larger than that of RTDs or thermocouples. Transducers may be aged at high temperatures for long periods of time to improve their stability.

**Figure 12.6**  
The two types of transducer drift: sensitivity drift and zero drift.



**Checkpoint 12.4:** How does transducer drift affect accuracy?

#### 12.2.1.6 Manufacturing Issues

This subsection discusses issues involved in the manufacturing of the system. The transducer is often a critical device, affecting both the cost and performance of the entire system. A higher quality transducer may produce better signals but at an increased cost. An important manufacturing issue is the availability of components. The availability of a device may be enhanced by having a *second source* (more than one manufacturer that produces the device).

The use of standard cables and connectors will simplify the construction of your system. The power requirements, size, and weight of the device are important in some systems and thus should be considered when a transducer is selected. Some transducers require calibration, which can greatly increase the cost of manufacturing.

### 12.2.2 Dynamic Transducer Specifications

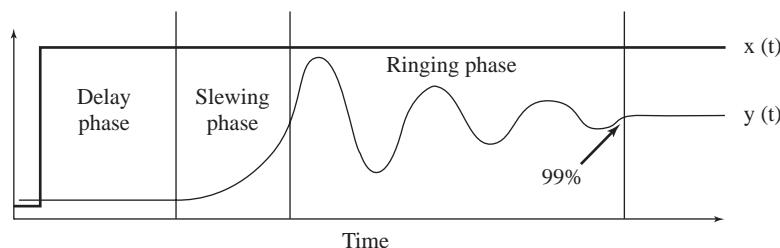
The *transient response* is the combination of the *delay phase*, the *slewing phase*, and the *ringing phase* (Figure 12.7). The total transient response is the time for the output  $y(t)$  to reach 99% of its final value after a step change in input,  $x(t) = u_0(t)$ .

The transient response of a temperature transducer to a sudden change in signal input can sometimes be approximated by an exponential equation (assuming first-order response):

$$y(t) = y_f + (y_0 - y_f) e^{-t/\tau}$$

**Figure 12.7**

The step response often has delay, slewing, and ringing phases.



where  $y_0$  and  $y_f$  are the initial and final transducer outputs, respectively. The *time constant*  $\tau$  of a transducer is the time to reach 63.2% of the final output after the input is instantaneously increased. This time is dependent on both the transducer and the experimental setup. Manufacturers often specify the time constant of thermistors and thermocouples in well-stirred oil (fastest) or still air (slowest). In your applications, you must consider the situation. If the transducer is placed in a high-flow liquid like an artery or a water pipe, it may be reasonable to use the stirred-oil time constant. If the transducer is in air or embedded in a solid, then thermal conduction in the medium will determine the time constant almost independently of the transducer.

The *frequency response* is a standard technique to describe the dynamic behavior of linear systems. Let  $y(t)$  be the system response to  $x(t)$ . Let

$$x(t) = A \sin(\omega t) \quad y(t) = B \sin(\omega t + \phi) \quad \omega = 2\pi f$$

The magnitude  $B/A$  and the phase  $\phi$  responses are both dependent on frequency. Differential equations can be used to model linear transducers. Let  $x(t)$  be the time domain input signal,  $X(j\omega)$  be the frequency domain input signal,  $y(t)$  be the time domain output signal, and  $Y(j\omega)$  be the frequency domain output signal (Table 12.4).

Classification	Differential equation	Gain response	Phase response
Zero-order	$y(t) = m x(t)$	$Y/X = m = \text{static sensitivity}$	
First-order	$y'(t) + a y(t) = b x(t)$	$Y/X = \frac{b}{\sqrt{a^2 + \omega^2}}$	$\text{Phase} = \arctan(-\omega/a)$
Second-order	$y''(t) + a y'(t) + b y(t) = c x(t)$		
Time delay	$y(t) = x(t-T)$	$Y/X = \exp(-j\omega T)$	

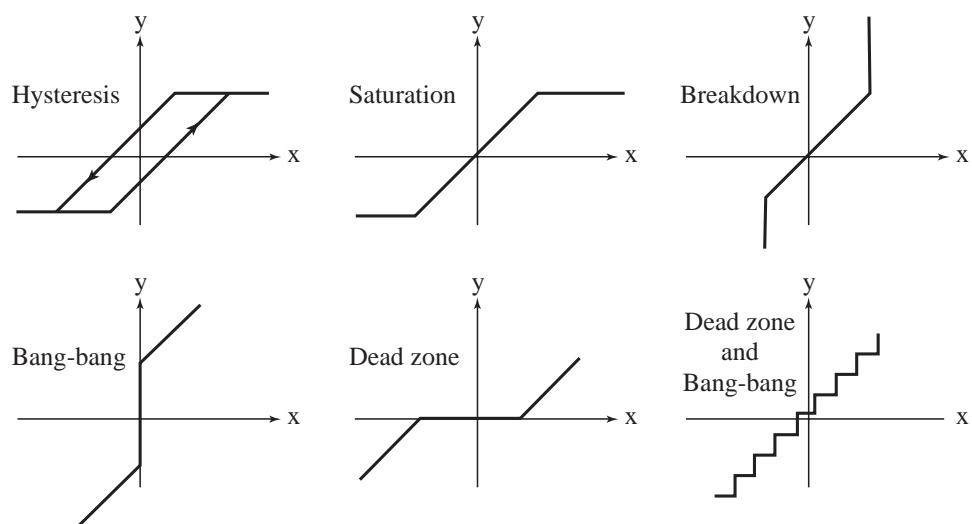
**Table 12.4**

Classifications of simple linear systems.

### 12.2.3 Nonlinear Transducers

Nonlinear characteristics include hysteresis, saturation, bang-bang, breakdown, and dead zone. *Hysteresis* is created when the transducer has memory. We can see in the Figure 12.8 that when the input was previously high it falls along the higher curve, and when the input was previously low it follows along the lower curve. Hysteresis will cause a measurement error, because for any given sensor output  $y$ , there may be two possible measurand inputs. *Saturation* occurs when the input signal exceeds the useful range of the transducer. With saturation, the sensor does not respond to changes in input value when the input is either too high or too low. *Breakdown* describes a second possible result that may occur when the input exceeds the useful range of the transducer. With breakdown, the sensor output changes rapidly, usually the result of permanent damage to the transducer. Hysteresis, bang-bang, and dead zone all occur within the useful range of the transducer. *Bang-bang* is a sudden large change in the output for a small change in the input. If the bang-bang occurs predictably, then it can be corrected for in software. A *dead zone* is a condition where a large change in the input causes little or no change in the output. Dead zones cannot be corrected for in software; thus, if present, they will cause measurement errors.

**Figure 12.8**  
Nonlinear transducer responses.



There are many ways to model nonlinear transducers. A nonlinear transducer can be described as a piecewise-linear system. The first step is to divide the range of  $x$  into finite subregions, assuming the system is linear in each subregion. The second step is to solve the coupled linear systems so that the solution is continuous. Another method to model a nonlinear system is to use empirically determined nonlinear equations. The first step in this approach is to observe the transducer response experimentally. Given a table of  $x$  and  $y$  values, the second step is to fit the response to a nonlinear equation.

A third approach to model a nonlinear transducer uses a look-up table located in memory. This method is convenient and flexible. Let  $x$  be the measurand and  $y$  be the transducer output. The table contains  $x$  values, and the measured  $y$  value is used to index into the table. Sometimes a small table coupled with linear interpolation achieves equivalent results to a large table. There are two examples of this approach as part of the TExaS application called TBL.RTF and ETBL.RTF.

### 12.2.4 Position Transducers

One of the simplest methods to convert position into an electric signal uses a position-sensitive potentiometer. These devices are inexpensive to build and are sensitive to small displacements. The transducer is constructed by wrapping a fine uninsulated wire around

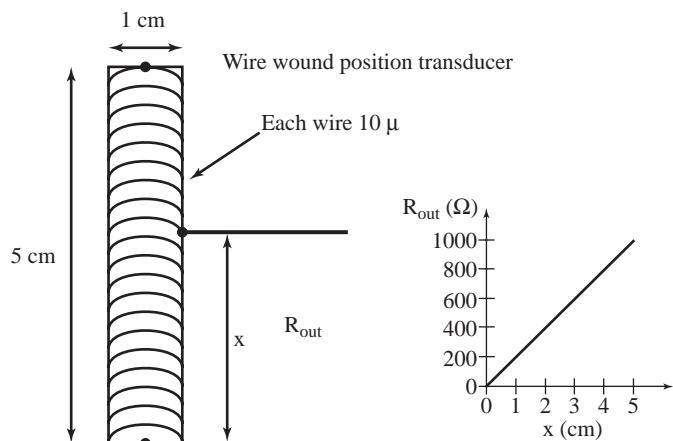
an insulating cylinder. The wires are close to one another, but do not touch. The total electric resistance of the transducer is given by

$$R = \rho L/A$$

where  $\rho$  is the resistivity of the wire,  $A$  is the cross-sectional area of the wire, and  $L$  is the length of the wire. A potentiometer is formed by placing a wiper blade that makes electric contact on the wires. The blade is free to move up and down along the axis of the cylinder. If the wire has been uniformly wrapped, then  $R_{out}$  will be linearly related to displacement  $x$ . The disadvantages of this transducer are its low-frequency response, its high mechanical input impedance, and its degeneration with time. Nevertheless, this type of transducer is adequate for many applications. This transducer will be interfaced two ways later in Section 12.5.5.

Given the fact that this transducer in actuality has multiple discrete outputs (i.e.,  $R_{out}$  versus  $x$  is not linear but has multiple bang-bangs and dead zones), what is the maximum number of ADC bits that can be used? For a discrete wire potentiometer like the one shown in Figure 12.9, the maximum precision is determined by the number of times the wire is wrapped around the post. In other words, the resistance cannot change until the variable arm moves enough to jump to the next wire. For a continuous solid potentiometer there is no fundamental limit.

**Figure 12.9**  
Potentiometer-based position sensor.



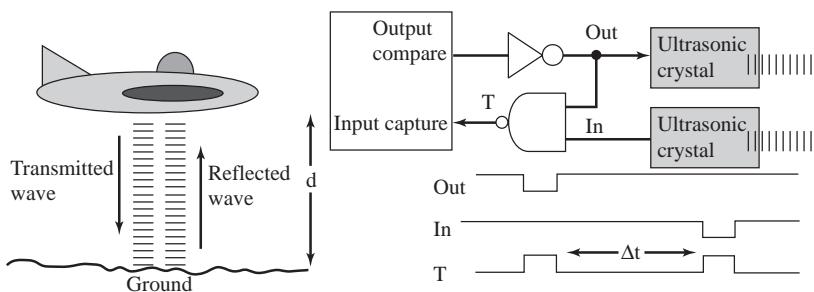
Similar transducers can be constructed using variable capacitors and inductors. The linear variable differential transducer (LVDT) uses a ferrite core to modify the mutual inductance between the single active primary coil and the two passive secondary coils. The primary coil is excited with an AC signal. The ferrite core is allowed to move up and down through the middle of the three inductors. If the ferrite core is midway between the two secondaries (S1, S2), then the AC voltage across S1 will equal the AC voltage across S2, and the DC output  $V_{out}$  will be zero. If the ferrite core is closer to the secondary S1 ( $x > 0$ ), then the AC voltage across S1 will be larger than the AC voltage across S2, and the DC output  $V_{out}$  will be positive. If the ferrite core is closer to the secondary S2 ( $x < 0$ ), then the AC voltage across S2 will be larger than the AC voltage across S1, and the DC output  $V_{out}$  will be negative. An ADC converter is required to generate a digital signal proportional to displacement.

A method to measure the distance between two objects is to transmit a sound wave from one object at the other and listen for the reflection (Figure 12.10). The instrument must be able to generate the sound pulse, hear the echo, and measure the time,  $\Delta t$ , between pulse and echo. If the speed of sound,  $c$ , is known, then the distance,  $d$ , can be calculated. Our microcontrollers also have mechanisms to measure the pulse width  $\Delta t$ . An example of this type of sensor is the **Ping**))) available at [www.parallax.com](http://www.parallax.com).

$$d = c\Delta t/2$$

**Figure 12.10**

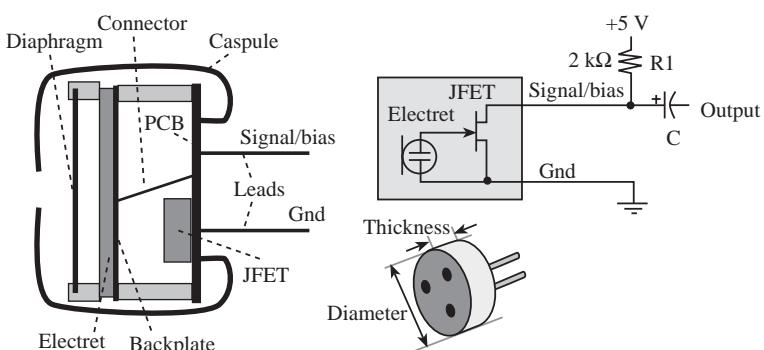
An ultrasonic pulse-echo transducer measures the distance to an object.



A *microphone* is a type of displacement transducer. Sound waves, which are pressure waves travelling in air, cause a diaphragm to vibrate, and the diaphragm motion causes the distance between capacitor plates to change. This variable capacitance creates a voltage, which can be amplified and recorded. The *electret condenser microphone* (ECM) is an inexpensive choice for converting sound to analog voltage. Electret microphones are used in consumer and communication audio devices because of their low cost and small size. For applications requiring high sensitivity, low noise, and linear response, we could use the dynamic microphone, like the ones used in high-fidelity audio recording equipment. The ECM capsule acts as an acoustic resonator for the capacitive electret sensor shown in Figure 12.11. The ECM has a *junction field effect transistor* (JFET) inside the transducer, providing some amplification. This JFET requires power as supplied by the R1 resistor. This local amplification allows the ECM to function with a smaller capsule than typically found with other microphones. ECM devices are cylindrically shaped, have a diameter ranging from 3 to 10 mm, and have a thickness ranging from 1 to 5 mm.

**Figure 12.11**

Physical and electrical view of an ECM with JFET buffer.



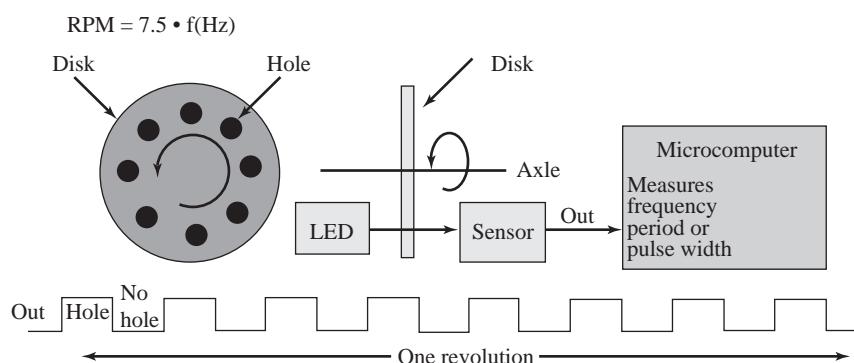
An ECM consists of a pre-charged, nonconductive membrane between two plates that form a capacitor. The backplate is fixed, and the other plate moves with sound pressure. Movement of the plate results in a capacitance change, which in turn results in a change in voltage due to the nonconductive, pre-charged membrane. An electrical representation of such an acoustic sensor consists of a signal voltage source in series with a source capacitor. The most common method of interfacing this sensor is a high-impedance buffer/amplifier. A single JFET with its gate connected to the sensor plate and biased as shown in Figure 12.11 provides buffering and amplification. The capacitor C provides high-pass filtering, so the voltage at the output will be less than  $\pm 100$  mV for normal voice. Audio microphones need additional amplification and band-pass filtering. Typical audio signals exist from 100 Hz to 10 kHz. The presence of the R1 resistor is called “phantom biasing.” The electret has two connections: Gnd and Signal/bias. Typically, the metallic capsule is connected to Gnd.

## 12.2.5 Velocity Measurements

One can use a LED-photosensor pair to measure the angular velocity of a rotating shaft (Figure 12.12). The circular disk is mounted on the rotating axle. The disk has eight equally spaced holes. The LED and sensor are positioned on opposite sides of the disk. The sensor output, Out, will be high when light passes through one of the holes in the disk. Out will

**Figure 12.12**

A LED-photosensor pair measures shaft rotation.



#### 12.2.5.1

##### Velocity Transducers

be low when no light hits the sensor. If the shaft is rotating at a constant velocity, then Out will be a square wave. The rotations per minute (RPM) of the shaft can be calculated by measuring the frequency of the square wave,  $f$ , in hertz. The frequency can also be indirectly calculated from measurement of period or pulse width. Datasheets for this type of sensor can be found on the CD within the “sensor” directory.

$$\text{RPM} = 7.5 \cdot f \quad (\text{Hz})$$

**Checkpoint 12.5:** How was the 7.5 calculated? In what way is 7.5 important when designing a system to satisfy a velocity resolution specification?

#### 12.2.5.2

##### Velocity Calculations

Given a method to measure position, we can use calculus to determine velocity and acceleration.

$$v(t) = \frac{dx(t)}{dt} \quad a(t) = \frac{dv(t)}{dt}$$

The continuous derivative can be expressed as a limit.

$$v(t) = \lim \frac{x(t + \Delta t) - x(t)}{\Delta t} \quad \Delta t \rightarrow 0$$

The microcomputer can approximate the velocity by calculating the discrete derivative.

$$v(t) = \frac{x(t) - x(t - \Delta t)}{\Delta t}$$

where  $\Delta t$  is the time between velocity samples. This calculation of derivative is very sensitive to errors in the measurement of  $x(t)$ . A more stable calculation averages two or more derivative terms taken over different time windows. In general, we can define such a robust, calculation as

$$v(t) = \frac{a}{a+b} \frac{x(t) - x(t - n\Delta t)}{n\Delta t} + \frac{b}{a+b} \frac{x(t - c\Delta t) - x(t - (c+m)\Delta t)}{m\Delta t}$$

If the integers  $n$ ,  $m$ , and  $c$  are all positive, this calculation can be performed in real time. The first term is the derivative over the large time window of  $n\Delta t$ . The second window term has a smaller size of  $m\Delta t$ . It normally fits entirely inside the first with  $c > 0$  and  $c - m < n$ . The coefficients  $a$ ,  $b$  create the weight for combining the short and long intervals. With  $a = b = 1$ ,  $n = 3$ ,  $m = 1$ , and  $c = 1$ , we get

$$v(t) = \frac{1}{2} \frac{x(t) - x(t - 3\Delta t)}{3\Delta t} + \frac{1}{2} \frac{x(t - \Delta t) - x(t - 2\Delta t)}{\Delta t} = \frac{x(t) + 3x(t - \Delta t) - 3x(t - 2\Delta t) - x(t - 3\Delta t)}{6\Delta t}$$

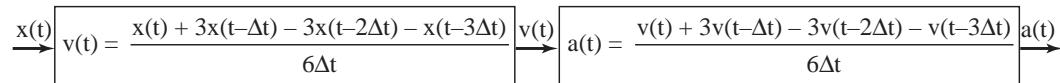
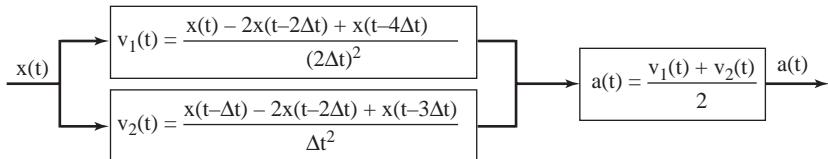
The acceleration can also be approximated by a discrete derivative.

$$a(t) = \frac{x(t) - 2x(t - \Delta t) + x(t - 2\Delta t)}{\Delta t^2}$$

To make a more stable calculation of second derivative, you could average two or more second-derivative terms taken over different time windows (Figure 12.13).

Another robust second-derivative calculation concatenates two robust first-derivative calculations (Figure 12.14).

**Figure 12.13**  
Robust calculation of acceleration.



**Figure 12.14**  
Sequential calculation of acceleration.

**Observation:** In the above calculations of derivative, a single error in one of the  $x(t)$  input terms will propagate to only a finite number of the output calculations.

**Observation:** Although the central difference calculation of  $v(t) = [x(t + \Delta t) - x(t - \Delta t)] / 2\Delta t$  is theoretically valid, we cannot use it for real-time applications, because it requires knowledge about the future,  $x(t + \Delta t)$ , which is unavailable at the time  $v(t)$  is being calculated.

Similarly, we can perform integration of velocity to determine position.

$$x(t) = \int_0^t v(s) ds$$

The microcomputer can perform a discrete integration by summation.

$$x(t) = x(0) + \sum_{n=0}^t v(n) \Delta t$$

There are two problems with this approach. The first difficulty is determining  $x(0)$ . The second problem is the accumulation of errors. If one is calculating velocity from position and an error occurs in the measurement of  $x(t)$ , then that error affects only two calculations of  $v(t)$ . Unfortunately, if one is calculating position from velocity and an error occurs in the measurement of  $v(n)$ , then that error will affect all subsequent calculations of  $x(t)$ .

**Observation:** In the above calculation of integration, a single error in one of the  $x(t)$  input terms will propagate into all remaining output calculations.

The following function, which is quite similar to the derivative, is actually a low-pass digital filter. We will learn more about digital filters in Chapter 15.

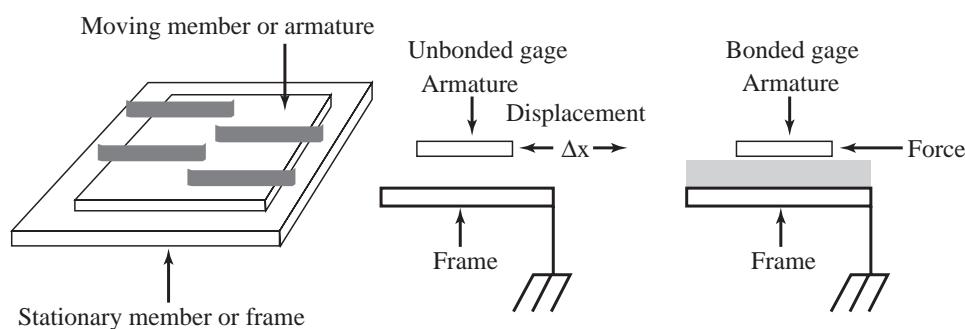
$$y(t) = \frac{x(t) + x(t - \Delta t)}{2}$$

## 12.2.6 Force Transducers

The most common device to measure force is the *strain gage* (Figure 12.15). As a wire is stretched, its length increases and its cross-sectional area decreases. These geometric changes cause an increase in the electric resistance of the wire,  $R$ . The transducer is constructed with four sets of wires mounted between a stationary member (frame) and a moving member (armature). As the armature moves relative to the frame, two wires are stretched (increase in  $R1, R4$ ), and two wires are compressed (decrease in  $R2, R3$ ). The strain gage is a displacement

**Figure 12.15**

Strain gages used for displacement or force measurement.



transducer such that a change in the relative position between the armature and frame,  $\Delta x$ , causes a change in resistance,  $\Delta R$ . The sensitivity of a strain gage is called its gage factor:

$$G = \frac{\Delta R/R}{\Delta x/x}$$

The gage factor for an Advance strain gage is 2.1. The typical resistance  $R$  is  $120 \Omega$ . If the gage is bonded onto a material with a spring characteristic

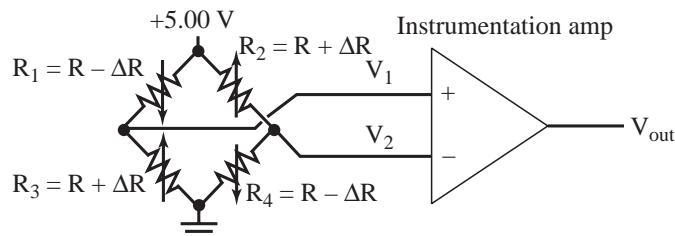
$$F = -kx$$

then the transducer can be used to measure force. The wires each have a significant temperature drift. When the four wires are placed into a bridge configuration, the temperature dependence cancels (Figure 12.16). A high-gain, high-input impedance, high-CMRR differential amplifier is required.

**Checkpoint 12.6:** How can a force transducer be used to measure pressure?

**Figure 12.16**

Four strain gauges are placed in a bridge configuration.



## 12.2.7 Temperature Transducers

### 12.2.7.1 Classification of Temperature Transducers

An increase in thermal energy causes an increase in the average spacing between molecules. For an ideal gas the kinetic energy [ $(\frac{1}{2}) mv^2$ ] equals the thermal energy [ $(\frac{1}{2}) kT$ ]. For liquids and most gases, an increase in kinetic energy is also accompanied by an increase in potential energy. The mercury thermometer and bimetallic strips are common transducers that rely on thermal expansion. A RTD is constructed from long narrow-gage metal wires. Over a moderate range of temperature (e.g.,  $50^\circ\text{C}$ ) the wire resistance is approximately linearly related to its temperature.

$$R = R_0 [1 + \alpha(T - T_0)]$$

The sensitivity  $\alpha$  for various metals is given in Table 12.5.  $R_0$  is a function of the resistivity and geometry of the wire. Although many metals such as gold, nichrome (nickel-chromium), nickel, and silver can be used, platinum is typically selected because of its excellent stability.

**Table 12.5**  
Electric properties of various materials.

Material	$\alpha$ , fractional sensitivity ( $1/\text{ }^\circ\text{C}$ )	$\rho$ , electric resistivity ( $\Omega \cdot \text{cm}$ )
Gold	+0.004	$2.35 \cdot 10^{-6}$
Nickel	+0.0069	$6.84 \cdot 10^{-6}$
Platinum	+0.003927	$10^{-5}$
Copper	0.0068	$1.59 \cdot 10^{-6}$
Silver	+0.0041	$1.673 \cdot 10^{-6}$
NTC thermistor	-0.04	$10^3$
PTC thermistor	+0.1	

The rates of first-order processes are proportional to the Boltzmann factor,  $e^{-E/kT}$ . This strong temperature dependence can be exploited to make measurements. Rates that are governed by the Boltzmann factor include the evaporation of liquids and the population of charge carriers in a conduction band. The conductance (1/resistance) of a thermistor is proportional to the occupation of charge carriers in the conduction band:

$$G = G_0 e^{-E/kT}$$

The electric resistance is the reciprocal of the conductance:

$$R = R_0 e^{+E/kT} = R_0 e^{+ / T}$$

where  $\gamma$  is  $E/k$ .

A thermocouple is constructed using two wires of different metals welded to form two junctions. One junction is placed at a known reference temperature (e.g.,  $0^\circ\text{C}$  ice in thermal equilibrium with liquid and gaseous water at 1 atmosphere), and the other junction is used to measure the unknown temperature. The Seebeck effect involves thermal-to-electric energy conversion. In a closed-loop configuration, the current around the loop is proportional to the temperature difference between the two junctions. In the open-loop configuration, a voltage is generated proportional to the temperature difference:

$$V = a(T_2 - T_1) + b(T_2 - T_1)^2$$

Electromagnetic radiation is emitted by all materials above absolute zero, of which black or gray bodies are a special case. The spectral properties of a surface are a function of the material and its temperature. The total wide-band radiation power  $W$  is a function of the fourth power of surface temperature.

$$W = (T^4 - T_0^4)$$

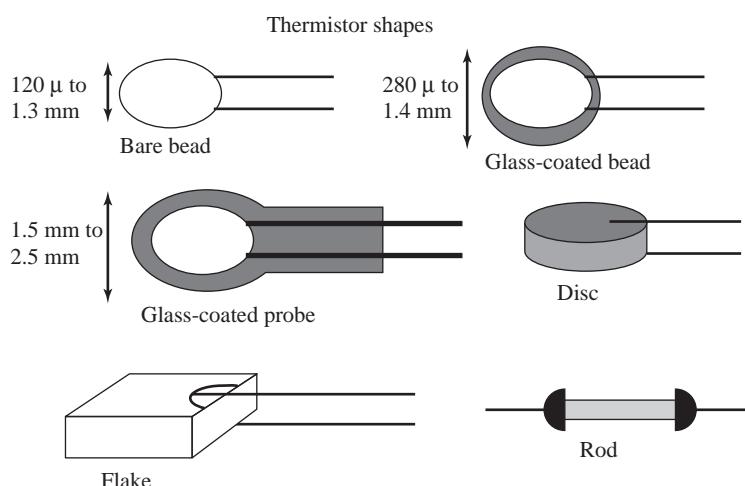
The sudden transition of state that occurs for a particular substance can be used to measure temperature. In particular, this property is frequently exploited to produce temperature references. An ice bath at  $0^\circ\text{C}$  (water's triple point) is often used as the reference junction of a thermocouple. The melting point of gallium is conveniently  $29.7714^\circ\text{C}$ . A triple point occurs under pressure when a substance exists simultaneously in all three states: gas, liquid, and solid. The triple point of water is  $0.01 \pm 0.0005^\circ\text{C}$  at 1 atmosphere, the triple point of rubidium is  $39.265^\circ\text{C}$ , and the triple point of succinonitrile is  $58.0805^\circ\text{C}$ .

### 12.2.7.2 Thermistors

*Thermistors* are a popular temperature transducer made from a ceramic-like semiconductor (Figure 12.17). A negative-temperature coefficient (NTC) thermistor is made from combinations of metal oxides of manganese, nickel, cobalt, copper, iron, and titanium. A mixture of milled semiconductor oxide powders and a binder is shaped into the desired geometry. The mixture is dried and sintered (under pressure) at an elevated temperature.

**Figure 12.17**

Thermistors come in many shapes and sizes.



The wire leads are attached and the combination is coated with glass or epoxy. By varying the mixture of oxides, a range of resistance values from  $30 \Omega$  to  $20 M\Omega$  (at  $25^\circ C$ ) is possible. Table 12.6 lists the trade-offs between thermistors and thermocouples. Two thermistor design spreadsheets can be found on the book Web site called therm.xls and therm12.xls.

**Table 12.6**

Trade-offs between thermistors and thermocouples.

Thermistors	Thermocouples
More sensitive	More sturdy
Better temperature resolution	Faster response
Less susceptible to noise	Inert, interchangeable V versus T curves
Less thermal perturbation	Requires less frequent calibration
Does not require a reference	More linear

A precision thermometer, an ohmmeter, and a water bath are required to calibrate thermistor probes. The following empirical equation yields an accurate fit over a wide range of temperature:

$$T = \frac{1}{H_0 + H_1 \ln(R) + H_3 [\ln(R)]^3} - 273.15$$

where  $T$  is the temperature in  $^\circ C$  and  $R$  is the thermistor resistance in  $\Omega$ . The cubic term was added to the above equation to improve accuracy. It is preferable to use the ohmmeter function of the eventual instrument for calibration purposes so that influences of the resistance measurement hardware and software are incorporated into the calibration process.

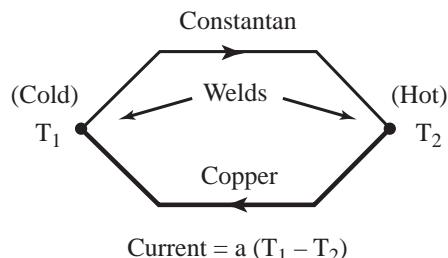
#### 12.2.7.3 Thermocouples

A *thermocouple* is constructed by spot welding two different metal wires together. Probe transducers include a protective casing that surrounds the thermocouple junction. Probes come in many shapes including round tips, conical needles, and hypodermic needles. Bare thermocouple junctions provide faster response but are more susceptible to damage and noise pickup. Ungrounded probes allow electric isolation but are not as responsive as grounded probes. Commercial thermocouples have been constructed in 16 to 30 gage hypodermic needles—a 30 gage needle has an outside diameter of above 0.03 cm. Bare thermocouples can be made from  $30 \mu m$  wire, producing a tip with an  $80 \mu m$  diameter. A spot weld is produced by passing a large current through the metal junction that fuses the two metals together.

If the wires form a loop, and the junctions are at different temperatures, then a current will flow in the loop. This thermal-to-electric energy conversion is called the Seebeck effect (see Figure 12.18). If the temperature difference is small, then the current  $I$  is linearly proportional to the temperature difference  $T_1 - T_2$ .

**Figure 12.18**

When the two thermocouple junctions are at different temperatures, current will flow.

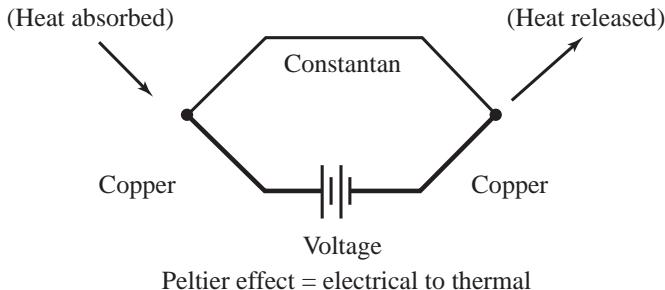


Seebeck effect = thermal to electrical

If the loop is broken, and an electric voltage is applied, then heat will be absorbed at one junction and released at the other. This electric-to-thermal energy conversion is called the Peltier effect (see Figure 12.19). If the voltage is small, then the heat transferred is linearly proportional to the voltage  $V$ .

**Figure 12.19**

When voltage is applied to two thermocouple junctions, heat will flow.



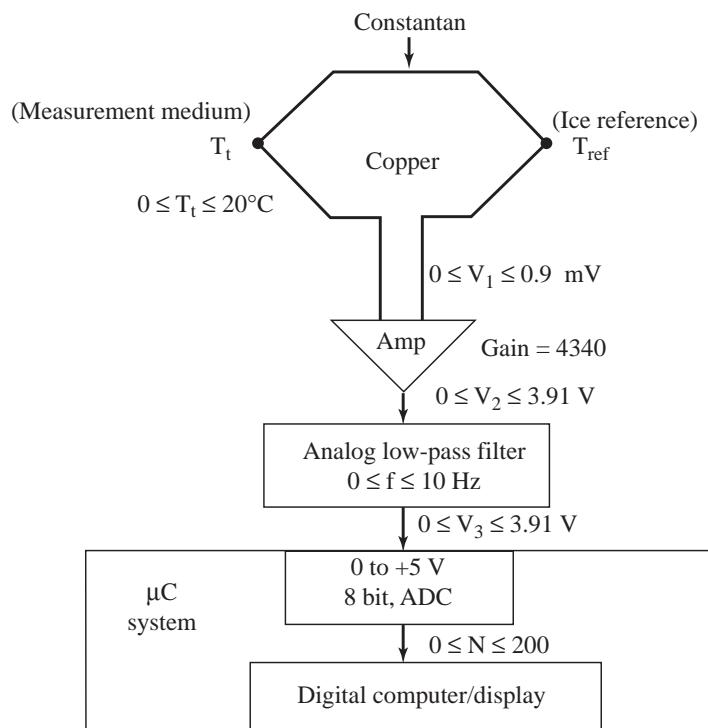
If the loop is broken and the junctions are at different temperatures, then a voltage will develop because of the Seebeck effect. If the temperature difference is small, then the voltage  $V$  is nearly linearly proportional to the temperature difference  $T_1 - T_2$ . Thermocouples are characterized by (1) low impedance (resistance of the wires), (2) low temperature sensitivity ( $45 \mu\text{V}/^\circ\text{C}$  for copper/constantan), (3) low power dissipation, (4) fast response (because of the metal), (5) high stability (because of the purity of the metals), and (6) interchangeability (again because of the physics and the purity of the metals).

Figure 12.20 shows a simple approach to measure temperature using a thermocouple. Typically, an ice bath ( $T_{\text{ref}} = 0^\circ\text{C}$ ) is used for the reference junction. A high-gain differential amplifier is used to convert the low-level voltage (e.g., 0 to 0.9 mV) into the range of the ADC (e.g., 0 to 5 V). Table 12.7 gives the sensitivities and temperature ranges of typical thermocouple devices. The amplifier gain is chosen to provide a simple relationship between the A/D output  $N$  and the temperature of the medium  $T_t$ . If a linear relationship between  $T_t$  and  $V_1$  is assumed,  $T_t = N/10$ .

If the temperature range is less than  $25^\circ\text{C}$ , then the linear approximation can be used to measure temperature. Let  $N$  be the digital sample from the ADC for the unknown medium temperature  $T_1$ . A calibration is performed under the conditions of a constant reference temperature: typically, one uses the extremes of the temperature range ( $T_{\text{min}}$  and  $T_{\text{max}}$ ). A precision thermometer system is used to measure the “truth” temperatures. Let

**Figure 12.20**

An instrumentation and low-pass filter are used to interface a thermocouple.



Type—Thermocouple	$\mu\text{V}/^\circ\text{C}$ at $20^\circ\text{C}$	Useful range, $^\circ\text{C}$	Comments
T—Copper/constantan	45	−150 to +350	Moist environment
J—Iron/constantan	53	−150 to +1000	Reducing environment
K—Chromel/alumel	40	−200 to +1200	Oxidizing environment
E—Chromel/constantan	80	0 to +500	Most sensitive
R S—Platinum/platinum-rhodium	6.5	0 to +1500	Corrosive environment
C—Tungsten/rhenium	12	0 to +2000	High temperature

**Table 12.7**

Temperature sensitivity and range of various thermocouples.

$N_{min}$  and  $N_{max}$  be the digital samples at  $T_{min}$  and  $T_{max}$ , respectively. Then the following equation can be used to calculate the unknown medium temperature from the measured digital sample:

$$T_1 = T_{min} + (N - N_{min}) \cdot \frac{T_{max} - T_{min}}{N_{max} - N_{min}}$$

Because the thermocouple response is not exactly linear, the errors in the above linear equation will increase as the temperature range increases. For instruments with a larger temperature range, a quadratic equation can be used,

$$T_1 = H_0 + H_1 \cdot N + H_2 \cdot N^2$$

where  $H_0$ ,  $H_1$ , and  $H_2$  are determined by calibration of the instrument over the range of interest. Linear regression can be used by letting  $z = T_1$ ,  $x = N$ , and  $y = N^2$ .

## 12.3 DAS Design

### 12.3.1 Introduction and Definitions

Before designing a DAS we must have a clear understanding of the system goals. We can classify a system as a *quantitative DAS* if the specifications can be defined explicitly in terms of desired range ( $r_x$ ), resolution ( $\Delta x$ ), precision ( $n_x$ ), and frequencies of interest ( $f_{\min}$  to  $f_{\max}$ ). If the specifications are more loosely defined, we classify it as a *qualitative DAS*. Examples of qualitative DASs include systems that mimic the human senses where the specifications are defined using terms like “sounds good,” “looks pretty,” and “feels right.” Other qualitative DASs involve the detection of events.

We will consider two examples: a burglar detector and an instrument to diagnose cancer. For binary detection systems required to detect the presence or absence of a burglar or the presence or absence of cancer, we define a true positive (TP) when the condition exists (there is a burglar) and the system properly detects it (the alarm rings). We define a false positive (FP) when the condition does not exist (there is no burglar) but the system thinks there is (the alarm rings). A false negative (FN) occurs when the condition exists (there is a burglar) but the system does not think there is (the alarm is silent). A true negative (TN) occurs when the condition does not exist (the patient does not have cancer) and the system properly detects it (the instrument says the patient is normal). **Prevalence** is the probability that the condition exists, sometimes called pre-test probability. In the case of diagnosing the disease, prevalence tells us what percentage of the population has the disease. **Sensitivity** is the fraction of properly detected events (a burglar comes and the alarm rings) over the total number of events (number of robberies). It is a measure of how well our system can detect an event. For the burglar detector, a sensitivity of 1 means when a burglar breaks in, the alarm will go off. For the diagnostic instrument, a sensitivity of 1 means every sick patient will get treatment. **Specificity** is the fraction of properly handled nonevents (a patient doesn't have cancer and the instrument claims the patient is normal) over the total number of nonevents (the number of normal patients). A specificity of 1 means no people will be treated for a cancer they don't have. The **positive predictive value** of a system (PPV) is the probability that the condition exists when restricted to those cases in which the instrument says it exists. It is a measure of how much we believe the system is correct when it says it has detected an event. A PPV of 1 means when the alarm rings, the police will come and arrest a burglar. Similarly, a PPV of 1 means if our instrument says a patient has the disease, then that patient is sick. The **negative predictive value** of a system (NPV) is the probability that the condition doesn't exist when restricted to those cases where the instrument says it doesn't exist. A NPV of 1 means if our instrument says a patient doesn't have cancer, then that patient is not sick. Sometimes the true negative condition doesn't really exist (how many times a day does a burglar not show up at your house?). For these situations, only sensitivity and PPV are relevant.

$$\text{Prevalence} = \frac{\text{TP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad \text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad \text{NPV} = \frac{\text{TN}}{\text{TN} + \text{FN}}$$

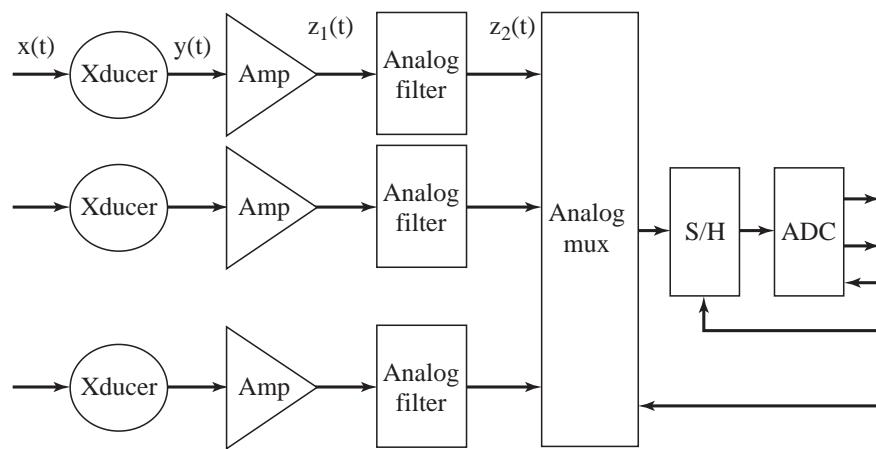
**Checkpoint 12.7:** Explain how a design decision can involve a trade off between sensitivity and PPV?

The *transducer* converts the physical signal into an electric signal. The *amplifier* converts the weak transducer electric signal into the range of the ADC (e.g., 0 to +5 V). The *analog filter* removes unwanted frequency components within the signal. The analog filter is required to remove aliasing error caused by the ADC sampling. The *analog multiplexer* is

used to select one signal from many sources. The *S/H* is an analog latch used to keep the ADC input voltage constant during the ADC conversion. The *clock* is used to control the sampling process. Inherent in digital signal processing is the requirement that the ADC be sampled on a fixed time basis. The *computer* is used to save and process the digital data. A *digital filter* may be used to amplify or reject certain frequency components of the digitized signal. The MACQ is a convenient data structure to use with the digital filter (Figure 12.21).

**Figure 12.21**

Block diagram showing how an analog multiplexer is used to sample multiple signals.



### 12.3.2 Using Nyquist Theory to Determine Sampling Rate

There are two errors introduced by the sampling process. *Voltage quantizing* is caused by the finite word size of the ADC. The *precision* is determined by the number of bits in the ADC. If the ADC has  $n$  bits, then the number of distinguishable alternatives is

$$n_z = 2^n$$

*Time quantizing* is caused by the finite discrete sampling interval. *Nyquist theory* states that if the signal is sampled at  $f_s$ , then the digital samples contain frequency components from only 0 to  $0.5 f_s$ . Conversely, if the analog signal does contain frequency components larger than  $0.5 f_s$ , then there will be an *aliasing* error. Aliasing is when the digital signal appears to have a different frequency than the original analog signal.

Simply put, if one samples a sine wave at a sampling rate of  $f_s$ ,

$$V(t) = A \sin(2\pi ft + \phi)$$

is it possible to determine  $A$ ,  $f$ , and  $\phi$  from the digital samples? Nyquist theory says that if  $f_s$  is strictly greater than twice  $f$ , then one can determine  $A$ ,  $f$ , and  $\phi$  from the digital samples. In other words, the entire analog signal can be reconstructed from the digital samples. But if  $f_s$  is less than or equal to twice  $f$ , then one cannot determine  $A$ ,  $f$ , and  $\phi$ . In this case, the apparent frequency, as predicted by analyzing the digital samples, will be shifted to a frequency between 0 and  $0.5 f_s$ .

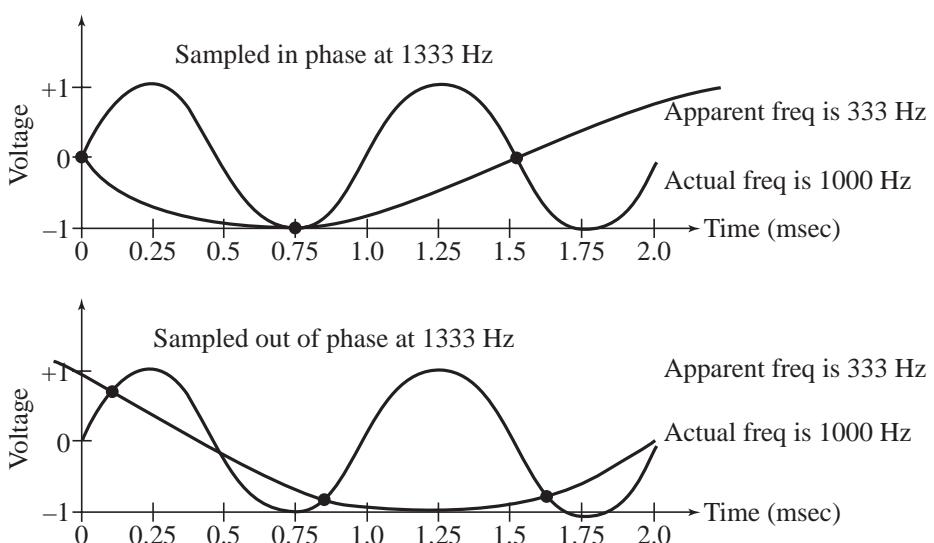
In this first example, the frequency of an input sine wave at 1000 Hz and the sampling rate  $f_s$ , is varied

$$V = \sin(2\pi \cdot 1000 \cdot t)$$

where  $t$  is in seconds. In this case the largest (and only) frequency component  $f_{max}$  is 1000 Hz. If the signal is sampled at 1333 Hz ( $f_s \leq 2 f_{max}$ ), then an aliasing error will occur (Figure 12.22). This error occurs regardless of the phase between the signal and the ADC sampling. Notice that the apparent frequency of the digital samples (333 Hz) is different from the actual frequency of 1000 Hz.

**Figure 12.22**

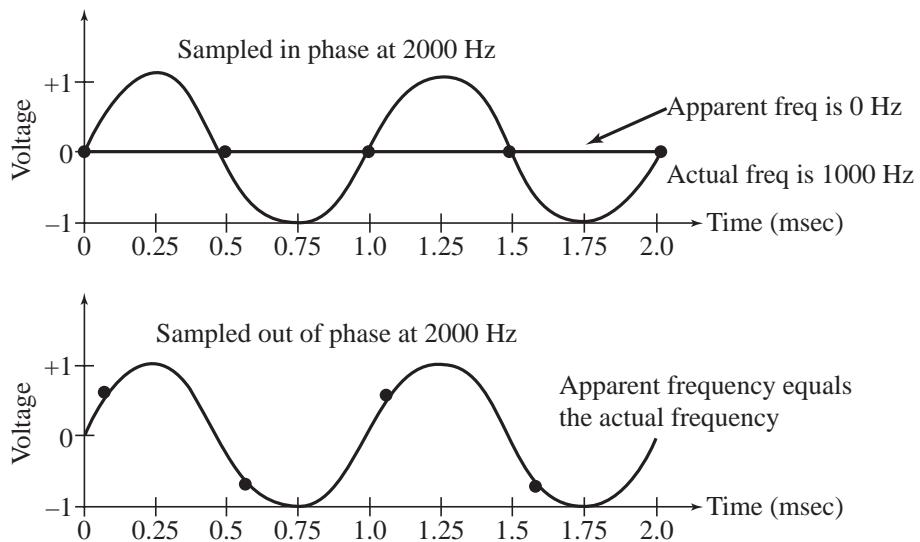
Aliasing makes the 1000 Hz signal appear as a 333 Hz signal.



If the 1000 Hz sine wave is sampled in phase at 2000 Hz, then the digital samples appear as a constant (0 Hz). When the sampling frequency is exactly equal to twice the input frequency, the aliasing error is dependent on the phase between the signal and the ADC sampling (Figure 12.23).

**Figure 12.23**

Right at the Nyquist frequency aliasing may or may not occur.

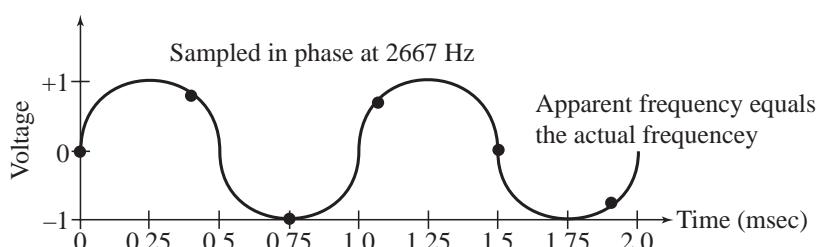


If the 1000 Hz sin wave is sampled above 2000 Hz, then the frequency, magnitude, and phase of the signal can be reconstructed from the digital samples. This reconstruction can be performed regardless of the phase between the signal and the ADC sampling (Figure 12.24).

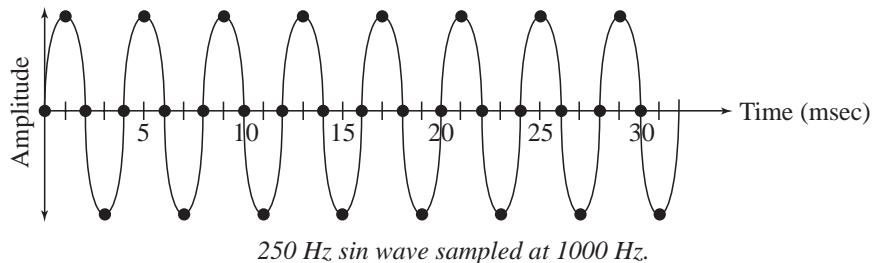
In this next example the sampling rate  $f_s$  will be fixed at 1000 Hz, and the frequency of the input signal will be varied. In the first two cases, the sampling rate is more than twice the input frequency, so the original signal can be properly reconstructed (Figure 12.25).

**Figure 12.24**

Aliasing does not occur when the sampling rate is more than twice the signal frequency.

**Figure 12.25**

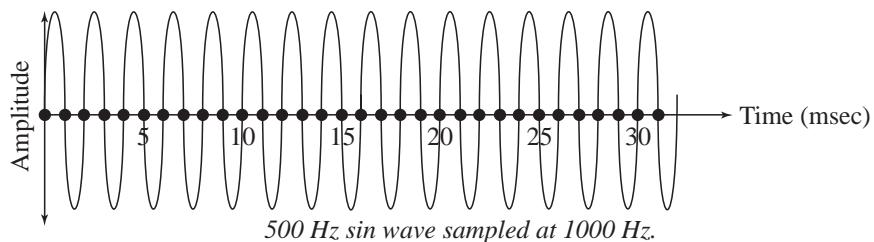
Aliasing does not occur when the sampling rate is more than twice the signal frequency.



When sampling rate is exactly twice the input frequency, the original signal may or may not be properly reconstructed. In this specific case, it is frequency shifted (aliased) to 0 Hz and lost (Figure 12.26).

**Figure 12.26**

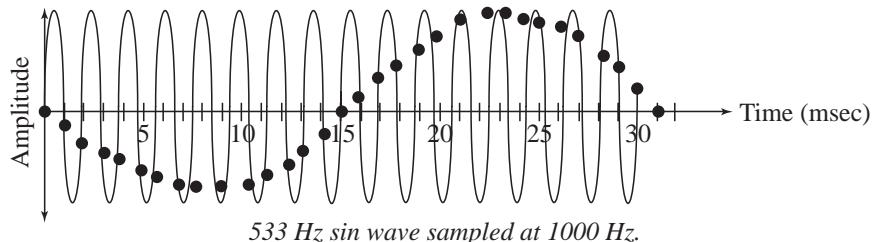
Right at the Nyquist frequency aliasing may or may not occur.



When sampling rate is slower than twice the input frequency, the original signal cannot be properly reconstructed. It is frequency shifted (aliased) to a frequency between 0 and  $0.5 f_s$ . In this case the 533 Hz wave was aliased to 33 Hz (Figure 12.27).

**Figure 12.27**

Aliasing makes the 533 Hz signal appear as a 33 Hz signal.



The choice of *sampling rate*  $f_s$  is determined by the maximum useful frequency contained in the signal. One must sample at least twice this maximum useful frequency. Faster sampling rates may be required to implement various digital filters and digital signal processing.

$$f_s > 2f_{\max}$$

Even though the largest signal frequency of interest is  $f_{\max}$ , there may be significant signal magnitudes at frequencies above  $f_{\max}$ . These signals may arise from the input  $\mathbf{x}$ , from added noise in the transducer, or from added noise in the analog processing. Once the sampling rate is chosen at  $f_s$ , then a low-pass analog filter may be required to remove frequency components above  $0.5 f_s$ . A digital filter cannot be used to remove aliasing.

### 12.3.3 How Many Bits Does One Need for the ADC?

The choice of the *ADC precision* is a compromise of various factors. The overall objective of the DAS will dictate the potential number of useful bits in the signal. If the transducer is nonlinear, then the ADC precision must be larger than the precision specified in the problem statement. For example, let  $\mathbf{y}$  be the transducer output and let  $\mathbf{x}$  be the real-world signal. Assume for now that the transducer output is connected to the ADC input. Let the range of  $\mathbf{x}$  be  $\mathbf{r}_x$ , let the range of  $\mathbf{y}$  be  $\mathbf{r}_y$ , and let the required precision of  $\mathbf{x}$  be  $\mathbf{n}_x$ . The resolutions of  $\mathbf{x}$  and  $\mathbf{y}$  are  $\Delta_x$  and  $\Delta_y$ , respectively. Let the following describe the nonlinear transducer.

$$\mathbf{y} = \mathbf{f}(\mathbf{x})$$

The required ADC precision  $\mathbf{n}_y$  (in alternatives) can be calculated by

$$\Delta_x = \frac{\mathbf{r}_x}{\mathbf{n}_x}$$

$$\Delta_y = \min \{ f(x + \Delta_x) - f(x) \text{ for all } x \text{ in } \mathbf{r}_x \}$$

$$\mathbf{n}_y = \frac{\mathbf{r}_y}{\Delta_y}$$

For example, consider the nonlinear transducer  $\mathbf{y} = \mathbf{x}^2$ . The range of  $\mathbf{x}$  is  $0 \leq x \leq 1$ . Thus, the range of  $\mathbf{y}$  is also  $0 \leq y \leq 1$ . Let the desired resolution be  $\Delta_x = 0.01$ .  $\mathbf{n}_x = \mathbf{r}_x / \Delta_x = 100$  alternatives, or about 7 bits. From the above equation,  $\Delta_y = \min\{(x + 0.01)^2 - x^2\} = \min\{0.02x + 0.0001\} = 0.0001$ . Thus,  $\mathbf{n}_y = \mathbf{r}_y / \Delta_y = 10000$  alternatives, or almost 15 bits.

**Checkpoint 12.8:** How many ADC bits are required, assuming the system is linear, if the range is 0 to 10 cm and the desired resolution is 0.01cm?

### 12.3.4 Specifications for the Analog Signal Processing

If the analog signal processing is linear, then

$$\mathbf{z} = \mathbf{G}\mathbf{y} + \mathbf{b}$$

where  $\mathbf{G}$  is the gain and  $\mathbf{b}$  is the offset. The resolution and range of  $\mathbf{z}$  can be found from the previous section.

$$\begin{aligned} \Delta_z &= \mathbf{G}\Delta_y \\ \mathbf{r}_z &= \mathbf{G}\mathbf{r}_y \end{aligned}$$

Thus, the precision at  $\mathbf{z}$  equals the precision at  $\mathbf{y}$ .

$$\mathbf{n}_z = \frac{\mathbf{r}_z}{\Delta_z} = \frac{\mathbf{G}\mathbf{r}_y}{\mathbf{G}\Delta_y} = \frac{\mathbf{r}_y}{\Delta_y} = \mathbf{n}_y$$

If the transducer and analog signal processing are both linear, then  $\mathbf{n}_z = \mathbf{n}_y = \mathbf{n}_x$ . Another factor to consider in the choice of ADC word size is the electric noise in the signal. For example, if the signal ranges from 0 to +8 V and there is 1 mV of noise in the signal, then any ADC bits beyond 13 would be wasteful. Other factors include cost and convenience for digital processing.

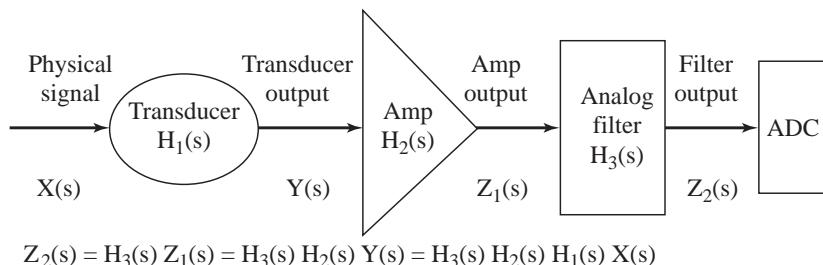
An *analog low-pass filter* may be required to remove aliasing. The cutoff of this analog filter should be less than  $0.5 f_s$ . Some transducers automatically remove these unwanted frequency components. For example, a thermistor is inherently a low-pass device. Other types

of filters (analog and digital) may be used to solve the DAS objective. One useful filter is a 60 Hz band-reject filter.

Let  $\mathbf{X}(s)$  be the Fourier transform of the physical signal, let  $\mathbf{Y}(s)$  be the Fourier transform of the transducer output, let  $\mathbf{Z}_2(s)$  be the Fourier transform of the ADC input, let  $\mathbf{H}_1(s)$  be the Fourier transfer function for the transducer, let  $\mathbf{H}_2(s)$  be the Fourier transfer function for the amplifier, let  $\mathbf{H}_3(s)$  be the Fourier transfer function for the analog filter, and let  $f_{\min} \leq f \leq f_{\max}$  be the frequency range in the signal to be processed. The analog system (transducer, amplifier, filter) must pass frequencies  $f_{\min}$  to  $f_{\max}$ . To avoid aliasing, the analog system must reject frequencies above  $0.5 f_s$  (Figure 12.28).

**Figure 12.28**

A DAS shown in the frequency domain.

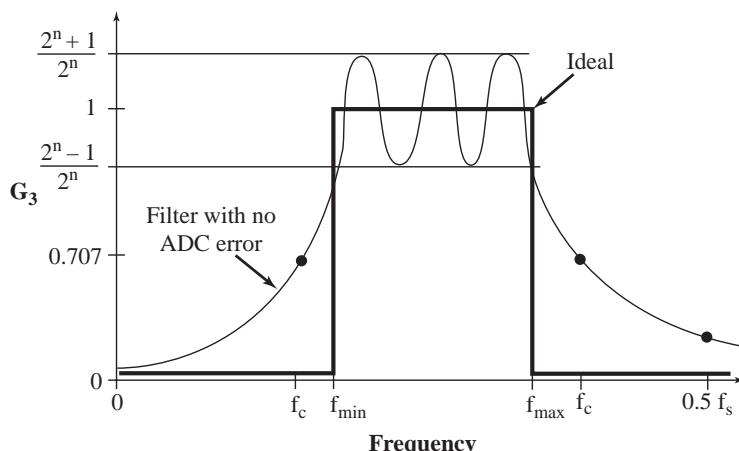


Let the gain of the analog filter be  $G_3 = |H_3(s)|$ . Then the system should pass, with no error, as seen by the ADC, for signal frequencies between  $f_{\min}$  and  $f_{\max}$ , as described in Figure 12.29. For example,

$$\frac{2^{n+1} - 1}{2^{n+1}} \leq G_3 \leq \frac{2^{n+1} + 1}{2^{n+1}} \dots \text{for } f_{\min} \leq f \leq f_{\max}$$

**Figure 12.29**

Ideal and practical filter responses.



For example, consider a system that uses an 8-bit ADC. If the gain is lower than  $255/256$  or larger than  $257/256$ , then an error will result in the ADC sample. Conversely, if the gain is between  $255/256$  and  $257/256$ , then no error should occur in the ADC sample. We will add a safety factor of  $1/2$  and require that

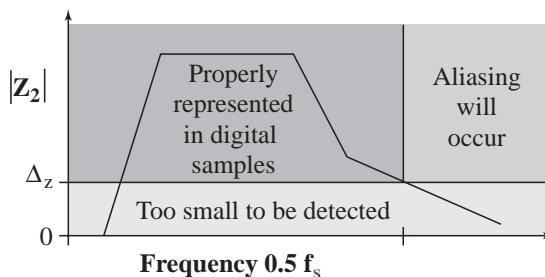
$$\frac{511}{512} \leq G_3 \leq \frac{513}{512} \quad \text{for } f_{\min} \leq f \leq f_{\max}$$

**Observation:** Most DASs do not need to pass "without ADC error" all frequencies from  $f_{\min}$  to  $f_{\max}$ . In this situation we simply place the filter cutoff frequencies at  $f_{\min}$  and  $f_{\max}$ .

To prevent aliasing, one must know the frequency spectrum of the ADC input voltage. This information can be measured with a spectrum analyzer. Typically, a spectrum analyzer samples the analog signal at a very high rate ( $>1$  MHz), performs a discrete Fourier transform, and displays the signal magnitude versus frequency. Let  $|Z_2|$  be the magnitude of the ADC input voltage as a function of frequency. For example, a bandpass device might have a frequency spectrum like the following. There are three regions in the magnitude versus frequency graph shown in Figure 12.30. We will classify any signal with an amplitude less than the ADC resolution,  $\Delta_z$ , to be undetectable. This region is labeled “too small to be detected.” Undetectable signals cannot cause aliasing regardless of their frequency. We will classify any signal with an amplitude larger than the ADC resolution at frequencies less than  $0.5 f_s$  to be properly sampled. This region is labeled “properly represented in digital samples.” It is information in this region that is available to the software for digital processing. The last region includes signals with an amplitude above the ADC resolution at frequencies greater than or equal to  $0.5 f_s$ . Signals in this region will be aliased, which means their apparent frequencies will be shifted into the 0 to  $0.5 f_s$  range.

**Figure 12.30**

To prevent aliasing, there should be no measurable signal above  $0.5 f_s$ .



Aliasing will occur if  $|Z_2|$  is larger than the ADC resolution for any frequency larger than or equal to  $0.5 f_s$ . In order to prevent aliasing,  $|Z_2|$  must be less than the ADC resolution. Our design constraint will again include a safety factor of  $1/2$ . Thus, to prevent aliasing we will make

$$|Z_2| < 0.5 \Delta_z \quad \text{for all frequencies larger than or equal to } 0.5 f_s$$

This condition usually can be satisfied by increasing the sampling rate or increasing the number of poles in the analog low-pass filter. We cannot remove aliasing with a digital low-pass filter, because once the high-frequency signals are shifted into the 0 to  $0.5 f_s$  range, we will be unable to separate the aliased signals from the regular ones.

There are errors caused by *impedance loading* between stages of the system. In general, one tries to maximize the input impedance and minimize the output impedance of the modules. A good rule is to design your system such that the error due to impedance loading causes no error in the ADC sample. The errors in the first stages must be multiplied by the amplifier gain so that they can be compared to the ADC resolution (Figure 12.31).

In this example, assume the  $V_y$  and  $V_z$  are unipolar, and the analog signal processing is linear.

$$0 \leq V_y \leq r_y \quad 0 \leq V_z \leq r_z$$

$$V_z = GV_y$$

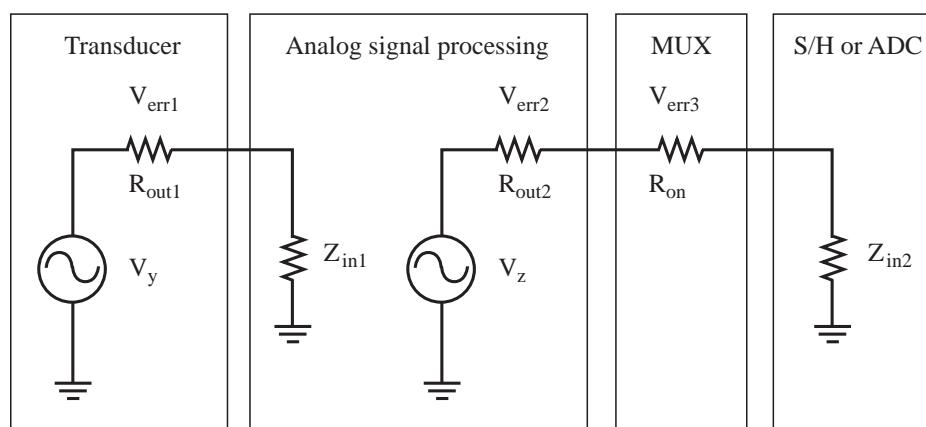
Using the simple impedance model, the voltage errors are

$$V_{err1} = V_y \frac{R_{out1}}{R_{out1} + Z_{in1}} \approx V_y \frac{R_{out1}}{Z_{in1}}$$

$$V_{err2} = V_z \frac{R_{out2}}{R_{out2} + R_{on} + Z_{in2}} \approx V_z \frac{R_{out2}}{Z_{in2}}$$

**Figure 12.31**

Block diagram for considering the problem of impedance loading.



$$V_{\text{err}3} = V_z \frac{R_{\text{on}}}{R_{\text{out}2} + R_{\text{on}} + Z_{\text{in}2}} \approx V_z \frac{R_{\text{on}}}{Z_{\text{in}2}}$$

The voltage error at the input increases by the gain,  $G$ , so the total error referred to the output is

$$G V_{\text{err}1} + V_{\text{err}2} + V_{\text{err}3} \approx G V_y \frac{R_{\text{out}1}}{Z_{\text{in}1}} + V_z \frac{R_{\text{out}2}}{Z_{\text{in}2}} + V_z \frac{R_{\text{on}}}{Z_{\text{in}2}} = V_z \left( \frac{R_{\text{out}1}}{Z_{\text{in}1}} + \frac{R_{\text{out}2} + R_{\text{on}}}{Z_{\text{in}2}} \right)$$

One wishes to keep the error below the ADC resolution,  $\Delta_z$ . The factor 1/2 is again arbitrarily chosen. The largest error occurs when  $V_z$  equals  $r_z$ :

$$r_z \left( \frac{R_{\text{out}1}}{Z_{\text{in}1}} + \frac{R_{\text{out}2} + R_{\text{on}}}{Z_{\text{in}2}} \right) \approx 0.5 \Delta_z$$

or

$$\left( \frac{R_{\text{out}1}}{Z_{\text{in}1}} + \frac{R_{\text{out}2} + R_{\text{on}}}{Z_{\text{in}2}} \right) \approx \frac{0.5}{n_z}$$

where the units of  $n_z$  are alternatives. In general, the ratio of the input impedance of each stage divided by the output impedance of the previous stage must exceed the precision of the ADC.

The voltage drop across the multiplexer can be a source of impedance-loading error (Figure 12.32). Consider the issue of impedance loading on the following two circuits. The input impedance of the ADC is 10 kΩ. The ADC is configured for  $-10$  to  $+10$  V. The max “On Resistance, at  $-25^\circ\text{C} \leq T_A \leq 85^\circ\text{C}$ ” of the PMI MUX08EP is 400 Ω. The output impedance of an inverting amplifier is the open-circuit voltage divided by the short-circuit current.

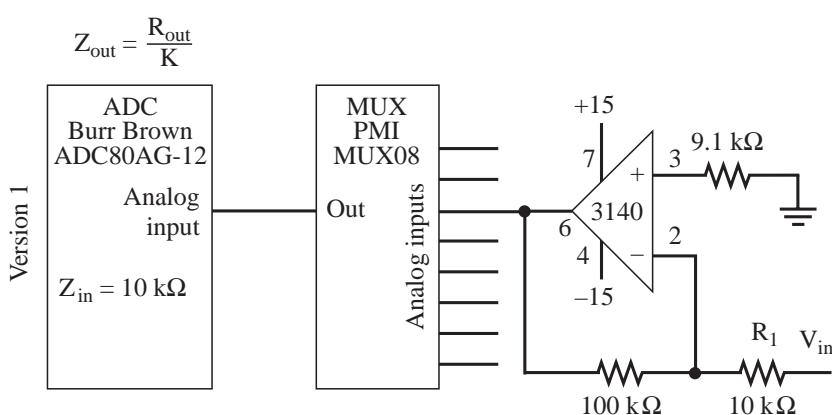
$$Z_{\text{out}} \equiv \frac{V_{\text{open}}}{I_{\text{short}}} = \frac{R_{\text{out}}(R_1 + R_2)}{K R_1}$$

where  $R_1$  and  $R_2$  are as shown below.  $K$ , the op amp open-loop gain, is 100,000.  $R_{\text{out}}$ , the op amp output impedance, is 60 Ω. The output impedance of a unity gain buffer is

$$Z_{\text{out}} \equiv \frac{R_{\text{out}}}{K}$$

Version 1 (incorrect, see Figure 12.32):  $V_{\text{max}} = +10$  V

**Figure 12.32**  
A DAS with an impedance-loading program.



$$\begin{aligned} Z_{\text{out}} (\text{of } 3140 \text{ inverter}) &= \frac{R_{\text{out}}(R_1 + R_2)}{K R_1} \\ &= \frac{60(10 \text{ k}\Omega + 100 \text{ k}\Omega)}{100000 \times 10 \text{ k}\Omega} = 0.0066 \Omega \end{aligned}$$

$$R_{\text{on}} (\text{of MUX08}) = 400 \Omega$$

$$Z_{\text{in}} (\text{of ADC}) = 10 \text{ k}\Omega$$

$$\begin{aligned} V_{\text{err}} &= V_{\text{max}} \frac{R_{\text{on}} + Z_{\text{out}}}{Z_{\text{in}} + R_{\text{on}} + Z_{\text{out}}} \\ &= 10 \text{ V} \frac{400 \Omega + 0.0066 \Omega}{10 \text{ k}\Omega + 500 \Omega + 0.0066 \Omega} \\ &\approx 10 \text{ V} \frac{400 \Omega}{10400 \Omega} \end{aligned}$$

$$= 0.4 \text{ V} \quad (80 \text{ times the ADC resolution of } 20/4096 = 0.005 \text{ V})$$

A unity gain amplifier is added between the MUX and the ADC (Figure 12.33). The purpose of this unity gain buffer amplifier in the second design is to prevent a significant voltage drop from occurring across the multiplexer. Ohm's law can be used to calculate the voltage drop across the MUX08 for both circuits.

Version 2 (correct, see Figure 12.33):  
Stage 1:

$$Z_{\text{out}} (\text{of } 3140 \text{ inverter}) = \frac{R_{\text{out}}(R_1 + R_2)}{K R_1} = \frac{60(10 \text{ k}\Omega + 100 \text{ k}\Omega)}{100000 \times 10 \text{ k}\Omega} = 0.0066 \Omega$$

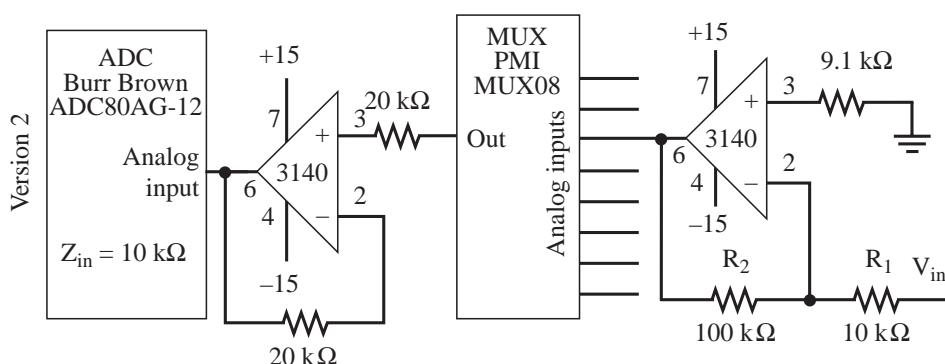
$$R_{\text{on}} (\text{of MUX08}) = 400 \Omega$$

$$Z_{\text{in}} (\text{of } 3140 \text{ buffer}) = 10^{12} \Omega$$

$$\begin{aligned} V_{\text{err}} &= V_{\text{max}} \frac{R_{\text{on}} + Z_{\text{out}}}{Z_{\text{in}} + R_{\text{on}} + Z_{\text{out}}} \\ &= 10 \text{ V} \frac{400 \Omega + 0.0066 \Omega}{10^{12} \Omega + 400 \Omega + 0.0066 \Omega} \cdot 10 \text{ V} \frac{400 \Omega}{10^{12} \Omega} \\ &= 4 \text{ nV} \quad (\text{This is well below the ADC resolution} = 20/4096 = 5 \text{ mV}) \end{aligned}$$

**Figure 12.33**

The impedance-loading program is solved with a voltage follower before the ADC.



Stage 2:

$$Z_{\text{out}} \text{ (of 3140 buffer)} = \frac{R_{\text{out}}}{K} = \frac{60}{100000} = 0.0006 \Omega$$

$$Z_{\text{in}} \text{ (of ADC)} = 10 \text{ k}\Omega$$

$$\begin{aligned} V_{\text{err}} &= V_{\text{max}} \frac{Z_{\text{out}}}{Z_{\text{in}} + Z_{\text{out}}} = 10 \text{ V} \frac{0.0006 \Omega}{10 \text{ k}\Omega + 0.0006 \Omega} \approx 10 \text{ V} \frac{0.0006 \Omega}{10 \text{ k}\Omega} \\ &= 0.6 \mu\text{V} \quad (\text{This is well below the ADC resolution} = 20/4096 = 5 \text{ mV}) \end{aligned}$$

### 12.3.5

#### How Fast Must the ADC Be?

The *ADC conversion time* must be smaller than the quotient of the sampling interval by the number of multiplexer signals. Let  $m$  be the number of multiplexer signals that must be sampled at a rate  $f_s$ , let  $t_{\text{mux}}$  be the settling time of the multiplexer, and let  $t_c$  be the ADC conversion time. Then, without a S/H,

$$m \cdot (t_{\text{mux}} + t_c) < 1/f_s$$

With a S/H, one must include both the acquisition time,  $t_{\text{aq}}$  and the aperture time,  $t_{\text{ap}}$ :

$$m \cdot (t_{\text{mux}} + t_{\text{aq}} + t_{\text{ap}} + t_c) < 1/f_s$$

### 12.3.6

#### Specifications for the S/H

A S/H is required if the analog input changes more than one resolution during the conversion time. Let  $dz/dt$  be the maximum slope of the ADC input voltage, let  $\Delta_z$  be the ADC resolution, and let  $t_c$  be the ADC conversion time. A S/H is required if

$$\frac{dz}{dt} \cdot t_c > 0.5 \Delta_z$$

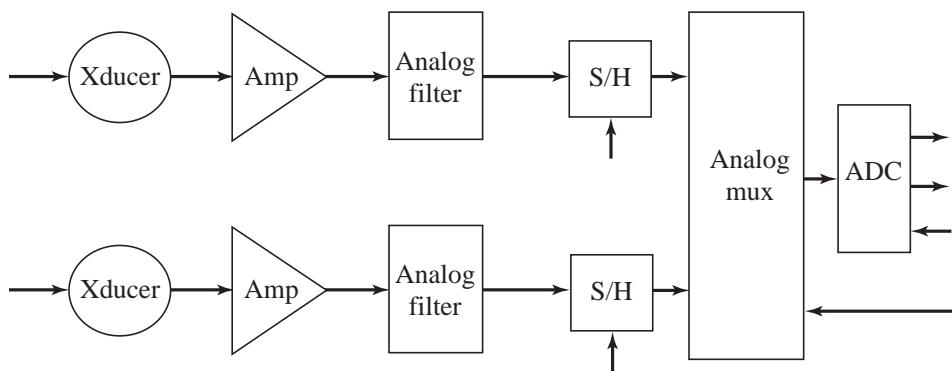
If the transducer and analog signal processing are both linear, the determination of whether or not to use a S/H can be calculated from the input signals. A S/H is required if

$$\frac{dx}{dt} \cdot t_c > 0.5 \Delta_x$$

Two S/Hs can be used to analyze the timing between two signals. For example, the circuit in Figure 12.34 can be used to measure the phase between two signals with one ADC. The

**Figure 12.34**

Multiple S/Hs can be used to implement synchronized sampling.



two S/Hs are given the hold command simultaneously; then the signals are sequentially converted. The digital samples represent the two signals at the same time.

## 12.4 Analysis of Noise

The consideration of noise is critical for all instrumentation systems. The success of an instrument does depend on careful transducer design, precision analog electronics, and clever software algorithms. But any system will fail if the signal is overwhelmed by noise. Fundamental noise is defined as an inherent and nonremovable error. It exists because of fundamental physical or statistical uncertainties. We will consider three types of fundamental noise:

- Thermal noise (also called white noise or Johnson noise)
- Shot noise
- 1/f noise

Although fundamental noise cannot be eliminated, there are ways to reduce its effect on the measurement objective. In general, added noise includes the many disturbing external factors that interfere with or are added to the signal. We will consider three types of added noise:

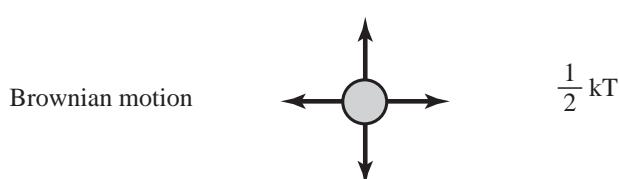
- Galvanic noise
- Motion artifact
- Electromagnetic field induction

### 12.4.1 Thermal Noise

Thermal fluctuations occur in all materials at temperatures above absolute zero. Brownian motion, the random vibration of particles, is a function of absolute temperature (Figure 12.35). As the particles vibrate, there is an uncertainty as to the position and velocity of the particles. This uncertainty is related to the thermal energy:

**Figure 12.35**

Brownian motion of individual particles.



Absolute temperature T (K)  
 Boltzmann's constant  $k = 1.67 \times 10^{-23}$  J/K  
 Uncertainty in thermal energy  $\approx (1/2) kT$

Because the electric power of a resistor is dissipated as thermal power, the uncertainty in thermal energy produces an uncertainty in electric energy. The electric energy of a resistor depends on

Resistance R ( $\Omega$ )  
 Voltage V (V)  
 Time (s)  
 $\text{Electric power} = V^2/R$  (W)  
 $\text{Electric energy} = V^2 \cdot \text{time}/R$  (W · s)

By equating these two energies we can derive an equation for voltage noise similar to the empirical findings of J. B. Johnson. In 1928, he found that the open-circuit RMS voltage noise of a resistor was given by

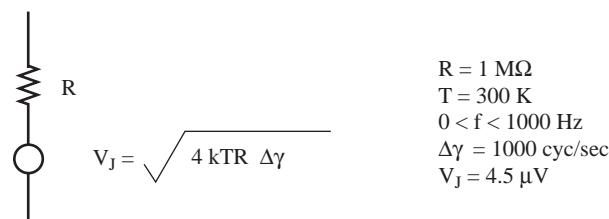
$$V_J^2 = 4kTR \Delta\gamma$$

$$\Delta\gamma = f_{\max} - f_{\min}$$

where  $f_{\max} - f_{\min}$  is the frequency interval, or bandwidth, over which the measurement was taken. For instance, if the system bandwidth is DC to 1000 Hz, then  $\Delta\gamma$  is 1000 cycles/s. Similarly, if the system is a bandpass from 10 to 11 kHz, then  $\Delta\gamma$  is also 1000 cycles/s. The term *white noise* comes from the fact that thermal noise contains the superposition of all frequencies and is independent of frequency. It is analogous to optics, where *white light* is the superposition of all wavelengths (Figure 12.36). Table 12.8 illustrates that white noise increases with resistance value and with system bandwidth.

**Figure 12.36**

White noise exists in all resistors.



**Table 12.8**

White noise for resistors at 300K = 27°C.

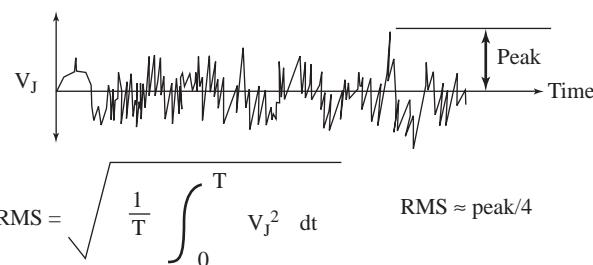
	1 Hz	10 Hz	100 Hz	1 kHz	10 kHz	100 kHz	1 MHz
10 kΩ	14 nV	45 nV	142 nV	448 nV	1.4 μV	4.5 μV	14 μV
100 kΩ	45 nV	142 nV	448 nV	1.4 μV	4.5 μV	14 μV	45 μV
1 MΩ	142 nV	448 nV	1.4 μV	4.5 μV	14 μV	45 μV	142 μV

Interestingly, only resistive, not capacitive or inductive, electric devices exhibit thermal noise. Thus a transducer that dissipates electric energy will have thermal noise, and a transducer that simply stores electric energy will not.

Figure 12.37 defines RMS as the square root of the time average of the voltage squared. RMS noise is proportional to noise power. The crest factor is the ratio of peak value divided by RMS. The peak value is half the peak-to-peak amplitude and can be measured easily from recorded data. From Table 12.9, we see that the crest factor is about 4. The crest factor can be defined for other types of noise.

**Figure 12.37**

RMS is a time average of the voltage squared.

**Table 12.9**

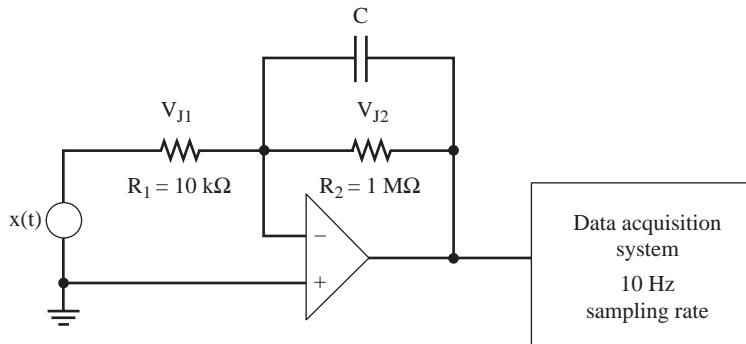
Crest factor for thermal noise.

Percent of the time the peak is exceeded	Crest factor (peak/RMS)
1.0	2.6
0.1	3.3
0.01	3.9
0.001	4.4
0.0001	4.9

**Example analysis.** The objective of the following DAS is to sample  $x(t)$  at 10 Hz and perform a software calculation based on the DC to 5-Hz components of  $x$ . The gain of this amplifier is  $-100$ . Assume the bandwidth of the amplifier without the capacitor  $C$  is 100 kHz. With  $C = 0.01 \mu\text{F}$ , the bandwidth of the amplifier is reduced to 100 Hz. This reduction does not affect the DC to 5 Hz components of  $x$  (Figure 12.38).

**Figure 12.38**

A DAS with a gain of 100 and a frequency range of 100 Hz.



Let  $V_{J1}$  be the thermal noise of  $R_1$  and  $V_{J2}$  be the thermal noise of  $R_2$ . Since the noise is related to the thermal power, the voltage amplitudes are combined at the power level. That is, if two devices generate voltage noises of  $V_1$ , and  $V_2$  respectively, then together they will generate

$$V_{\text{total}} = \sqrt{V_1^2 + V_2^2}$$

When considering the overall effect of noise,  $V_{J1}$  is amplified by the op amp, whereas  $V_{J2}$  is not. The presence of the capacitor reduces the noise by 32 without degrading the system performance (Table 12.10). In addition to white noise in the resistor, there is white noise in the op amp.

**Table 12.10**

Reducing the system bandwidth reduces the thermal noise.

	Without C	With C
$V_{J1}$	$4.5 \mu\text{V}$	$142 \text{nV}$
$V_{J2}$	$45 \mu\text{V}$	$1.4 \mu\text{V}$
$\sqrt{[(100V_{J1})^2 + (V_{J2})^2]}$	$452 \mu\text{V}$	$14.3 \mu\text{V}$

**12.4.2 Shot Noise** Shot noise arises from the statistical uncertainty of counting discrete events. Thermal cameras, radioactive detectors, photomultipliers, and O<sub>2</sub> electrodes count individual photons, gamma rays, electrons, and O<sub>2</sub> particles, respectively, as they impinge upon the transducer. Let dn/dt be the count rate of the transducer, and let Δt be the measurement interval or count time. The average count is

$$n = \frac{dn}{dt} \Delta t$$

On the other hand, the statistical uncertainty of counting random events is  $\sqrt{n}$ . Thus the shot noise is

$$\text{Shot noise} = \sqrt{\frac{dn}{dt} \Delta t}$$

The S/N ratio is

$$S/N = \frac{n}{\sqrt{n}} = \sqrt{\frac{dn}{dt} \Delta t}$$

The solutions are to maximize the count rate (by moving closer or increasing the source) and to increase the count time. There is a clear trade-off between accuracy and measurement time.

**12.4.3 1/f, or Pink, Noise** Pink noise is somewhat mysterious. The origin of 1/f lacks rigorous theory. It is present in devices that have connections between conductors. 1/f noise results from a fluctuating conductivity. It is of particular interest to low-bandwidth applications because of the 1/f behavior. Wire-wound resistors do not have 1/f noise, but semiconductors and carbon resistors do. One of the confusing aspects of 1/f noise is its behavior as the frequency approaches 0 Hz. The noise at DC is not infinite because although 1/f is infinite at DC, Δγ is zero. Garrett gives an equation to calculate the 1/f noise of a carbon resistor (Table 12.11):

$$V_c = (10^{-6})\sqrt{1/f} R I \sqrt{\Delta\gamma}$$

where  $V_c$  = 1/f voltage noise, V  
 $f$  = frequency, Hz  
 $R$  = resistance, Ω  
 $I$  = average direct current, A  
 $\Delta\gamma$  = system bandwidth, Hz

**Table 12.11**  
 $V_c$  versus frequency for  
 $R = 10 \text{ k}\Omega$ ,  $I = 1 \text{ mA}$ ,  
 $\Delta\gamma = 1 \text{ kHz}$ .

$f(\text{Hz})$	$V_c (\mu\text{V})$
1	316
10	100
100	32
1000	10

**12.4.4 Galvanic Noise** The contact between dissimilar metals will induce a voltage, due to the electrochemistry at the metal-metal interface. Voltages will also develop when a conductive liquid contacts a metal. This problem usually arises as a metal surface within a connector oxidizes (corrosion due to moisture). The materials least susceptible to corrosion are silver, graphite, gold, and platinum.

**12.4.5 Motion Artifact** Motion can introduce errors in many ways. According to Faraday's law, a conducting wire that moves in a magnetic field will induce an EMF. This voltage error is proportional to the strength of the magnetic field, the length of the wire that is moving, the velocity of the motion, and the angle between the velocity and the field. Another problem occurring with moving cables is that

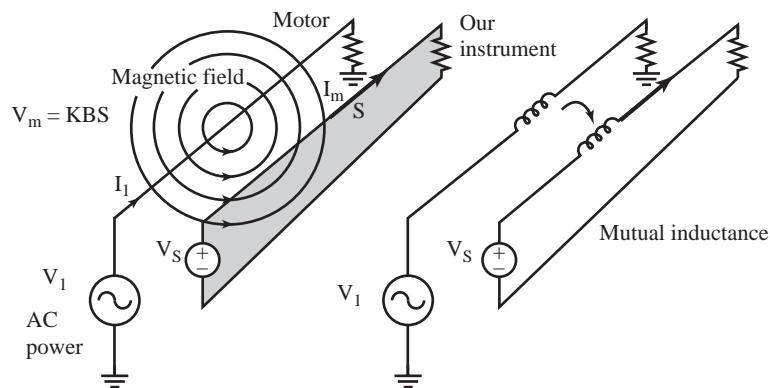
the connector impedance may change or disconnect. Acceleration of the transducer will induce forces inside the device, often affecting its response.

### 12.4.6 Electromagnetic Field Induction

Usually, the largest source of noise is electromagnetic field induction. According to Faraday's law, changing magnetic fields can induce a voltage into our circuits. The changing magnetic field must pass through a wire loop, drawn as the shaded area in Figure 12.39. This voltage noise ( $V_m$ ) is proportional to the strength of the magnetic field,  $B$  (wb/m<sup>2</sup>), the area of the loop  $S(m^2)$ , and a geometric factor,  $K$  (volts/wb.) The drawing on the left of Figure 12.39 illustrates the physical situation causing magnetic field pickup. A typical situation causing magnetic field noise occurs when AC power is being delivered to a low-impedance load, such as a motor. The voltage  $V_1$  is the 120 VAC 60 Hz power line, and  $V_s$  is a signal in our instrument. The alternating current ( $I_1$ ) will create a magnetic field,  $B$ . This magnetic field ( $B$ ) also alternates as it flows through the loop area ( $S$ ) formed by the wires in our circuit (such as the lead wires between the transducer and the instrument box). This will induce a current ( $I_m$ ) along the wire, causing a voltage error ( $V_m$ ). We can test for the presence of magnetic field pickup by deliberately changing the loop area and observing the magnitude of the noise as a function of the loop area. The drawing on the right of Figure 12.39 illustrates an equivalent circuit we can use to model magnetic field pickup. Basically, we can model magnetic field-induced noise as a mutual inductance between an AC current flow and our electronics.

**Figure 12.39**

Magnetic field noise pickup can be modeled as a transformer.



The second way EM fields can couple into our circuits is via the electric field. Changing electric fields will capacitively couple into the lead wires. The drawing on the left of Figure 12.40 illustrates the physical situation causing electric field pickup. The alternating voltage ( $V_1$ ) will create an electric field. This electric field also traverses near the wires in our circuit (such as the lead wires between the transducer and the instrument box). This will induce a displacement current ( $I_d$ ) along the wire. We can test for the presence of electric field pickup by placing a shield separating our electronics from the source of the field and observing the magnitude of the noise. The drawing on the right of Figure 12.40 illustrates an equivalent circuit we can use to model electric field pickup. Basically, we can model electric field-induced noise as a stray capacitance between an AC voltage and our electronics.

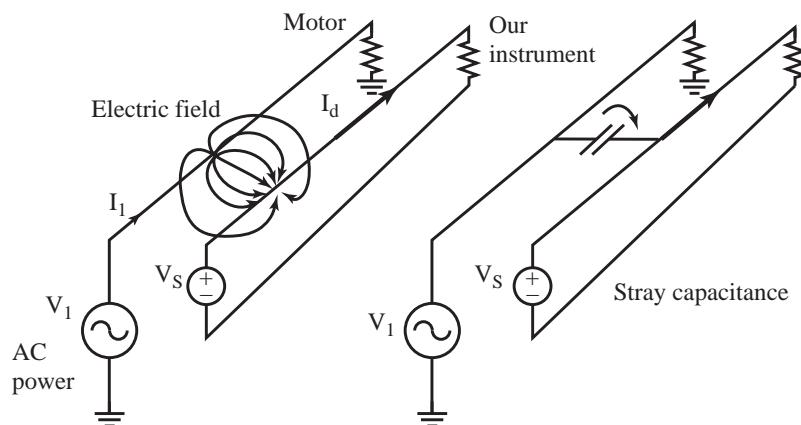
**Observation:** Sometimes EM fields originate from inside the instrument box, such as high-frequency digital clocks and switching power supplies.

### 12.4.7 Techniques to Measure Noise

There are two objectives when measuring noise. The first objective is to classify the type of noise. In particular, we wish to know whether the noise is broadband (i.e., all frequencies, such as white noise) or does the noise contain specific frequencies (e.g., 60, 120, 180 Hz, . . . , such as a 60 Hz EM field pickup). The type of noise is of great importance when determining where

**Figure 12.40**

Electric field noise pickup can be modeled as a stray capacitance.



the noise is coming from. Classifying the noise type is essential in developing a strategy for reducing the effect of the noise. The second objective is to quantify the magnitude of the noise. Quantifying the noise is helpful in determining whether a change to the system design has increased or decreased the noise. The measurement resolution of many data acquisition systems is limited by noise rather than by ADC precision and software algorithms. For these systems, quantitative noise measurements are an important performance parameter of the instrument.

#### Digital Voltmeter (DVM) in AC Mode.

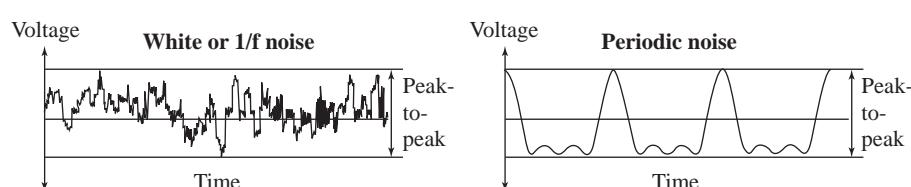
Root-mean-squared (RMS) is defined as the square root of the time-average of the voltage squared (Figure 12.37). If you remove the input signal, the output of the system contains just noise. Because the resistance load is usually constant, squaring the voltage results in a signal proportional to noise power. The averaging calculation gives a measure related to average power, and the square root produces a result with units in volts. RMS noise of a signal can be measured with a DVM using AC mode. Most DVMs in AC mode perform a direct measurement of RMS; hence this method is the most precise. For example, a  $3\frac{1}{2}$  digit DVM has a precision of about 11 bits. A calibrated voltmeter in AC mode will provide the most accurate quantitative method to measure noise.

#### Analog Oscilloscope.

The second method is to connect the signal to an oscilloscope and measure the peak-to-peak noise amplitude, as illustrated in Figure 12.41. The crest factor is the ratio of peak value divided by RMS. The peak value is one-half of the peak-to-peak amplitude, and can be estimated from the scope tracing. From Table 12.9, we see for white noise that the crest factor is about 4, so we can approximate the RMS noise amplitude by dividing the peak-to-peak noise by 8. Because the quantitative assessment of noise with a scope requires visual observation, this method can be used only to approximate the quantitative level of noise. On the other hand, oscilloscopes have very high bandwidth, and therefore they are good for classifying high-frequency noise. White noise and 1/f noise look random, like the left graph in Figure 12.41. For white and 1/f noise, the scope trigger will not be able to capture a repeating waveform.

**Figure 12.41**

Quantifying noise by measuring peak-to-peak amplitude.



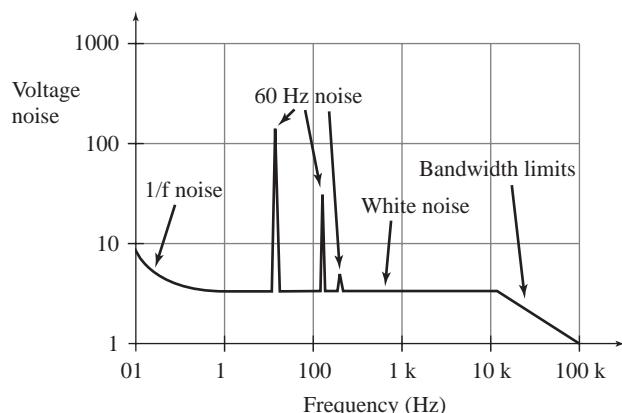
Noise from EM fields on the other hand is repeating and can be triggered by the scope. In fact the **line-trigger** setting on the scope can be used specifically to see whether the noise is correlated to the 60 Hz 120VAC power line. In particular, 60 Hz noise will trigger when the line-trigger setting of the scope is used. The shape of the noise varies depending on the relative strengths of the fundamental and harmonic frequencies. The graph on the right is a periodic wave with a fundamental plus a 50% strength first harmonic.

### Spectrum Analyzer.

The third method uses a spectrum analyzer, which combines a computer, high-speed ADC sampling, and the fast Fourier transform (FFT), as illustrated in Figure 12.42. Being able to see the noise in the frequency domain is a particularly useful way to classify the type of noise. Both 1/f and white noise components exist in a typical analog amplifier, but the amplitude is usually small compared to EM field noise. The 1/f and white noise levels occurring in electronics can be reduced by dropping the temperature or spending more money to buy a better device. Typically, we see the fundamental and multiple harmonics for EM field noise. For example, 60 Hz noise also includes components at 120, 180, 240 Hz, and so on.

**Figure 12.42**

Classifying noise by measuring the amplitude versus frequency with a spectrum analyzer.



### 12.4.8 Techniques to Reduce Noise

It is much simpler to reduce noise early in the design process. Conversely, it can be quite expensive to eliminate noise after an instrument has been built. Therefore, it is important to consider noise at every stage of the development cycle. We can divide noise reduction techniques into three categories. The first category involves reducing noise from the source. You can enclose noisy sources in a grounded metal box. If a cable contains a high-frequency noise signal, that signal could be filtered. Magnetic and electric field strength depends on  $dI/dt$  and  $dV/dt$ . So whenever possible, you should limit the rise and fall times of noisy signals. For example, the square wave on the left of Figure 12.43 will radiate more noise than the smooth signal on the right.

Motors have coils to create electromagnets, and noise can be reduced by limiting the  $dI/dt$  in the coil. Cables with noisy signals should be twisted together, so the radiated magnetic fields will cancel. These cables should also be shielded to reduce electric field radiation, and this shield should be grounded on both sides.

**Figure 12.43**

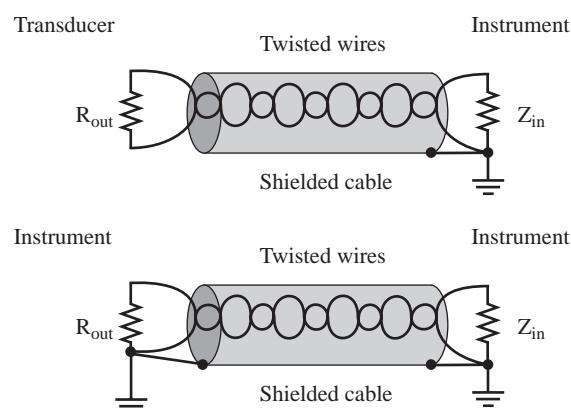
Limiting rise/fall times can reduce radiated noise.



The second category of noise reduction involves limiting the coupling between the noise source and your instrument. Whenever possible, maximize the distance between the noisy source and the delicate electronics. All transducer cables should use twisted wire, as shown in Figure 12.44. For situations in which a remote sensor is attached to an instrument, the ground shield should be connected only on the instrument side. If the cable is connected between two instruments, then connect the ground shield on both instruments. For high-frequency signals coaxial cable is required. The shield should be electrically insulated to eliminate direct electrical connection to other devices. If noisy signals must exist in the same cable as low-level signals, then separate the two with a ground wire in between. Whenever possible, reduce the length of a cable. Similarly minimize the length of leads that extend beyond the ground shield. Place the delicate electronics in a grounded case. You can use optical or transformer isolation circuits to separate the noisy ground from the ground of the delicate electronics.

**Figure 12.44**

Proper cabling can reduce noise when connecting a remote transducer or when connecting two instruments.

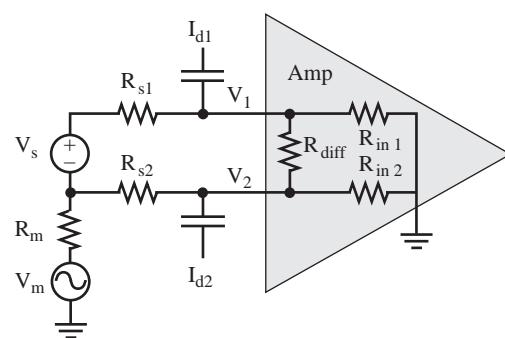


The last category involves techniques that reduce noise at the receiver. The bandwidth of the system should be as small as possible. In particular, you can use an analog low-pass filter to reduce the bandwidth, which will also reduce the noise. You can add frequency-reject digital filters to reduce specific noise frequencies such as 60, 120, or 180 Hz. You should use power supply decoupling capacitors on each chip to reduce the noise coupling from the power supply to the electronics. Figure 12.45 illustrates how EM field noise will affect our instrument. If the cable has twisted wires, then  $I_{d1} = I_{d2}$ . The input impedance of the amplifier is usually much larger than the source impedance of the signal. Thus,  $V_1 - V_2 = R_{s1} I_{d1} - R_{s2} I_{d2}$ .

Some capacitors have a foil wrap surrounding their cylindrical shape. This foil should be grounded. For more information about noise, refer to Henry Ott, *Noise Reduction Techniques in Electronic Systems*, Wiley, 1988, or Ralph Morrison, *Grounding and Shielding Techniques*, Wiley, 1998.

**Figure 12.45**

Capacitively coupled displacement currents.



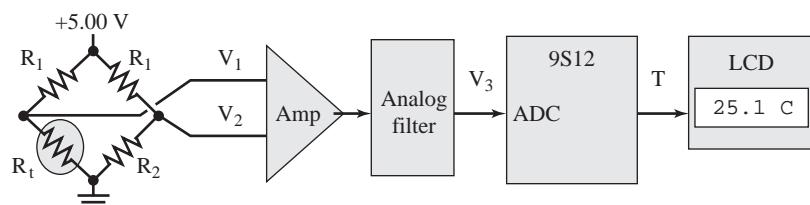
## 12.5 Data Acquisition Case Studies

We introduced the design process back in Chapter 1. In this section, we will present the designs of four data acquisition systems. The first example is a quantitative measurement of temperature, the second example is a qualitative measurement of electrocardiogram signals, and the third example compares measurements made with the ADC versus timing measurements using input capture. In the **analysis phase**, we determine the requirements and constraints for our proposed system. In the **high-level design** phase, we define our input/output, break the system in modules, and show the interconnection using data flow graphs. In the **engineering design** phase, we design the hardware/software subcomponents using techniques such as simulation, mechanical mockups, and call-graphs. We define specific I/O signals, analog circuits, power sources, noise filters, software algorithms, data structures, and testing procedures. During the **implementation** and **testing** phases, we build and test the modules. Modularity allows for concurrent development.

**Example 12.1** Design an instrument that measures temperature with a range of 25 to 50°C, and a resolution of 0.1°C. The frequency range of interest is 0 to 0.1 Hz.

**Solution** The first decision to make is the choice of transducer. The RTD has a linear resistance versus temperature response. RTDs are expensive but a good choice for ease of calibration, interchangeability, and accuracy. Thermocouples are inexpensive and a good choice for large temperature ranges, harsh measurement conditions, fast response time, interchangeability, and large temperature ranges. Interchangeability means we can buy multiple transducers and they will all have similar temperature curves. Thermistors will be used in this design because they are inexpensive and have better sensitivity than RTDs and thermocouples. A thermometer built using a thermistor will be harder to mass-produce because each transducer must be separately calibrated in order to create an accurate measurement. From a first glance, we might expect an 8-bit ADC to generate a temperature resolution of 0.1°C (range/precision = resolution). On the other hand, because the thermistor is nonlinear, we will need to verify that the resolution specification has been met. Figure 12.46 shows the block diagram of the instrument, which also illustrates the data flow in our system.

**Figure 12.46**  
Block diagram of a temperature measurement system using a thermistor.



The resistance bridge is a classic means to convert resistance to voltage. Table 12.12 is used during the design phase to show the signal values as they transverse the electronics. The +5.00 reference drives the bridge. The value of resistor  $R_1$  is chosen to eliminate errors due to self-heating of the thermistor ( $500\text{ k}\Omega$ ). Since we will be using rail-to-rail electronics, we need to have all voltages within the 0 to +5 V range. We can make the bridge output ( $V_1 - V_2$ ) positive by selecting the value of resistor  $R_2$  equal to the thermistor resistance at the maximum temperature (200  $\text{k}\Omega$ ). Next, we choose the gain of the amplifier to map the minimum temperature into the +5 V limit of the ADC (4.232). Since the thermistor is nonlinear, we will tabulate explicit values to determine the ADC precision required (Table 12.13). There are two possible approaches to the design of the amplifier. If we use an instrumentation

**Table 12.12**

Signals as they pass through the temperature data acquisition system.

T (°C)	R <sub>t</sub> (kΩ)	V <sub>1</sub> (V)	V <sub>1</sub> –V <sub>2</sub> (V)	V <sub>3</sub> (V)	ADC	T (0.1°C)
25	546.0	2.610	1.181	5.000	255	250
30	444.3	2.353	0.924	3.911	199	300
35	364.0	2.106	0.678	2.869	146	350
40	300.1	1.875	0.447	1.890	96	400
45	248.9	1.662	0.233	0.987	50	450
50	207.6	1.467	0.039	0.163	8	500

amp, the input impedance will be large enough not to affect the bridge. If we use a single op amp differential amp, then the amplifier will load the bridge and affect the bridge response. In this system, an instrumentation amp will be used, because the entire amplifier and LPF can be built with a single MAX494 device. The first two columns of Table 12.12 show the resistance temperature calibration of the 500 kΩ thermistor. The third column, V<sub>1</sub>, is the voltage across the thermistor. The fourth column is the output of the bridge, V<sub>1</sub>–V<sub>2</sub>. V<sub>3</sub>; the output of the instrumentation amp, is shown in the fifth column. The ADC value gives the digital output of an 8-bit converter, and the last column will be calculated by our software as a decimal fixed-point with resolution 0.1°C. We use this table in two ways. Initially, we use theoretical values to design the electronics and software. During the implementation phase, we substitute resistors with standard values to bring down the cost. During the testing phase, we measure actual values to verify proper operation. Measured values for the last two columns will be stored in software as a calibration table. To measure temperature, the software measures the ADC value, and then uses a table look-up and linear interpolation to get the decimal fixed-point temperature (last column). The fixed-point number is output to an LCD display.

**Observation:** There is an Excel worksheet named Therm.xls on the book Web site that was used to create this design. See also Therm10.xls, Therm12.xls, and Therm16.xls.

The fastest slew rate for the thermistor is 10°C/sec. Assuming the ADC conversion time is 25 μs, no S/H is needed, because the maximum slope of T multiplied by the ADC conversion time is less than the temperature resolution.

$$10\text{°C/sec} \cdot 25\text{ }\mu\text{s} = 0.00025\text{°C} << 0.25\text{°C}$$

In order to prevent noise in the ADC samples, the noise must be less than the resolution. The resolution of V<sub>1</sub> – V<sub>2</sub> is its range (1.181V) divided by its precision (256), which is 4.6 mV. Again a safety factor of 1/2 is included. Thus, in the frequency range 0 to 1 Hz, the maximum allowable noise referred to the input of the differential amp should be

$$\text{amplifier noise} \leq \frac{\text{resolution}}{2} = 2.3\text{ mV}$$

A two-pole low-pass analog filter is used to pass the temperature signal having frequencies from 0 to 0.1 Hz, to reject noise having frequencies above 0.1 Hz, and to prevent aliasing. We design the filter so the ADC error is less than 1/2 a resolution at 0.1 Hz (i.e., gain = 511/512). The gain of the two-pole low-pass filter is

$$G_3 = \left| \frac{1}{1 + (f/f_c)^4} \right|$$

where f<sub>c</sub> must be chosen large enough so the gain, G, is at least = (2<sup>n+1</sup> – 1)/2<sup>n+1</sup> = 511/512 for frequencies 0 to 0.1 Hz. Thus, f<sub>c</sub> must be larger than 1.1 Hz. In order to build the filter with standard components, a 2.25 Hz LPF will be designed (see Figure 12.47). The 0.2 μF capacitor can be created by placing two 0.1 μF capacitors in parallel.

$$f_c = \frac{0.1\text{ Hz}}{\sqrt{(512/511)^4 - 1}} = 1.1\text{ Hz}$$

**Observation:** In most situations, we could simply place the LPF cutoff frequency at 0.1 Hz.

**Observation:** There is an Excel worksheet named LPF.xls on the book Web site that was used to create this LPF.

In order to prevent aliasing,  $Z_2$  must be less than the ADC resolution for all frequencies larger than or equal to  $0.5 f_s$ . As an extra measure of safety, we make the amplitude less than  $0.5 \Delta_z$  for frequencies above  $0.5 f_s$ . Thus,

$$|Z_2| < 0.5\Delta_z = \frac{5 \text{ V}}{512} \approx 0.01 \text{ V}$$

For discussion, assume the following criteria for frequencies above 1 Hz, which are dominated by the thermistor response and the LPF.

$$|X| = \frac{0.5^\circ\text{C}}{\sqrt{1 + (f/1 \text{ Hz})^2}} \quad \text{LPF} = \frac{1}{\sqrt{1 + (f/2 \text{ Hz})^4}}$$

The approximate gain of the entire circuit is 5 V/25°C. Thus

$$|Z_2| = \frac{0.1 \text{ V}}{\sqrt{[1 + (f/1 \text{ Hz})^2][1 + (f/2 \text{ Hz})^4]}}$$

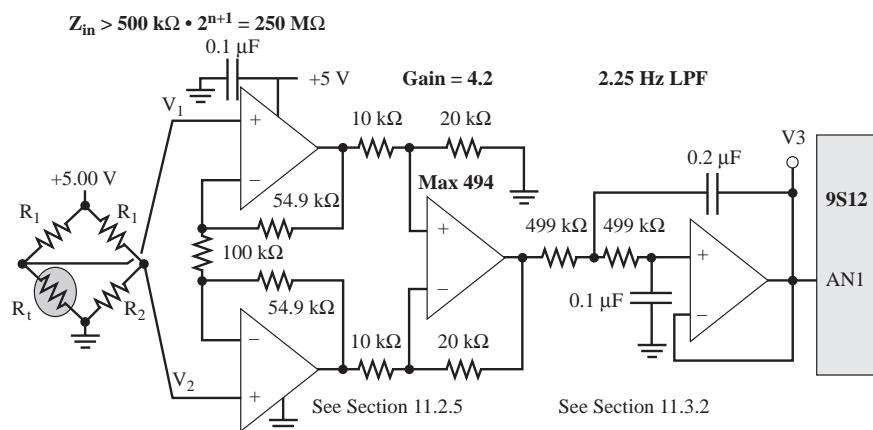
We choose the sampling rate,  $f_s$ , to prevent aliasing, we need a sampling rate above 6.6 Hz.

$$\frac{0.1 \text{ V}}{\sqrt{[1 + (f_s/2)^2][1 + (f_s/4)^4]}} < 0.01 \text{ V}$$

The effective output impedance of the bridge is 500 kΩ. The input impedance of the differential amp must be high enough not to affect the ADC conversion. Although the design equations specified a gain of 4.232, the circuit in Figure 12.47 implements a slightly smaller gain (4.2) in order to use standard resistor values.

**Figure 12.47**

Amplifier and low-pass filter.  $R_1 = 500 \text{ k}\Omega$  and  $R_2 = 200 \text{ k}\Omega$ .



To determine the resolution we work backwards, as illustrated in Table 12.13. The basic approach to verifying the temperature resolution is to work backwards through the circuit,

**Table 12.13**

Equations calculated in reverse to show that the resolution meets the design specification.

ADC	$V_3$ (V)	$V_1 - V_2$ (V)	$V_1$ (V)	$R_T$ (kΩ)	T (°C)	$\Delta T$ (°C)
255	5.000	1.181	2.610	546.0	25.000	
254	4.980	1.177	2.605	544.0	25.089	0.089
101	1.980	0.468	1.896	305.5	39.524	
100	1.961	0.463	1.892	304.3	39.628	0.103
1	0.020	0.005	1.433	200.9	50.925	
0	0.000	0.000	1.429	200.0	51.053	0.128

showing that a change in ADC value of 1 corresponds to a temperature change of about 0.1°C.

**Checkpoint 12.9:** What would be the temperature resolution if the ADC precision were increased from 8 to 10 bits?

There are three possible approaches to converting ADC sample to temperature (the last two columns of Table 12.12). First, we could fit the transfer function to a polynomial equation and save the coefficients of that equation as the calibration file. This approach performs well for simple situations. Second, we could calculate the temperature output for each possible ADC and save it in a 256-entry lookup table. This conversion is fast because we just need to use the ADC data to index into the big table. This method is fast, but requires a lot of memory. The third approach is shown in Programs 12.1 and 12.2, which use a small table of paired (ADC,T) data. These points are determined from experimental calibration. To find the corresponding temperature for a given ADC value, the program first searches the table for a pair of ADC values that surround the input. Then, it uses linear interpolation to calculate the temperature, given the four entries in the table and the ADC input. When speed is important, we can consider writing time-critical functions in assembly. Linear interpolation is one of three or four operations on the 9S12 for which an assembly solution is significantly faster than a C program (the others are Fuzzy Logic, mixed 16/32-bit math, and the  $\text{SUM} = \text{SUM} + X * Y$  calculation). Program 12.1 is written in Metrowerks assembly syntax. This subroutine is saved as a `LookUp.asm` file and included in the Metrowerks project of the main data acquisition system. The output result is a fixed-point number with a resolution of 0.1°C.

### Program 12.1

Assembly language program to convert 8-bit ADC into fixed-point temperature.

```

        xref ADCtable ; monotonic list of ADC values
        xref Ttable    ; list of corresponding temperature values
        absentry LookUp
;*****LookUp*****
;Inputs: RegD is 0 to 65534 Xdata point, xL
;      RegD input must be greater than or equal to first Xdata point
;      RegD input must be less than last Xdata point
;Output: RegD is 0 to 65535 Ydata point, decimal fixed-point 0.1C
;Registers destroyed: X,Y,B,CCR
LookUp: ldx #ADCtable ; first find x1<=xL<x2
        ldy #Ttable
search  cpd 2,x      ; check xL<x2
        blo found       ; stops when X points to x1
        leax 2,x
        leay 2,y
        bra search
found   subd 0,x      ; xL-x1
        pshd
        ldd 2,x        ; x2
        subd 0,x      ; D=x2-x1
        tfr D,X        ; X=x2-x1
        puld          ; D=(xL-x1)
        fdiv           ; X=(65536*(xL-x1))/(x2-x1)
        tfr X,D
        tfr A,B        ; B=(256*(xL-x1))/(x2-x1)
        etbl 0,y        ; Y=>>y1,y2
        rts             ; D=Y1+B*(Y2-Y1)

```

Program 12.2, written in C, performs the real-time data acquisition. The calibration data in `ADCtable` and `Ttable` are stored in EEPROM. Extra entries were added at the beginning and end of the table to guarantee that the search step of `LookUp` function will always be successful. In general, we perform time-critical tasks such as ADC sampling in the background, and noncritical functions such as LCD output in the foreground. The interrupt service routine passes the measured temperature to the foreground through an FIFO queue, and the main program has the responsibility of outputting the result to the LCD.

**Program 12.2**

Real-time measurement of temperature.

```
// table of multiple unsigned (x,y), piece-wise linear function
unsigned short const ADCtable[8] = { 0, 8, 50, 96, 146, 199, 255, 65535 };
unsigned short const Ttable[8] = { 500, 500, 450, 400, 350, 300, 250, 250 };
unsigned short LookUp(unsigned short data);
interrupt 11 void TOC3handler(void){ // TCNT is 4us
    unsigned short Data; // raw ADC result, 0 to 255
    Data = ADC_In(0x81); // 0 to 255
    Fifo_Put(LookUp(Data)); // 250 to 500 (0.1C)
    TC3 = TC3+25000; // every 0.1s at 4 MHz
    TFLG1 = 0x08; // acknowledge OC3
}
```

**Example 12.2** Design a system to measure the electrical activity in the heart. In particular, we wish to measure heart rate.

**Solution** Biopotentials are important measurements in many research and clinical situations. **Biopotentials** are electric voltages produced by individual cells and can be measured on the skin surface. The status of the heart, brain, muscles, and nerves can be studied by measuring biopotentials. Electrodes, which are attached to the skin, interface the machine to the body. Electronic instrumentation amplifies and filters the signal. For example, Figure 12.48 shows a normal Lead II **electrocardiogram**, or **EKG**, which is measured with the positive terminal attached to the left arm, the negative terminal attached to the right arm, and ground connected to the right leg. Each wave represents one heartbeat, and the shape and rhythm of this wave contain a lot of information about the health and status of the heart.

**Figure 12.48**  
Normal II-lead electrocardiogram.

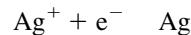


There are two types of electrodes used to record biopotentials. **Nonpolarizable** electrodes such as silver/silver chloride involve the following chemical reaction in the electrode at the electrode/tissue interface:



A nonpolarizable electrode has a low electrical impedance, because electrons can freely pass the electrode/tissue interface. In the electrode, current flows by moving electrons, but in the tissue, current flows by physical motion of charged ions (e.g.,  $\text{Na}^+$ ,  $\text{K}^+$ , and  $\text{Cl}^-$ ). A silver/silver chloride electrode does include a half-cell potential of 0.223 V, but since biopotentials are always measured with two electrodes, these half-cell potentials cancel. On the other hand, if you tried to use these electrodes to measure DC voltages, then the foregoing chemical reaction would saturate and fail. Fortunately, biopotentials produced by muscles and nerves are AC only and have no DC component.

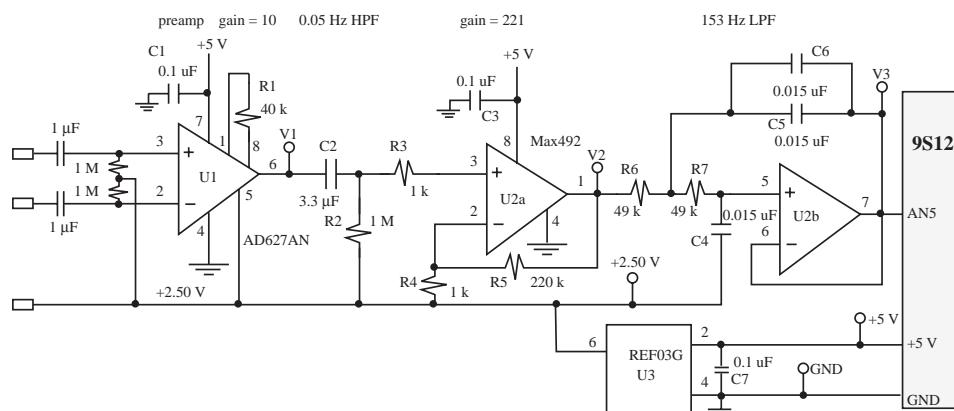
**Polarizable** electrodes, made from metals such as platinum gold or silver, have a high electrical impedance, because electrons cannot freely pass the electrode/tissue interface. Charge can develop at the electrode/tissue interface effectively creating a capacitive barrier. Displacement current can flow across the capacitor, allowing the AC biopotentials to be measured by the electronics. The metallic electrodes also include a half-cell potential, but again, these potentials will cancel.



The graphical display of EKG versus time is an example of a qualitative data acquisition system. The measurement of heart rate is quantitative. The parameters of an EKG amplifier include high input impedance (larger than  $1\text{ M}\Omega$ ), high gain, a 0.05 to 100 Hz bandpass filter, and a good common-mode rejection ratio. The EKG signal is about  $\pm 1\text{ mV}$ , so an overall gain of 2200 will produce a range of 0 to  $+5\text{ V}$  on V3. The data flow graph of this system is similar to Figure 12.46. This EKG amp (Figure 12.49) begins with a preamp stage having a good CMRR, high input impedance, and a gain of 50. Pin 5 of the AD627 is the analog ground, which in this circuit is the  $2.50\text{ V}$  reference voltage. The AD627 is rail-to-rail. A 0.05 Hz passive high-pass filter is created by R2 and C2. A low-leakage capacitor for C2 is critical for elimination of DC offset drift. Polypropylene or polystyrene would be a good choice for C2, but a low-leakage ceramic is acceptable. The remaining gain is performed with a non-inverting amplifier (U2a). The LPF is implemented as a 2-pole Butterworth LPF. The 153 Hz cutoff was chosen because it is greater than 100 Hz and can be implemented with standard components. If the signal V1 saturates, you can reduce the gain on the preamp and increase the gain on the amp.

**Figure 12.49**

A battery-power EKG instrument (see Figure 1.34).



Program 12.3 shows the real-time data acquisition and 60 Hz digital notch filter. The design of digital filters is presented in Chapter 15. The ADC sampling occurs in the background, and the data are passed to the foreground using an FIFO queue. The large pulse in the EKG, originating from the contraction of the ventricles, is called the R-wave, and it occurs once a heartbeat. Program 12.4 shows the foreground process, where there are four calculation steps performed on the EKG data. A low-pass filter followed by a high-pass filter capture a narrow band of information around 8 Hz. The square function calculates power, and the 200 ms-wide moving average gives an output very specific for the R-wave. Hysteresis is implemented with two thresholds. A heartbeat is counted (**RCount++**) when the moving average goes below the **LOW** threshold and then above the **HIGH** threshold. This software uses a combined frequency-period method to calculate heart rate. The theory of this method was developed as Exercise 6.3 in Chapter 6. The algorithm to measure heart rate searches for R-waves in a 5-second interval. **Rfirst** is the time (in 1/120 sec units) of the first R-wave, and **Rlast** is the time (also in 1/120 sec units) of the last R-wave. (**RCount-1**) is the number of beat-to-beat intervals between **Rfirst** and **Rlast**. The number 7200 is the conversion between the sample period (1/120 sec) and one minute. For example, at 72 BPM there will be 6 R-waves detected in the 5-second interval, making (**Rcount-1**) equal to 5, and the difference **Rlast-Rfirst** will be 500. For more information on EKG systems, see Webster's book *Medical Instrumentation*, published by Wiley, 1997, or Pan and Tompkins, "A Real-Time QRS Detection Algorithm," *IEEE Transactions on Biomedical Engineering*, pp. 230–236, March 1985.

**Program 12.3**

Real-time sampling of EKG.

```
interrupt 11 void TOC3handler(void){      // 480Hz
static short sum=0,n=0; short data;
    data = (0x00FF&ADC_In(7))-128;          // real EKG
    sum += data;      // sum=x(n)+x(n-1)+x(n-2)+x(n-3)
    n++;
    if(n==4){
        n = 0;
        Fifo_Put(sum/4);      // 120Hz
        sum = 0;
    }
    TC3 = TC3+16667;           // 2083.375us = 479.99 Hz
    TFLG1 = 0x08;             // acknowledge OC3
}
```

**Warning:** If you are going to build an EKG, please have a trained engineer verify the safety of your hardware and software before you attach patients to your machine.

**Program 12.4**

Measurement of heart rate.

```
short Data;      // ADC sample, -128 to +127, 8-bit signed ADC sample
short x[50];    // 60Hz notch-filtered EKG, 120Hz
short y[50];    // low pass filter, 120Hz
short z[50];    // high pass filter, 120Hz
short w[50];    // squared result, R-wave power, 120Hz
short Rwav;     // moving average of R-wave power, energy
unsigned short n=25;   // 25,26, ..., 49
unsigned short Trigger;
#define HIGH 100    // trigger when over this
#define LOW 20      // reset when under this
unsigned short Rcount;    // number of R-waves
unsigned short Rfirst;    // time of first R-wave
unsigned short Rlast;     // time of last R-wave
unsigned short HeartRate; // units bpm
void main(void) { unsigned short time;      // units 1/120sec
short lpfSum=0,hpfSum=0,RwavSum=0;
    LCD_Open(); LCD_OutString("EKG System - Valvano");
    Fifo_Init();
    ADC_Init();    // Activate ADC
    Timer_Init();  // initialize 480Hz OC3
    Trigger =0;    // looking for HIGH
    for(;;) {
        Rcount = 0;
        Rlast = 0;
        for(time=0;time<600;time++){ // 120 Hz, every 5 second
            while(Fifo_Get(&Data)){ // Get data from background thread
                LCD_Plot(Data); // draw voltage versus time plot
                n++; if(n==50) n=25;
                x[n] = x[n-25] = Data; // new data
                lpfSum = lpfSum+x[n]-x[n-4];
                y[n] = y[n-25] = lpfSum/4; // Low Pass Filter
                hpfSum = hpfSum+y[n]-y[n-10];
                z[n] = z[n-25] = y[n]-hpfSum/10; // High Pass Filter
                w[n] = w[n-25] = (z[n]*z[n])/10; // Power calculation
                RwavSum = RwavSum+w[n]-w[n-24]; // 200ms wide moving average
                Rwav = RwavSum/24;
        }
```

*continued on p. 636*

**Program 12.4 (cont'd)**

Measurement of heart rate.

*continued from p. 635*

```

        if(Trigger){
            if(Rpow<LOW){
                Trigger = 0;      // found low
            }
        } else{
            if((Rpow>>HIGH)&&((time-Rlast)>>30)){ // max HR= 240bpm
                Trigger = 1;      // found high
                if(Rcount){
                    Rlast = time; // mark time of last R-wave, units 1/120sec
                } else{
                    Rfirst = time; // mark time of first R-wave
                }
                Rcount++;
            }
        }
        if(Rcount>=2){
            HeartRate = (7200*(long)(Rcount-1))/(long)(Rlast-Rfirst);
        } else{
            HeartRate = 0;
        }
        LCD_OutUDec(HeartRate); // display results
    }
}

```

**Checkpoint 12.10:** What is the theoretical heart rate resolution of this approach when the HR is 60 BPM?

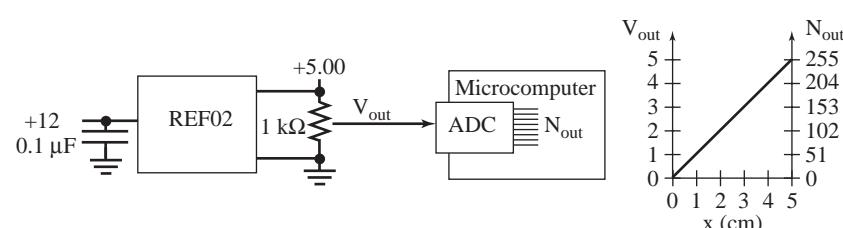
**Example 12.3** Design a system to measure position. The range is 0 to 5 cm and the resolution at least 0.02 cm.

**Solution** A potentiometer can be used to convert position into resistance. In this particular interface the full-scale range is 0 to 1000  $\Omega$ .

$$R_{out} = 200 \cdot x$$

where the units of  $R_{out}$  and  $x$  are in  $\Omega$  and centimeters, respectively. To interface this transducer to the microcomputer we drive the potentiometer with a stable DC voltage using a precision voltage reference (Figure 12.50). If we were to drive the circuit with the +5 V power, then any noise ripple in the power line would couple directly into the measurement. The ADC produces a digital output dependent on its analog input. The resolution is 5 cm/256, which is about 0.02 cm.

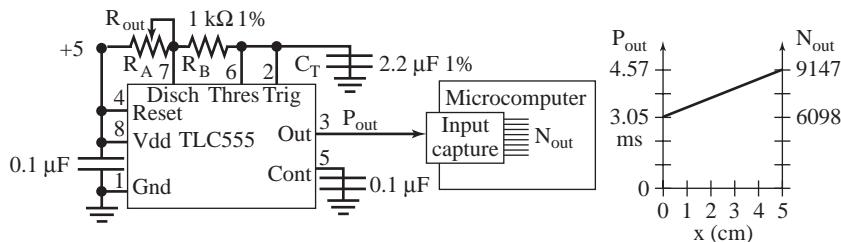
**Figure 12.50**  
Potentiometer interface using an ADC.



**Checkpoint 12.11:** What would be the position resolution if the ADC precision were increased from 8 to 10 bits?

Another approach to interface this transducer to the microcomputer would be to use an astable multivibrator. The period of a 555 timer is  $0.693 \cdot C_T \cdot (R_A + 2R_B)$ . In our circuit,  $R_A$  is  $R_{out}$ ,  $R_B$  is  $1\text{ k}\Omega$ , and  $C_T$  is  $2.2\text{ }\mu\text{F}$ . Given a fixed  $R_B$ ,  $C_T$ , the period of the square wave,  $P_{out}$ , is a linear function of  $R_{out}$ . Our microcontrollers have a rich set of mechanisms to measure frequency, pulse width, or period. To change the slope and offset of the conversion between  $R_{out}$  and  $P_{out}$ , the fixed resistor and capacitor can be adjusted. Even though the period does not include zero, the precision of this measurement is over 3000 alternatives, or more than 11 bits. The precision can be improved by increasing the capacitor  $C_T$ , or decreasing the period on the measurement clock (Figure 12.51). The position resolution is 5 cm/3049, which is about 0.002 cm.

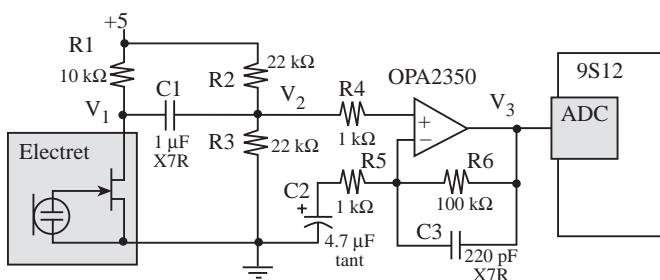
**Figure 12.51**  
Potentiometer  
interfacing using input  
capture.



#### Example 12.4 Design a system that can input and output sound.

**Solution** The electret microphone was described previously in Figure 12.11 as a low-cost, small-size transducer capable of converting sound into voltage. Many electret data sheets suggest an  $R_1$  of  $2\text{ k}\Omega$ , but signal-to-noise ratio can be improved by using a  $10\text{ k}\Omega$  resistor. Because the output of a HPF would normally include positive and negative voltages, we will need a way to offset the circuit so all voltages exist from 0 to  $+5\text{ V}$ , allowing the use of a single supply and rail-to-rail op amps.  $R_2$  and  $R_3$  provide an offset for the HPF, so the signal  $V_2$  will be the sound signal plus  $2.5\text{ V}$ . The effective impedance from  $V_2$  to ground is  $11\text{ k}\Omega$ , so the HPF cutoff is  $1/(2\pi * 0.22\text{ }\mu\text{F} * 11\text{ k}\Omega) = 66\text{ Hz}$ . The gain of the system is  $1+R_6/R_5$ , which will be 101. The capacitor  $C_2$  will make the signal  $V_3$  be the amplified sound plus  $2.5\text{ V}$ . The gain is selected so the  $V_3$  signal is  $2.5 \pm 1\text{ V}$  for the sounds we wish to record. The capacitor  $C_3$  provides a little low-pass filtering, causing the amplifier gain to drop to one for frequencies above  $1/(2\pi * 220\text{ pF} * 100\text{ k}\Omega) = 7.2\text{ kHz}$ . A better LPF would be to add an active LPF. The LPF would need a  $2.5\text{ V}$  offset like the one in Figure 12.49. If we wish to process sound with frequency components from 100 to 5 kHz, then we should sample at or above 10 kHz. If we sampled sound with a 10-bit ADC, we should select a 10-bit or 12-bit DAC to output the sound. We could improve signal to noise by replacing the  $+5\text{ V}$  connected to  $R_1$  and  $R_2$  in Figure 12.52 with a LM4040BIZ-4.1 4.096 V reference.

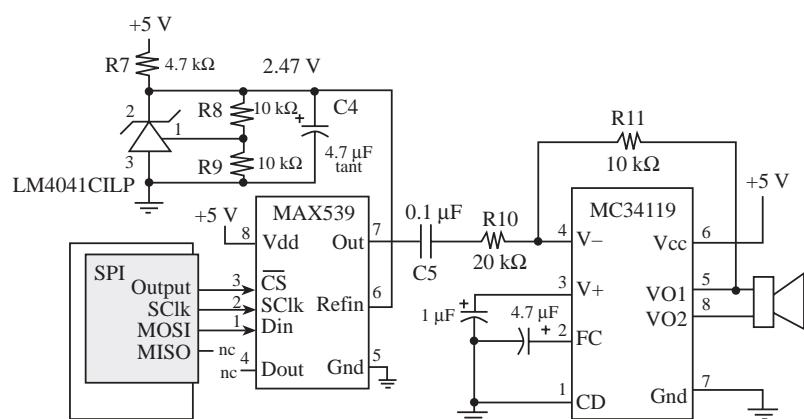
**Figure 12.52**  
An electret microphone  
can be used to record  
sound.



The LM4041CILP is a shunt reference used to make the analog reference required by the MAX539 12-bit DAC. The reference output will be  $1.233V^*(1+R9/R8)$ , which will be 2.47 V, as shown in Figure 12.53. The R7 resistor is selected to provide the necessary current. The LM4041 itself requires 70  $\mu$ A, the R8 and R9 resistors will sink 125  $\mu$ A, and the MAX539 needs 63  $\mu$ A. As a safety margin, we design the shunt reference to supply 500  $\mu$ A by setting  $R7 = (5 - 2.47)/500 \mu\text{A} = 5 \text{ k}\Omega$ . The MC34119 audio amp, first shown in Figure 11.70, can be used to amplify the DAC output providing the current needed to drive a typical 8  $\Omega$  speaker. The gain of the audio amplifier is  $2*R11/R10$ , which for this circuit will be one. This means a 2 V peak-to-peak signal out of the DAC will translate to a 2 V peak-to-peak signal on the speaker. The maximum power that the MC34119 can deliver to the speaker is 250 mW, so the software should limit the sound signal below 1.4 Vrms when driving an 8  $\Omega$  speaker. The quality of sound can be increased by selecting a better speaker and placing the speaker into an enclosure. For more information on how to design a speaker box, perform a Web search on “speaker enclosure.”

**Figure 12.53**

A DAC and an audio amplifier allow the microcontroller to output sound.



Software in Program 12.5 can be used to interface the MAX539 12-bit DAC. Program 12.6 performs the sound input and output. The sound will be processed in the foreground by getting input sound from the Rx FIFO and putting output sound into the Tx FIFO. The sampling rate is 10 kHz. The FIFO queues can be found as Program 4.14 and **ADC\_In** as Program 11.13.

### Program 12.5

Software to output to the MAX539 12-bit DAC.

```

void DAC_Init(void){ // PM3=CS=1
    DDRM |= 0x38; // PM5=SCLK=SPI clock out
    PTM |= 0x08; // PM4=Din=SPI master out
    SPICR1 = 0x50;
    SPICR2 = 0x00; // normal mode
    SPIBR = 0x00; // 12 MHz
}
void static send(unsigned char code){ unsigned char dummy;
    while((SPISR&0x20)==0){};// wait SPTEF
    SPIDR = code; // data out
    while((SPISR&0x80)==0){};// wait SPIF
    dummy = SPIDR; // clear SPIF
}
void DAC_Out(unsigned short code){ unsigned char dummy;
    PTM &= ~0x08; // clear PM3=Max539CS
    send(data>>8); // output msdata frame to DAC
    send(data&0xFF); // output lsdata frame to DAC
    PTM |= 0x08; // CS=1, latching data into DAC
}

```

**Program 12.6**

Real-time sound output  
input/output.

```
interrupt 11 void TOC3handler(void){ // 10 kHz
    RxFifo_Put(ADC_In(0x81)); // input sound
    if(TxFifo_Get(&data)) DAC_Out(data);
    TC3 = TC3+100; // every 100 us
    TFLG1 = 0x08; // acknowledge OC3
}
```

## 12.6 Exercises

**12.1** For each term, give a definition in 32 words or less.

- |                      |                        |                                     |
|----------------------|------------------------|-------------------------------------|
| <b>a)</b> Measurand  | <b>f)</b> Bang-bang    | <b>k)</b> Positive predictive value |
| <b>b)</b> Transducer | <b>g)</b> Deadzone     | <b>l)</b> Negative predictive value |
| <b>c)</b> Hysteresis | <b>h)</b> Phantom bias | <b>m)</b> Impedance loading         |
| <b>d)</b> Saturation | <b>i)</b> Prevalence   | <b>n)</b> Crest factor              |
| <b>e)</b> Breakdown  | <b>j)</b> Triple point |                                     |

**12.2** For each transducer parameter, give a definition in 32 words or less.

- |                       |                              |                        |
|-----------------------|------------------------------|------------------------|
| <b>a)</b> Accuracy    | <b>e)</b> Input impedance    | <b>h)</b> Slew rate    |
| <b>b)</b> Linearity   | <b>f)</b> Drift              | <b>i)</b> First order  |
| <b>c)</b> Sensitivity | <b>g)</b> Frequency response | <b>j)</b> Second order |
| <b>d)</b> Specificity |                              |                        |

**12.3** For each instrument parameter, give a definition in 32 words or less.

- |                         |                      |                                 |
|-------------------------|----------------------|---------------------------------|
| <b>a)</b> Accuracy      | <b>c)</b> Resolution | <b>e)</b> Reproducibility       |
| <b>b)</b> Maximum error | <b>d)</b> Precision  | <b>f)</b> Signal-to-noise ratio |

**12.4** For each transducer, give its measurand.

- |                       |                              |                             |
|-----------------------|------------------------------|-----------------------------|
| <b>a)</b> Thermistor  | <b>e)</b> Thermocouple       | <b>i)</b> ADXL202           |
| <b>b)</b> LVDT        | <b>f)</b> Ultrasonic crystal | <b>j)</b> Ag-AgCl electrode |
| <b>c)</b> Electret    | <b>g)</b> Shaft encoder      |                             |
| <b>d)</b> Strain gage | <b>h)</b> RTD                |                             |

**12.5** For each concept, give a definition in 32 words or less.

- |                          |                                |                         |
|--------------------------|--------------------------------|-------------------------|
| <b>a)</b> Nyquist Theory | <b>c)</b> Voltage quantization | <b>e)</b> Time jitter   |
| <b>b)</b> Aliasing       | <b>d)</b> Time quantization    | <b>f)</b> Sampling rate |

**12.6** For each type of noise, give a definition in 32 words or less. Also give one way to reduce the effect of this noise.

- |                       |                                |
|-----------------------|--------------------------------|
| <b>a)</b> White noise | <b>c)</b> Magnetic field noise |
| <b>b)</b> 1/f noise   | <b>d)</b> Electric field noise |

**12.7** Give a short answer in 32 words or less.

- a)** How do we choose the sampling rate?
- b)** How do we choose the gain of our amplifier?
- c)** When do we need a S/H module?
- d)** When does a data acquisition system have aliasing?

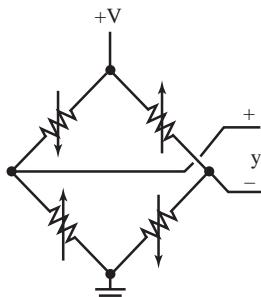
**D12.8** Design a computer-based DAS that measures pressure. The pressure transducer is built with four resistive strain gages placed in a DC bridge. When the pressure is zero, each gage has a  $120\ \Omega$  resistance making the bridge output 0. When pressure is applied to the transducer, two gages are compressed (which lowers their resistance) and two are expanded (which increases their resistance). At full-scale pressure ( $p = 100\ \text{dynes/cm}^2$ ), the bridge output is  $10\ \text{mV}$ . The transducer/bridge output impedance is therefore  $120\ \Omega$ . You may assume the transducer is linear. The desired pressure resolution is  $1\ \text{dyne/cm}^2$ . The frequencies of interest are 0 to 100 Hz, and the two-pole Butterworth analog low-pass filter will have a cutoff (gain = 0.707) frequency of 100 Hz (you will design it in part b). In terms of choosing a sampling rate, you may assume the low-pass filter removes all signals above 100 Hz.

- a)** Show the interface of the ADC to your computer. Justify your ADC precision.
- b)** Design the analog interface between the transducer/bridge and the your ADC. Use the full-scale ADC range even though it will complicate the conversion software. For example,

if the ADC has a range of 0 to +5 V, then a pressure of  $p = 0$  maps to a voltage at the ADC input of 0, and  $p = 100 \text{ dynes/cm}^2$  maps to a voltage at the ADC input of +5 V. Include the Butterworth low-pass filter. (Figure 12.54)

**Figure 12.54**

Transducers are placed in a bridge.

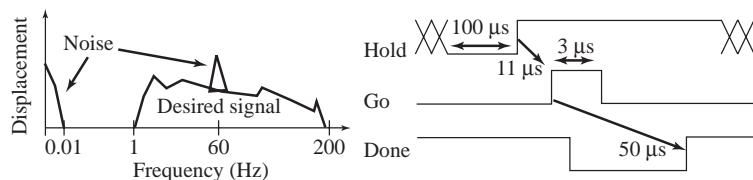


- c) Show the ritual that initializes any global variables, the ADC, and an appropriate interrupt.
- d) Show the interrupt handler(s) that samples the ADC, calculates pressure in  $\text{dynes/cm}^2$ , and stores the value in global variable Pressure. The software conversion maps a 0 ADC result into Pressure=0, and a 255 ADC result into Pressure=100. *Optimize the interrupt handler so that the number of execution cycles is minimized.*

- D12.9** The objective of this problem is to measure vibrations (displacement versus time) using a strain gage. The displacement range is  $-100$  to  $+100 \mu\text{m}$ . The frequencies of interest are 1 to 200 Hz. The four resistors of the strain gage are placed in a bridge: two in compression, two in expansion. The bridge output voltage  $V_b$  has a sensitivity of  $100 \text{ V/m}$ . Each resistance  $R$  in the bridge is  $100 \Omega$  at a zero displacement. There is a lot of unwanted DC and 60-Hz noise in this system. You will remove the unwanted DC signal using analog signal processing and remove the unwanted 60-Hz signal using digital signal processing.
- a) Show the interface between the transducer and the microcomputer. The S/H input impedance is  $1 \text{ T}\Omega$ . The S/H aperture time is  $10 \mu\text{s}$ , and the acquisition time is  $100 \mu\text{s}$ . The ADC input voltage range is  $-5$  to  $+5 \text{ V}$ , the ADC digital output is 12-bit signed 2s complement, and the conversion time is  $50 \mu\text{s}$  (Figures 12.55 and 12.56).

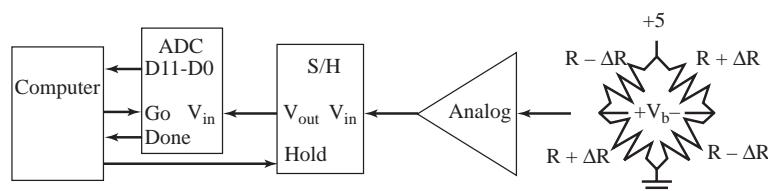
**Figure 12.55**

Vibration signals and ADC timing.



**Figure 12.56**

Vibration measuring system.



- b) What is the maximum allowable drop rate for the S/H? Show your work.
- c) What is the displacement resolution? Give units and show your work.
- d) What is the slowest sampling frequency that is feasible for this system. Justify your answer. You will implement a sampling rate of 480 Hz and execute the following digital filter to remove the 60 Hz.

$$y(n) = (x(n) + x(n - 4))/2$$

- e) Show the ritual subroutine that initializes the necessary microcomputer devices. Initialize all data structures including DONE. The output of your software is a global array, Y, containing

the filtered displacement versus time. The data acquisition and digital filtering will be performed in real time under interrupt control. The main program will call this ritual and perform other unrelated tasks until DONE is  $-1$ .

```
short Y[1000]; // filtered displacement measurements
// The resolution and units are given in c), fs is 480 Hz
char DONE; // initially zero, set to -1 when buffer is full
```

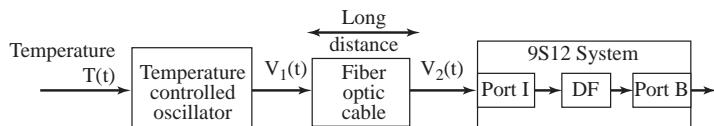
- f) Show the interrupt software which samples the ADC, implements the digital filter in real time, and saves the filter outputs in the buffer Y. Set DONE to  $-1$  when the buffer is full.

**D12.10** Design a remote-sensor temperature acquisition system. The goals are

Range	$1 \leq T \leq 200^{\circ}\text{F}$
Resolution	$\Delta T = 1^{\circ}\text{F}$
Frequencies of interest	DC to 5 Hz

The system has the block diagram shown in Figure 12.57. The timer is Port T on the 9S12.

**Figure 12.57**  
Remote temperature measuring system.



$V_1(t)$  is a low-power Schottky TTL square wave with a period  $P(t)$  that is linearly related to the temperature  $T(t)$ .

$$P(t) = \frac{10 \mu\text{s}}{^{\circ}\text{F}} T(t)$$

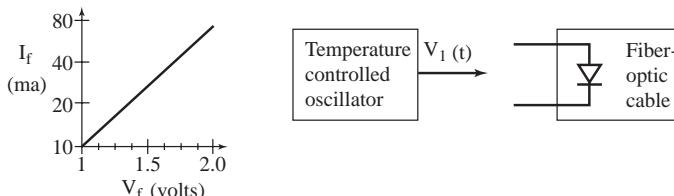
Your interface should connect the fiber-optic cable so that  $V_2(t)$  is also a TTL-level square wave with the same period. The 9S12 input capture/output compare system should be used to measure  $T(t)$  at equal intervals. Let  $f_s$  be the fixed sampling rate, and  $\Delta t = 1/f_s$ . Your system should measure  $T(t)$  exactly  $f_s$  times per second. Let  $x(n)$  be the temperature sampled every  $\Delta t$  with a resolution of  $1^{\circ}\text{F}$ . Your 9S12 system will implement the following low-pass finite impulse response (FIR) digital filter:

$$y(n) = \frac{y(n-1) + x(n)}{2}$$

The output of this filter,  $y(n)$ , should be written to Port B every  $\Delta t$  s.

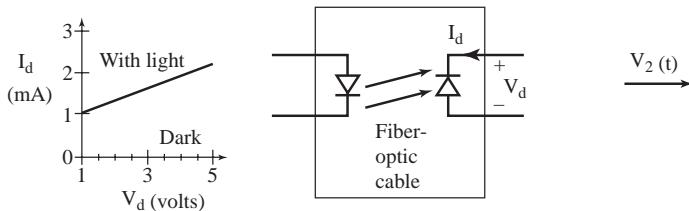
- a) Show the interface between  $V_1$  and the fiber-optic cable. The front end operates like a LED. Make  $I_f = 20 \text{ mA}$  (Figure 12.58).

**Figure 12.58**  
Fiber-optic cable transmitter interface.



- b) Show the interface between the fiber-optic cable and  $V_2$  (Figure 12.59). The output of the fiber-optic cable is a photodiode detector. The detector conducts when light shines on it. It is open when dark. Provide for some noise rejection.

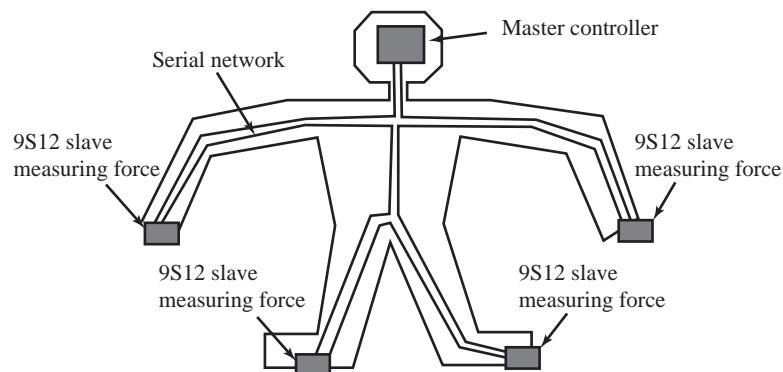
**Figure 12.59**  
Fiber-optic cable receiver interface.



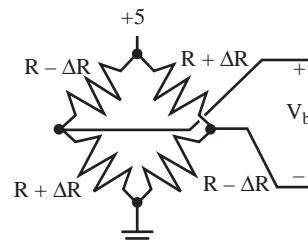
- c) Choose the slowest possible sampling rate  $f_s$ . Justify your answer.
- d) Show connections between the square wave  $V_2(t)$  and the 6811/6812.
- e) Show global data structure required to implement the digital filter. Full credit will be given to the best dynamic and static efficiency. This is not a question as to whether assembly is more efficient than C, but rather which data structure gets the job done in the simplest manner.
- f) Show the ritual and interrupt handler(s). After calling the ritual, the main program (foreground) is free to execute other unrelated tasks. The interrupt handler(s) (background) will implement the fixed rate data acquisition. The output of the digital filter should be output to Port B  $f_s$  times per second. You may use either assembly or C.

**D12.11** The objectives of this problem are (1) to use a single-chip embedded 9S12 with its ADC to measure force at various locations on a robot and (2) to transmit the measurements across a master/slave distributed network. The robot system will have multiple slaves, each with its own force-measuring circuitry (Figure 12.60). You will design the slaves and specify/implement the communication channel to the master. The force measurement range is  $0 \leq F \leq 200$  dynes. The desired force resolution is 1 dyne. The signals of interest are 0 to 10 Hz. Each slave has +5, +12, and -12 V power. The force transducers are placed in a resistance bridge powered by the +5 V supply (Figure 12.61). The bridge output has a sensitivity of 0.05 mV/dyne. The bridge output is *almost* zero when the force is zero. This transducer offset for each slave is a constant within the range from 0 to 0.2 mV. Each slave will require a no-force calibration that you will decide whether to handle in software or hardware.

**Figure 12.60**  
Distributed force  
measuring system.



**Figure 12.61**  
The force transducers  
are placed in a bridge.



- a) Show the analog interface between the bridge and the 9S12 ADC channel 1. Choose the appropriate op amp circuit, gain, offset, and analog filter. Include a mechanism to calibrate if you decide to handle the offset adjustment in hardware.
- b) Design the distributed network between the slaves and their master. All slaves have identical network connections. To get a measurement from one of the slaves, the master will transmit the slave address that should activate exactly one slave. The slave will respond with the current force with a resolution of 1 dyne. The first priority of your network is to minimize cost (i.e., fewest wires and interface chips). Given the cheapest network, the second priority is to

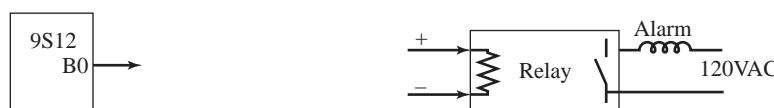
maximize bandwidth. Clearly show whether or not the grounds are connected between the multiple computers.

- b1)** Show the hardware interface between a single-chip 9S12 slave and the network. Remember that the hardware for all slaves will be identical. There is no noise interference.
- b2)** Show an example communication between the master and a single-chip 9S12 slave. Clearly identify the slave address, and force data components. Label the time axis.
- c)** Include all of the software that will exist in each slave. Your solution will be segmented into three parts: RAM, EEPROM, and ROM. Specific software requirements include the following:
  - Other than a one-time initialization, there will be no foreground (main) program
  - You may use the 9S12 ADC continuous scan mode
  - Clearly show where the 9S12 is to begin execution on a power on reset
  - Include a mechanism to calibrate if you decide to handle the offset adjustment in software
- c1)** Show the software that goes in the RAM (uninitialized on power up).
- c2)** Show the software that goes in the EEPROM (nonvolatile, but can be different for each slave).
- c3)** Show the software that goes in the ROM (nonvolatile, and must be the same for each slave).

- D12.12** The objective of this problem is to design a DAS alarm. An alarm should sound when noise is present in the room. Assume the output of the microphone,  $x(t)$ , is a  $\pm 10$  mV differential signal. The signals of interest are 100 to 500 Hz. You will sample the sound and calculate the sound energy once a second (sum of the signal squared). Be careful to subtract off the DC so that a quiet room (microphone = 0) results in an energy calculation of zero. If data are the 8-bit ADC, then  $x(n) = \text{data} - 128$  represent the current 8-bit signed sample. If you sample at 1000 Hz, sound the alarm if  $(x(0) + x(1) + \dots + x(999))$  greater than 100,000. *32-bit-long integer operations are allowed.* The 9S12 system has an 8-MHz E clock. The alarm can be activated by driving 20 mA through a 5-V EM relay. The software should turn on the alarm if the energy level is above a threshold (simply pick any constant). Once on, the alarm should continue until the operator types the code “213” on the keypad. This 9S12 will have a secret code of 213, but allow each 9S12 to have a different three-number code and a different threshold. The computer is dedicated to this task.
- a)** Show the analog interface between the differential microphone output and the 9S12 ADC channel 2. Choose the appropriate op amp circuit, gain, offset, and analog filter. The Analog Devices AD680 is a three-wire, low-cost, 2.5 V reference (+5 V input, ground +2.500 V output). This is a *qualitative*, and not a quantitative, DAS.
  - b)** Design the interface between the 9S12 signal B0 and the EM relay (Figure 12.62). To activate the alarm, the relay coil requires a voltage between +3.5 and 6.0 V and a current of 20 mA.

**Figure 12.62**

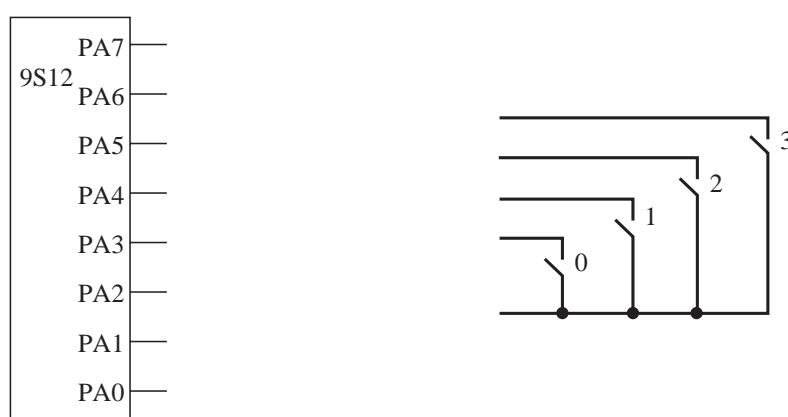
The alarm is controlled by an EM relay.



- c)** Show the interface between the keypad and the 9S12 (Figure 12.63). Each switch will bounce for about 10 ms, so you must implement a way to debounce. No other 9S12 connections may be used for this interface (although you may use the internal features of input capture/output compare). Your solution should minimize cost. *Periodic polling* should be used for this keypad because it is cheaper.
- d)** Include all of the software that will exist in the system. Your solution will be segmented into three parts: RAM, EEPROM, and ROM. Specific software requirements include the following:
  - Other than a one-time initialization, there will be no foreground (main) program
  - Use the ADC continuous scan mode to simplify the software
  - Clearly show where the 9S12 is to begin execution on a power on reset

**Figure 12.63**

There are four switches in the keypad.



- d1)** Show the software that goes in the RAM (uninitialized on power up).
- d2)** Show the software that goes in the EEPROM (nonvolatile, but can be different for each device).
- d3)** Show the initialization (ritual) software that goes in the ROM (nonvolatile, and must be the same for each device). This is where your system starts executing. Assume the two interrupt vectors are set by the compiler.
- d4)** Show the 1 kHz DAS interrupt handler that goes in the ROM (nonvolatile, and must be the same for each device). It is here that you will turn on the alarm if the sound energy goes above threshold.
- d5)** Show the keypad periodic polling interrupt handler that goes in the ROM (nonvolatile, and must be the same for each device). It is here that you will turn off the alarm if the operator types in the secret code.

**D12.13** This problem deals with the classification and reduction of noise.

- a)** Describe a *single experimental procedure* (measurement) that could *identify* (or differentiate) the *type(s)* of *noise* existing on the circuit.
- b)** For each of these three types of noise (white noise, 60 Hz noise, or 1/f noise), give a typical outcome of the experimental procedure.
- c)** Give one approach (other than analog or digital filtering) that will reduce white noise.
- d)** Give one approach (other than analog or digital filtering) that will reduce 60 Hz noise.
- e)** Give one approach (other than analog or digital filtering) that will reduce 1/f noise.

**D12.14** A temperature transducer has the relationship

$$R = 200 + 10T \quad \text{where } R \text{ is in } \Omega \text{ and } T \text{ is in } ^\circ\text{C}.$$

The problem specifications are

Range is  $30 \leq T \leq 50^\circ\text{C}$

Resolution is  $0.01^\circ\text{C}$

Frequencies of interest are 0 to 100 Hz

Transducer dissipation constant is  $20 \text{ mW}/^\circ\text{C}$

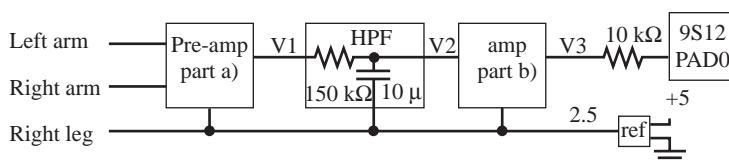
ADC range is 0 to  $+5 \text{ V}$

Sampling rate is 1000 Hz

- a)** How many ADC bits are required?
- b)** What is the maximum allowable noise at the amplifier *output*?
- c)** Design the *analog amplifier/filter*.

**D12.15** Design an EKG amplifier interface to a 9S12 (Figure 12.64). The biopotential is a  $\pm 5 \text{ mV}$  signal measured between the left and right arms using silver–silver chloride electrodes. A third electrode is placed on the right leg and is used as a reference. To make the entire system battery-operated, low-power, rail-to-rail TLC2274 op amps will be used, each powered with  $+5$  and ground. Notice that the analog system uses the  $2.5 \text{ V}$  signal as its reference.

**Figure 12.64**  
Block diagram of the EKG instrumentation.



- a) Design the preamplifier that has the following characteristics: differential input, gain=10, good CMRR, high Zin, low Zout, bandwidth > 200 Hz. Show resistor values, but not pin numbers.
  - b) Show the amplifier stage that has the following characteristics: single input, gain=250, high Zin, low Zout, bandwidth > 200 Hz. Show resistor values, but not pin numbers.
  - c) The signals of interest are 0.1 to 100 Hz. There is unwanted noise at 120, 240, 360, and 480 Hz. Discuss the trade-offs between the following two options:
    - c1) Add an active analog low-pass filter with a 100 Hz cutoff, and sample at 200 Hz.
    - c2) Sample at 1920 Hz and add a digital 120, 240, 360, 480 Hz reject filter.

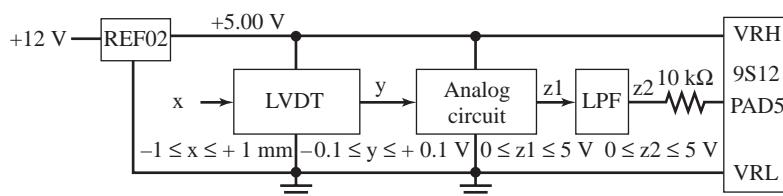
Which approach would you take and why? Under what conditions would the other choice be better?

- D12.16** Design a wind direction measurement instrument using the 8-bit ADC. You are given a transducer with a resistance that is linearly related to the wind direction. As the wind direction varies from 0 to 360 degrees, the transducer resistance varies from 0 to  $1000\ \Omega$ . The frequencies of interest are 0 to 0.5 Hz, and the sampling rate will be 1 Hz. (See Exercise 6.5.)

  - a) Show the analog interface between the transducer and the ADC port channel 7. Only the +5 V supply can be used. Show how the analog components are powered. Give chip numbers, but not pin numbers. Specify the type and tolerance of resistors and capacitors.
  - b) Write the ritual and gadfly function/subroutine that measures the wind direction and returns a 16-bit unsigned result with units of degrees. That is, the value varies from 0 to 359. (You do not have to write software that samples at 1 Hz, simply a function that measures wind direction once.)

- D12.17** Design a position DAS using a sensor, analog electronics, and a Freescale microcomputer with an 8-bit ADC. Let  $\mathbf{x}$  be the position to be measured (Figure 12.65). The input range is  $-1$  to  $+1$  mm, and the signals of interest are 0 to 1 Hz. A LVDT will be used to convert position  $\mathbf{x}$  into voltage  $\mathbf{y}$ . When the input position is  $-1$  mm, the LVDT output  $\mathbf{y}$  is  $-100$  mV. When the input position is zero, the LVDT output,  $\mathbf{y}$ , is zero. When the input position is  $+1$  mm, the LVDT output  $\mathbf{y}$  is  $+100$  mV. In between, the voltage output is linearly related to the position. A REF02 precision reference will provide the constant  $+5.00$  V for your analog circuit and the microcomputer ADC.

**Figure 12.65**  
Block diagram of the position measuring system.



- a) What is the transducer sensitivity? Give units.
  - b) What is the transfer relationship required to convert  $y$  into  $z_1$ ?
  - c) What is the maximum allowable noise for your analog circuit? Refer the noise to the amplifier input and give units.
  - d) Choose an appropriate sampling rate.
  - e) Design the preamplifier which has the following characteristics: single input (not differential), gain so that the 0 to +5 V ADC is used, high  $Z_{in}$ , low  $Z_{out}$ , bandwidth > 200 Hz. Show resistor values, but not pin numbers. You may use any analog op amps, but please include the chip number.

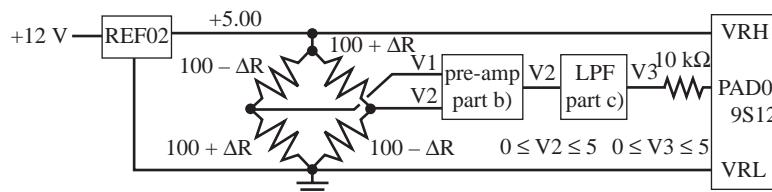
- f) The signals of interest are 0 to 1 Hz. There is unwanted noise at 60 Hz. Add a two-pole low-pass filter with a cutoff of about 10 Hz.  
 g) What is the system resolution in millimeters?

**D12.18** Design an electronic scale using a Freescale microcomputer with an 8-bit ADC (Figure 12.66). Let  $x$  be in mass to be measured. The input range is 0 to 1 kg and the signals of interest are 0 to 10 Hz. A bonded strain gage bridge will be used to convert mass  $x$  into voltage,  $V_1 - V_2$ . When the input mass is zero, each arm of the bridge is  $100 \Omega$ , and the bridge output ( $V_1 - V_2$ ) is zero. At full scale ( $x = 1 \text{ kg}$ ), two resistors go to  $99 \Omega$  and the other two go to  $101 \Omega$ . In between 0 and 1 kg, the resistance change is linearly related to the mass:

$$\Delta R = x \quad \text{where resistance is in ohms and the mass } x \text{ is in kilograms}$$

**Figure 12.66**

Block diagram of the mass measuring system.



A REF02 precision reference will provide the constant +5.00 V for the bridge and the ADC.

- a) What is the bridge output ( $V_1 - V_2$ ) at full scale ( $x = 1 \text{ kg}$ )? What gain is required to match the full range of  $0 \leq x \leq 1 \text{ kg}$  to the 0 to +5 V range of the ADC?  
 b) Design the preamplifier that has the following characteristics: differential input, gain so that the 0 to +5 V ADC is used, good CMRR, high  $Z_{in}$ , low  $Z_{out}$ , bandwidth > 200 Hz. Show resistor values, but not pin numbers.  
 c) The signals of interest are 0 to 10 Hz. There is unwanted noise at 60 Hz. Add a two-pole low-pass filter with a cutoff of 20 Hz.  
 d) What is the system resolution? Give units.  
 e) What sampling rate would you choose? Explain your answer.

**D12.19** You will design the analog hardware for a data-acquisition system to measure displacement (i.e., vibrations, distance). The range of displacement is -1 to +1 mm. Each resistance in the bridge ( $R_1, R_2, R_3$ , and  $R_4$ ) is linear to displacement having a sensitivity of  $10 \Omega/\text{mm}$ . At zero displacement, all four resistors are  $1000 \Omega$ . The displacement signal exists in the 0 to 1000 Hz frequency band. With displacement equal to -1 mm,  $R_1 = 1010, R_2 = 990, R_3 = 990$ , and  $R_4 = 1010 \Omega$ . With displacement equal to +1 mm,  $R_1 = 990, R_2 = 1010, R_3 = 1010$ , and  $R_4 = 990 \Omega$ .

- a) Create a design table with displacement  $R_1, R_2, R_3, R_4$ , bridge output ( $V_2 - V_1$ ), and ADC input.  
 b) A good CMRR is required. Design the analog circuit using just the single +5 V supply mapping the bridge output ( $V_2 - V_1$ ) into the ADC input channel 3 (V3). You do not need to add an antialiasing analog low-pass filter. Show chip numbers, resistor values, but not pin numbers.

**D12.20** You will design a data-acquisition system to measure force. The range of force is -10 to 10 N. The force signal exists in the 0 to 99 Hz frequency band. You will measure the ADC at a fixed rate, convert each sample to signed decimal fixed point, and write each fixed-point measurement into global variable called Force (which is the integer part of the fixed-point number). The ADC sampling must occur in real-time using an output compare periodic interrupt **Channel 5**.

- a) Choose the sampling rate. Explain why you chose that value.  
 b) The sensitivity of the bridge output is  $0.025 \text{ V/N}$ , meaning the differential voltage ( $V_2 - V_1$ ) varies from -0.25 to +0.25 V. However, both  $V_1$  and  $V_2$  are around 2.5 V (more specifically  $V_2$  varies from 2.375 to 2.625 V, while  $V_1$  varies from 2.625 to 2.375 V). A good CMRR is required. Design the analog circuit mapping the bridge output into the ADC input Channel 3. (You do not have to add an antialiasing analog low-pass filter.) Show chip numbers, resistor values, but not pin numbers.  
 c) Assuming the only error occurs in the 10-bit ADC, what is the expected force measurement resolution in Newtons (N)?

- d) Write the **entire program** that implements this real-time data-acquisition system. You are allowed to call the following functions (without showing the implementations of these functions).

```
PLL_Init();           // initializes the PLL, E clock to 24 MHz
ADC_Init();          // initializes the ADC
data=ADC_In(0x83); // returns 10-bit sample from chan 3, 0 to 1023
```

If you need other functions, you will have to show their implementations. Be sure to include the software that maps unsigned ADC (0 to 1023) into a signed fixed point (resolution you choose in part c). Include the output-compare 5 initialization and ISR.

## 12.7 Lab Assignments

**Lab 12.1** This experiment will use a thermistor and the ADC converter on the 9S12 to construct a digital thermometer. The temperature range should be 20 to 40°C. If the current temperature is above the upper limit in the specified range, a red LED should be turned on. You can test this feature by shorting the thermistor leads together (zero resistance). If it is below the lower limit of the specified range, a yellow LED should be turned on. Similarly, you can test this feature by disconnecting one wire of the thermistor (infinite resistance). Otherwise, a green LED will stay on, indicating that the temperature is within the specified range. The temperature measurements will be displayed as fixed-point numbers on an LCD. Design your system with the best possible resolution. The temperature component is 0 to 1 Hz. Experimentally verify the noise level, time-constant, and accuracy of your system.

**Lab 12.2** The objective of this lab is to build a digital sound recorder for human speech. You will first need to interface an external RAM to store the data. Next, you will need to design an analog circuit that interfaces a microphone to the ADC of the 9S12. Investigate the frequency components of human speech and design your system to pass these frequencies. You will also need a mechanism to play back the recorded sound, so design an audio amplifier that interfaces a DAC to a speaker. Your human interface should include buttons to trigger sound recording, stop recording, start playback, and stop playback.

**Lab 12.3** Design a thermocouple-based thermometer with a range of 0 to 50°C. You can use an ice bucket for the reference, or you could design the thermistor-based thermometer of Lab 12.1 and use it to compensate for the cold junction of the thermocouple. Design your system with the best possible resolution. The temperature measurements will be displayed as fixed-point numbers on an LCD. The temperature component is 0 to 1 Hz. Experimentally verify noise level, time-constant, and accuracy of your system.

**Lab 12.4** Design a digital scale using a force transducer. Select the range of the scale to match the linear range of your force transducer. You can build a force transducer using a slide pot and a spring. Design your system with the best possible resolution. The force measurements will be displayed as fixed-point numbers on an LCD. Experimentally verify the noise level, time-constant, and accuracy of your system.

**Lab 12.5** Design two digital position measurement systems using a slide potentiometer as the transducer. Select the range of the scale to match the linear range of your transducer. The position measurements will be displayed as fixed-point numbers on an LCD. The first system will use the ADC, and the second system will use an astable multivibrator (LM555) and input capture. Design your systems with the best possible resolution. Experimentally verify the noise level, time-constant, and accuracy of both systems.

**Lab 12.6** Design an autoranging voltmeter. The three ranges are 0 to 2 V, 0 to 0.2 V, and 0 to 0.02 V. The hardware/software system automatically adjusts the range providing the best possible measurement resolution. Display both the numerical and graphical results on an LCD display. The HD44780 LCD allows you to define up to 8 new character images (ASCII codes 0 to 7). By dynamically setting these 8 by 5-bit characters, you can create an 8-bit high by 40-bit wide graphical image. Write a graphical device driver for the LCD and use it to graph the time-varying voltage in real time. Experimentally determine the measurement resolution for each range and compare it to the expected theoretical resolution. Analyze the various sources of measurement error in your system.

# 13 Microcomputer-Based Control Systems

## Chapter 13 objectives are to

- ❖ Introduce the general approach to digital control systems
- ❖ Design and implement some simple open-loop control systems
- ❖ Design and implement some simple closed-loop control systems
- ❖ Develop a methodology for designing PID control systems
- ❖ Present the terminology and give examples of fuzzy logic control systems

In the last chapter, we developed systems that collected information concerning the external environment. In this chapter we will use this information to control the external environment. To build this microcomputer-based control system we will need an output device that the computer can use to manipulate the external environment. If we review the list of applications introduced in Section 1.1, we find that many involve control systems. Control systems originally involved just analog electronic circuits and mechanical devices. With the advent of inexpensive yet powerful microcomputers, implementing the control algorithm in software provided a lower cost and a more powerful product. The goal of this chapter is to provide a brief introduction to this important application area. Control theory is a richly developed discipline, and most of the theory is beyond the scope of this book. Consequently, this chapter focuses more on implementing the control system with an embedded computer and less on the design of the control equations.

### 13.1 Introduction to Digital Control Systems

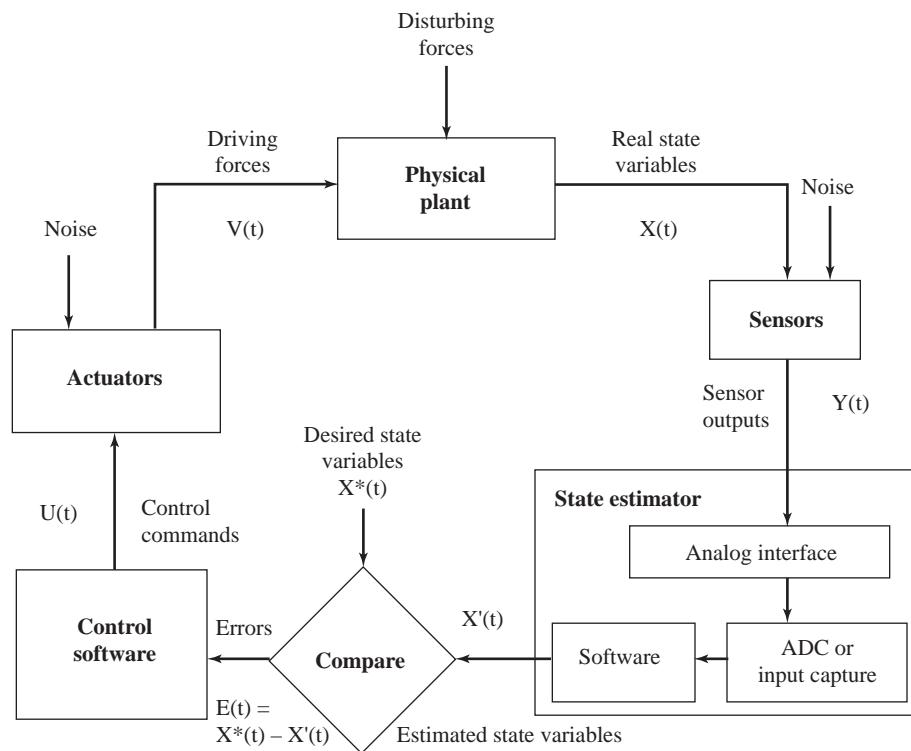
A *control system* is a collection of mechanical and electric devices connected for the purpose of commanding, directing, or regulating a *physical plant*. The *real state variables* are the properties of the physical plant that are to be controlled. The *sensor* and *state estimator* comprise a DAS, as discussed in Chapter 12. The goal of this DAS is to estimate the state variables. A *closed-loop* control system uses the output of the state estimator in a feedback loop to drive the system to a desired state. The control system compares these *estimated state variables*  $X'(t)$  to the *desired state variables*  $X^*(t)$  to decide appropriate action  $U(t)$ . The *actuator* is a transducer that converts the control system commands  $U(t)$  into driving forces  $V(t)$  that are applied to the physical plant. In general, the goal of the control system is to drive the real state variables to equal the desired state variables. In actuality though, the controller attempts to drive the estimated state variables to equal the desired state variables. It is important to have an accurate state estimator, because any

differences between the estimated state variables and the real state variables will translate directly into controller errors. If we define the error as the difference between the desired and estimated state variables,

$$e(t) = X^*(t) - X'(t)$$

then the control system will attempt to drive  $e(t)$  to zero. In general control theory,  $X(t)$ ,  $X'(t)$ ,  $X^*(t)$ ,  $U(t)$ ,  $V(t)$ , and  $e(t)$  refer to vectors (e.g.,  $x$ ,  $y$ ,  $z$  position or  $V_x$ ,  $V_y$ ,  $V_z$  velocity), but the examples in this chapter control only a single parameter. The focus of this book is the microcomputer interfacing, and it should be straightforward to apply standard multivariate control theory to more complex problems. We usually evaluate the effectiveness of a control system by determining three properties: steady-state controller error, transient response, and stability. The *steady-state* controller error is the average value of  $e(t)$ . The *transient* response is how long the system takes to reach 99% of the final output after  $X^*$  is changed. A system is *stable* if a steady state (smooth constant output) is achieved. An *unstable* system oscillates (Figure 13.1).

**Figure 13.1**  
Block diagram of a microcomputer-based closed-loop control system.

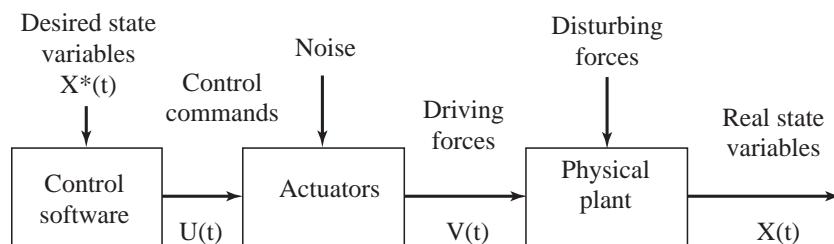


## 13.2 Open-Loop Control Systems

An *open-loop* control system does not include a state estimator. It is called open loop because there is no feedback path providing information about the state variable to the controller. It will be difficult to use open loop with the plant that is complex because the disturbing forces will have a significant effect on controller error. On the other hand, if the plant is well-defined and the disturbing forces have little effect, then an open-loop approach may be feasible (Figure 13.2).

**Figure 13.2**

Block diagram of a microcomputer-based open-loop control system.

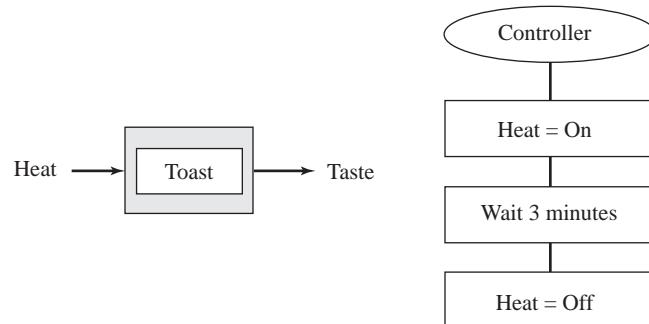


### Example 13.1 Design an open-loop controller for a toaster.

**Solution** This first example is a simple toaster (Figure 13.3). It is an open-loop control system because the control signal (heat applied to the bread) is independent of the state variable (taste of the toast). The controller simply applies heat for a fixed amount of time. If we provide an adjustable heat cycle to our toaster and allow the humans to adjust the heating cycle according to their taste, then the toast/humans combination becomes a closed-loop control system. Because an open-loop control system does not know the current values of the state variables, large errors can occur.

**Figure 13.3**

A simple open-loop control system.

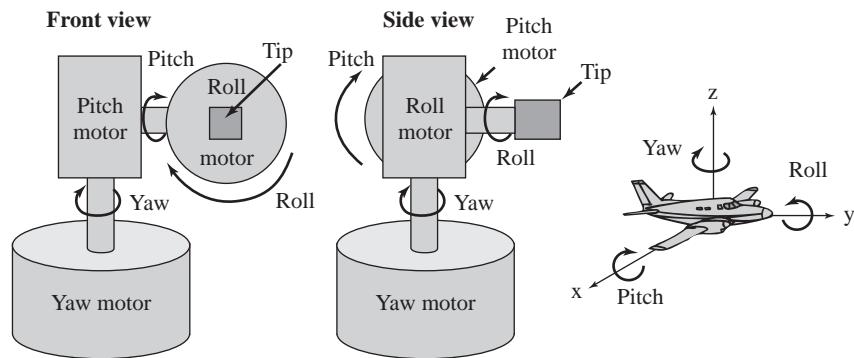


### Example 13.2 Design an open-loop controller for a three-axis robotic arm. The goal is to control the orientation of the tip.

**Solution** The three-axis robotic arm is shown in Figure 13.4. The three parts of rotational orientation are called pitch, roll, and yaw. For this simple robotic arm, there are independent

**Figure 13.4**

A three-axis open-loop stepper motor control system.



stepper motors controlling each axis. Although a typical robot arm would have more motors, and each motor would affect both translational and rotational motion, the design process illustrated in this example can be used for a wide range of open-loop controllers. Furthermore, this implementation can easily be extended into a closed-loop system by adding sensors and using the sensor information to adjust the command inputs.

**Pitch** is rotation around the lateral or transverse axis (x). For an airplane, this axis is parallel to the wings, thus the nose and tail can pitch both up or down. **Roll** is rotation around the longitudinal axis (y). For an airplane, this axis is drawn through the body of the vehicle from tail to nose. **Yaw** is rotation about the axis normal to the ground (z), which is perpendicular to the pitch and roll axes. If an airplane were placed on a flat surface and pivoted about the center of mass (coordinate origin), the motion would be described as yawing.

If the motors are strong enough, the system can be controlled in an open-loop fashion. For safety considerations, open-loop systems are not appropriate. An industrial robot needs feedback to prevent the robot from damaging itself or its environment. Safety features can easily be added to this implementation. There are four digital signals the computer uses to control each stepper motor. If the software outputs the sequence 6,5,10,9,6,5,10,9 . . . the motor will spin. For each change in output (6 to 5, 5 to 10, 10 to 9, or 9 to 6), the motor turns 1.8 degrees. Therefore, it takes 200 outputs to turn the shaft one complete rotation. The goal of the system is to simultaneously control the position and speed on three stepper motors. The acceleration and jerk parameters will not be explicitly controlled. Again, without sensors to measure the shaft position, this system is an open-loop controller. The control signals (power to the stepper coils) are independent of the state variables (shaft positions). For low torque applications (when the motor does not skip), the software can keep track of the shaft position without a sensor. For discussions of interfacing and minimizing jerk in the stepper motor section, refer back to Chapter 8.

In the **analysis phase**, we determine the requirements and constraints for our proposed system. For this open-loop system, the three motors must perform the actions specified by the speed and duration, where the acceleration and jerk are not specified. The ten-second pattern will be repeated continuously.

<b>Yaw</b>	<b>Pitch</b>	<b>Roll</b>
1 RPM, 1 sec	0 RPM, 2 sec	0 RPM, 1 sec
1 RPM, 1 sec	1 RPM, 2 sec	1 RPM, 2 sec
2 RPM, 0.5 sec	2 RPM, 1 sec	4 RPM, 0.5 sec
0.5 RPM, 2 sec	0 RPM, 5 sec	0 RPM, 6.5 sec
0 RPM, 5.5 sec		

**Checkpoint 13.1:** The 3-axis robot described in Figure 13.4 also generates some translational motion. Describe qualitatively the (x,y,z) position of the tip as a function of the yaw, pitch, and roll angles.

In the **high-level design** phase, we define our input/output, break the system into modules, and show the interconnection using data flow graphs. In this system we will have a central controller, executed using periodic interrupts. A data flow graph is shown in Figure 13.5, where the input is the desired speed of each motor and the outputs are 12 digital outputs, four for each stepper motor. The Yaw and Pitch motors are interfaced to Port T, and the Roll motor is connected to PM3-0. The high-level master controller will perform commands such as rotate at 1 RPM for 1 second, and the three low-level controllers will perform the commands step clockwise and step counter-clockwise. There will be a **Time** and **Direction** for each motor. **Time** will be the number of msec in between outputs to the stepper motor. To design the low-level controller,

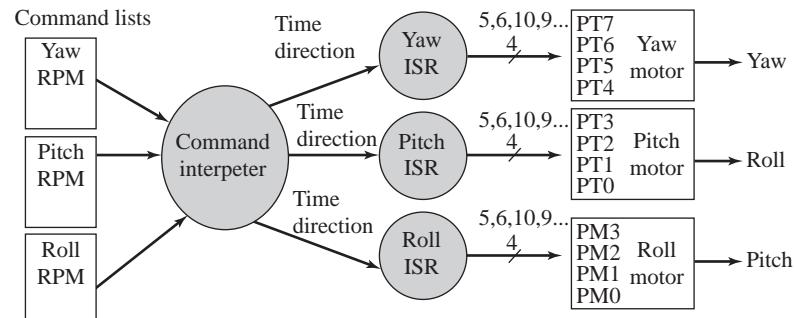
we must consider the time in between outputs for all the desired speeds. For the fastest speed of 4 RPM, there will be

$$1000(\text{ms/s}) * 60(\text{s/min}) / 200(\text{steps/rev}) / 4(\text{rev/min}) = 75 \text{ ms per step.}$$

Similarly, the speeds of 2, 1, 0.5 RPM will require a **Time** of 150, 300, and 600 respectively. A **Time** of zero will be defined as stop. **Direction** specifies the motion as clockwise or counterclockwise.

**Figure 13.5**

Data flow graph for the control system.



In the **engineering design** phase, we design the hardware/software subcomponents using techniques such as simulation, mechanical mockups, and call-graphs. The weights of the motors and tips together with the lengths of the shafts will determine the required torques of the stepper motors. The required torque should also include friction forces on the shaft. The electrical interface of stepper motors was presented previously in Chapter 8. To design the software controller, we first determine the time-interval between high-level commands, which is 0.5 second in this example. Next, the command specifications are rewritten as a list of nine operations, each defined by a speed, direction, and duration. The other alternative would have been to divide the 10-second pattern into 20 equally sized 0.5-sec intervals, and specify speed and direction for each interval.

Interval	Yaw				Pitch				Roll		
	1 RPM	300 ms	CW	0 RPM	0 ms	—	0 RPM	0 ms	—	—	—
0.0 to 1.0 sec	1 RPM	300 ms	CW	0 RPM	0 ms	—	0 RPM	0 ms	—	—	—
1.0 to 2.0 sec	1 RPM	300 ms	CCW	0 RPM	0 ms	—	1 RPM	300 ms	CW	—	—
2.0 to 2.5 sec	2 RPM	150 ms	CW	1 RPM	300 ms	CW	1 RPM	300 ms	CW	—	—
2.5 to 3.0 sec	0.5 RPM	600 ms	CCW	1 RPM	300 ms	CW	1 RPM	300 ms	CW	—	—
3.0 to 3.5 sec	0.5 RPM	600 ms	CCW	1 RPM	300 ms	CW	4 RPM	75 ms	CCW	—	—
3.5 to 4.0 sec	0.5 RPM	600 ms	CCW	1 RPM	300 ms	CW	0 RPM	0 ms	—	—	—
4.0 to 4.5 sec	0.5 RPM	600 ms	CCW	2 RPM	150 ms	CCW	0 RPM	0 ms	—	—	—
4.5 to 5.0 sec	0 RPM	0 ms	—	2 RPM	150 ms	CCW	0 RPM	0 ms	—	—	—
5.0 to 10.0 sec	0 RPM	0 ms	—	0 RPM	0 ms	—	0 RPM	0 ms	—	—	—

During the **implementation** and **testing** phases, we build and test the modules. Modularity allows for concurrent development. An important factor when designing data structures is to establish a one-to-one linkage between the foregoing abstract command table and the implementation of the data structure. Program 13.1 shows the implementation of the high-level command table. This one-to-one linkage allows us to separate the testing of the low-level functions (spinning motors) from the high-level control functions.

**Observation:** Open-loop controllers that have simple outputs like this one can also be implemented as finite state machines with no inputs.

**Program 13.1**

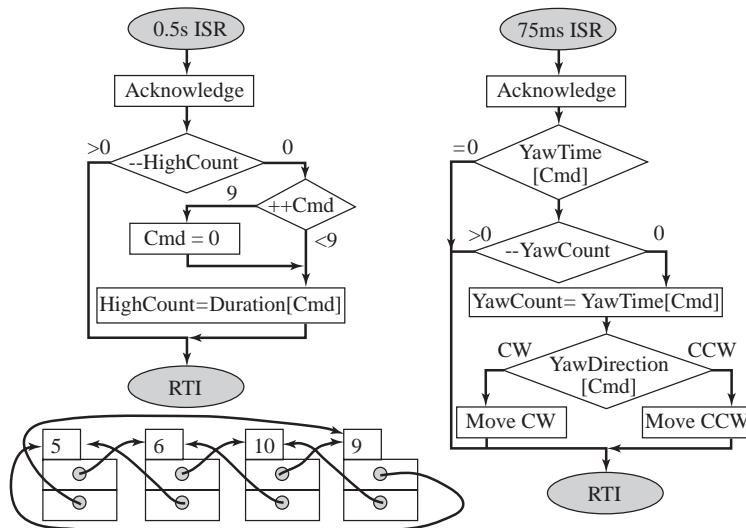
Table structure used to define high-level commands.

```
#define CW 0
#define CCW 1
#define RPM0 0 // means stop
#define RPM0_5 8 // 8*75ms=600ms => 0.5 RPM
#define RPM1 4 // 4*75ms=300ms => 1 RPM
#define RPM2 2 // 2*75ms=150ms => 2 RPM
#define RPM4 1 // 1*75ms=75ms => 4 RPM
const struct Command{
    unsigned short Duration; // time for the command, 0.5s units
    unsigned short YawTime; // determines Yaw speed, 75ms units
    unsigned short YawDirection; // 0 for CW, 1 for CCW
    unsigned short PitchTime; // determines Pitch speed, 75ms units
    unsigned short PitchDirection;
    unsigned short RollTime; // determines Roll speed, 75ms units
    unsigned short RollDirection; // 0 for CW, 1 for CCW
    const struct Command *Next; // circular link
};

typedef const struct Command CommandType;
CommandType Machine[9]={
    // Duration Yaw Pitch Roll
    {2, RPM1, CW, RPM0, 0, RPM0, 0, &Machine[1]},
    {2, RPM1, CCW, RPM0, 0, RPM1, CW, &Machine[2]},
    {1, RPM2, CW, RPM1, CW, RPM1, CW, &Machine[3]},
    {1, RPM0_5, CCW, RPM1, CW, RPM1, CW, &Machine[4]},
    {1, RPM0_5, CCW, RPM1, CW, RPM4, CCW, &Machine[5]},
    {1, RPM0_5, CCW, RPM1, CW, RPM0, 0, &Machine[6]},
    {1, RPM0_5, CCW, RPM2, CCW, RPM0, 0, &Machine[7]},
    {1, RPM0, 0, RPM2, CCW, RPM0, 0, &Machine[8]},
    {10, RPM0, 0, RPM0, 0, RPM0, 0, &Machine[0]}};
CommandType *CmdPt; // Current command
```

Figure 13.6 shows the flowcharts used to implement the robot controller. The high-level controller is implemented with a 0.5-second periodic output compare 0 interrupt. The global HighCount counter implements the high-level delay for each command. The low-level controller is implemented with a 75 ms periodic output compare 1 interrupt. Figure 13.6 just shows the algorithm for the Yaw motor, but the Pitch and Roll functions are identical. The global YawCount PitchCount RollCount counters implement the low-level delays for each motor.

**Figure 13.6**  
Flowchart and linked structure for the control system.



**Checkpoint 13.2:** Explain how this system could have been designed using just one ISR.

Program 13.2 contains a doubly circular linked list used to define the stepper motor sequence. The three global pointers (**YawPt** **PitchPt** **RollPt**) specify the current motor state for each motor. Both high and low nibbles data are included to speed execution (eliminating a shift operation).

### Program 13.2

Linked list structure used to define low-level stepper outputs.

```
const struct StepperState{
    unsigned char OutHigh; // Output for motors with bits 7-4
    unsigned char OutLow; // Output for motors with bits 3-0
    const struct StepperState *Next[2]; // CW or CCW
};

typedef const struct StepperState StepperStateType;
StepperStateType Stepper[4]={
    { 0x50,0x05,{&Stepper[1],&Stepper[3]}},
    { 0x60,0x06,{&Stepper[2],&Stepper[0]}},
    { 0xA0,0x0A,{&Stepper[3],&Stepper[1]}},
    { 0x90,0x09,{&Stepper[0],&Stepper[2]}}
};
StepperStateType *YawPt; // Current Stepper state for Yaw motor
StepperStateType *PitchPt; // Current Stepper state for Pitch motor
StepperStateType *RollPt; // Current Stepper state for Roll motor
```

Program 13.3 contains the initialization and the two interrupt service routines used to execute the controller. The global **Cmd** is an index into the **Machine[]** table indicating which command is active. It is the primary communication between the high-level and low-level controllers.

### Program 13.3

C software to spin three stepper motors.

```
unsigned short HighCount,YawCount,PitchCount,RollCount;
interrupt 8 void TC0handler(void){
    TFLG1 = 0x01; // acknowledge C0F
    TC0 = TC0+62500U; // 0.5sec period
    if(--HighCount==0){ // done?
        CmdPt = CmdPt->Next; // next command
        HighCount = CmdPt->Duration;
    }
}
interrupt 9 void TC1handler(void){
    TFLG1 = 0x02; // acknowledge C1F
    TC1 = TC1+9375; // 75ms period
    if(CmdPt->YawTime){
        if(--YawCount == 0){
            YawCount = CmdPt->YawTime;
            YawPt = YawPt->Next[CmdPt->YawDirection];
            PTT = (PTT&0x0F)+YawPt->OutHigh; //set Port T bits 7-4
        }
    }
    if(CmdPt->PitchTime){
        if(--PitchCount == 0){
            PitchCount = CmdPt->PitchTime;
            PitchPt = PitchPt->Next[CmdPt->PitchDirection];
            PTT = (PTT&0xF0)+PitchPt->OutLow; //set Port T bits 3-0
        }
    }
}
```

*continued on p. 655*

*continued from p. 654*

```

if(CmdPt->RollTime){
    if(--RollCount == 0){
        RollCount = CmdPt->RollTime;
        RollPt = RollPt->Next[CmdPt->RollDirection];
        PTM = (PTM&0x30)+RollPt->OutLow; //set Port M bits 3-0
    }
}
void Motor_Init(void){ // assumes 4MHz E clock
    PTT = 0x55;           // start at 5, first output moves motor
    PTM = 0x05;
    DDRT = 0xFF;          // PT7-4 is yaw, PT3-0 is pitch
    DDRM |= 0x0F;          // PM3-0 is roll
    TIOS |= 0x03;          // activate TC0, TC1 as output compares
    TSCR1 = 0x80;          // Enable TCNT, 8us
    TSCR2 = 0x05;          // divide by 32 TCNT prescale, TOI disarm
    PACTL = 0;              // timer prescale used for TCNT
    CmdPt = &Machine[8]; // first command will be 0
    YawPt = PitchPt = RollPt = &Stepper[0];
    HighCount = YawCount = PitchCount = RollCount = 1;
    TIE |= 0x03;           // arm OC1, OC0
    TC0 = TCNT+50;         // first interrupt right away
    TC1 = TC0+50;          // interrupts after TC0
    asm cli
}

```

**Checkpoint 13.3:** Cmd is a shared global variable with read and write activity. Does this activity cause a critical section?

### 13.3 Simple Closed-Loop Control Systems

A *bang-bang controller* uses a binary actuator, meaning the microcontroller output can be on or off. Other names for this controller are *binary controller*, *two-position controller*, and *on/off controller*. It is a closed-loop control system, because there is a sensor that measures the status of the system. This signal is called the measurand or state variable. Assume when the actuator is on, the measurand increases, and when the actuator is off, the measurand decreases. There is a desired point for the measurand. The bang-bang controller is simple. If the measurand is too small, the actuator is turned on, and if the measurand is too large, the actuator is turned off.

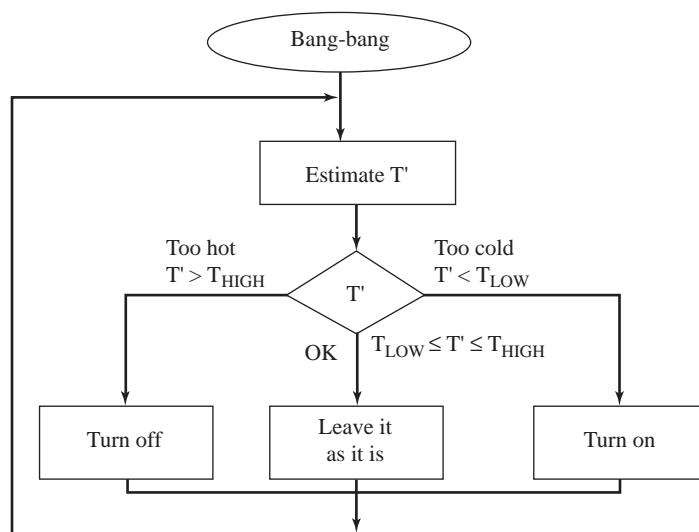
**Example 13.3** Design a bang-bang temperature controller.

**Solution** This digital control system applies heat to the room to maintain the temperature as close to  $T^*$  (labeled  $T_{star}$  in the software) as possible (Figure 13.7). This is a closed-loop control system because the control signals (heat) depend on the state variables (temperature). In this application, the actuator has only two states: *on*, which warms up the room, and *off*, which does not apply heat. For this application to function properly, there must be a passive heat loss that lowers the room temperature when the heater is turned off (Figure 13.8).

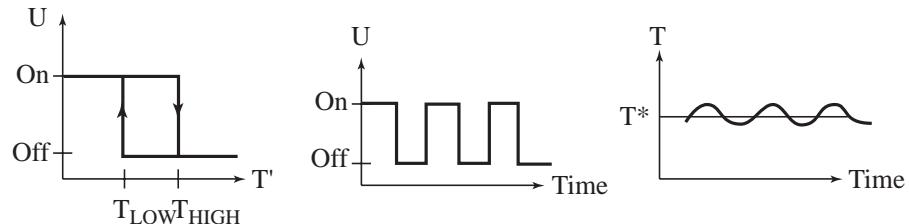
A bang-bang controller turns on the power if the temperature is too low and turns off the power if the temperature is too high. To implement hysteresis, we need two set point temperatures,  $T_{HIGH}$  and  $T_{LOW}$ . The controller turns on the power (activates relay) if the temperature goes below  $T_{LOW}$  and turns off the power (deactivates relay) if the temperature goes above  $T_{HIGH}$ . The difference  $T_{HIGH} - T_{LOW}$  is called hysteresis. The hysteresis extends the life of the relay by reducing the number of times the relay opens and closes.

**Figure 13.7**

Flowchart of a bang-bang temperature controller.

**Figure 13.8**

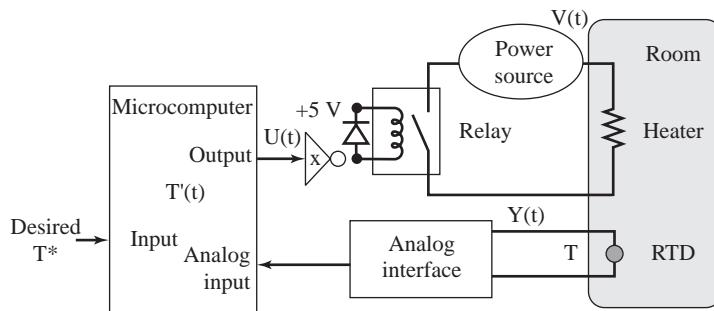
Algorithm and response of bang-bang temperature controller.



In this implementation, the RTD is made from a thin platinum wire and converts the room temperature into a resistance. The analog circuit matches the full-scale range of the room temperature to the 0 to +5 V ADC input voltage range. The software converts the ADC result into estimated temperature,  $T'$  (labeled  $T$  in the software) (Figure 13.9).

**Figure 13.9**

Interface of a simple bang-bang temperature controller.



Assume that the function `SE()` converts the ADC sample into the estimated temperature as a binary fixed-point number with a resolution of  $0.5^{\circ}\text{C}$ . The C code in Program 13.4 uses a periodic interrupt so that the bang-bang controller runs in the background. The interrupt `Period` is selected to be about 10 times faster than the time constant of the physical plant. The temperature variables  $T_{low}$ ,  $T_{high}$ , and  $T$  could be in any format, as long as the three formats are the same.

**Program 13.4**

Bang-bang temperature control software.

```

short Tlow,Thigh;      // controller set points, 0.5 C
// PTM bit 0 turns power on/off
interrupt 13 void TC5handler(void){
short T=SE(ADC_In(0)); // estimated temperature, 0.5 C
    if(T < Tlow){
        PTM |= 0x01;      // too cold so on
    }
    else if (T > Thigh){
        PTM &= ~0x01;     // too hot so off
    }
                    // leave as is if Tlow<T<Thigh
    TC5 = TC5+Period; // periodic rate
    TFLG1 = 0x20;      // acknowledge C5F
}

```

**Checkpoint 13.4:** What happens if  $T_{low}$  and  $T_{high}$  are too close together?

**Checkpoint 13.5:** What happens if  $T_{low}$  and  $T_{high}$  are too far apart?

**Observation:** Bang-bang control works well with a physical plant that has a very slow response.

**Observation:** The DS1620 implements this bang-bang controller.

An *incremental controller* uses an actuator with a finite number of discrete output states. For example, the actuator might have 256 possibilities from 0, 1, 2, ..., 255. It is a closed-loop control system, because there is a sensor that measures the state variable. Assume when the actuator increases, the measurand increases, and when the actuator decreases, the measurand decreases. There is a desired point for the measurand. The incremental controller is simple. If the measurand is too small, the actuator is increased, and if the measurand is too large, the actuator is decreased. It is important to choose the rate to run the controller properly. A good rule of thumb is to run the controller about 10 times faster than the time constant of the plant. The control system should make sure the actuator signal remains in the appropriate range. For example, you do not want to increment an actuator output of 255 and get 0! The incremental controller is usually slow, but it has good accuracy and is very stable.

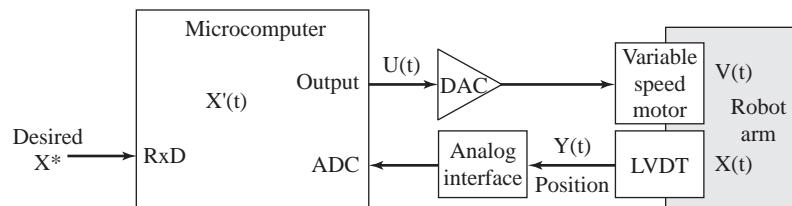
**Example 13.4** Design an incremental position controller.

**Solution** The objective of this *incremental control* system is to control the position of the robot arm,  $X$  (Figure 13.10). The control signals (power) are dependent on the state variables (position). The LVDT, which was described in Chapter 12, is used to sense the position of the robot arm.

An incremental control algorithm simply adds or subtracts a constant from  $U$  depending on the sign of the error. In other words, if  $X$  is too small, then  $U$  is incremented, and if  $X$  is too large, then  $U$  is decremented. It is important to choose the proper rate at which

**Figure 13.10**

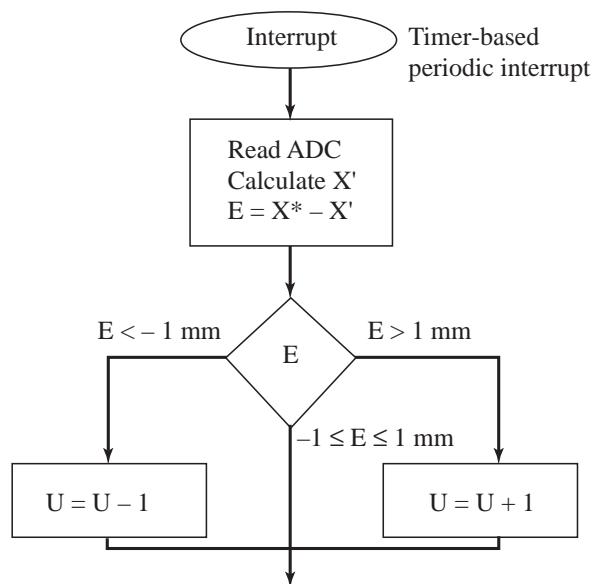
Interface of a position controller.



the incremental control software is executed. If it is executed too many times per second, then the actuator will saturate, resulting in a bang-bang system. If it is not executed often enough, then the system will not respond quickly to changes in the physical plant or changes in  $X^*$ . In this incremental controller we add or subtract “1” from the actuator, but a value larger than “1” would have a faster response at the expense of introducing oscillations (Figure 13.11).

**Figure 13.11**

Flowchart of a position controller implemented using incremental control.



**Common error:** An error will occur if the software does not check for overflow and underflow after  $U$  is changed.

**Observation:** If the incremental control algorithm is executed too frequently, then the resulting system behaves like a simple bang-bang controller.

**Observation:** Many control systems operate well when the control equations are executed about ten times faster than the step response time of the physical plant.

Assume that the function `SE()` converts the ADC sample into position as a signed decimal fixed-point number with a resolution of 0.1 mm. The 9S12 C code in Program 13.5 uses a

### Program 13.5

Incremental position control software.

```

short X,Xstar,E; // position, fixed-point in 0.1mm
// PTT is connected to the 8-bit DAC
interrupt 13 void TC5handler(void){ short U;
    X = SE(ADC_In(0)); // estimated (0.1 mm)
    E = Xstar-X; // error (0.1 mm)
    U = (short)PTT; // promote to 16 bits
    if(E < -10) U--; // decrease if less than -1mm
    else if(E > 10) U++; // increase if greater than +1mm
    // leave as is if -1mm<E<1mm
    if(U<0) U=0; // underflow
    if(U>255) U=255; // overflow
    PTT = U; // output to actuator
    TC5 = TC5+Period; // periodic rate
    TFLG1 = 0x20; // acknowledge C5F
}
  
```

periodic interrupt so that the incremental controller runs in the background. The interrupt Period is selected to be about ten times faster than the time constant of the physical plant. Even though the position variables  $x$  and  $x_{star}$  may be unsigned, the error calculation  $E$  will be signed.

**Checkpoint 13.6:** In what ways would the controller behave differently if  $-10$  and  $+10$  were to be changed to  $0$ ?

**Checkpoint 13.7:** What happens if **Period** is too small (i.e., it executes too frequently)?

**Observation:** It is a good debugging strategy to observe the assembly listing generated by the compiler when performing calculations on variables of mixed types (signed/unsigned, char/short).

**Observation:** Incremental control will work moderately well (accurate and stable) for an extremely wide range of applications. Its only shortcoming is that the controller response time can be quite slow.

## 13.4 PID Controllers

- 13.4.1 General Approach to a PID Controller** The simple controllers presented in the last section are easy to implement, but they will have either large errors or very slow response times. To make a faster and more accurate system, we can use linear control theory to develop the digital controller. There are three components of a *PID controller*.

$$U(t) = K_p E(t) + K_I \int_0^t E(\tau) d\tau + K_D \frac{dE(t)}{dt}$$

The error,  $E(t)$ , is defined as the present set-point,  $X^*(t)$ , minus the measured value of the controlled variable,  $X'(t)$ .

$$E(t) = X^*(t) - X'(t)$$

The PID controller calculates its output by summing three terms. The first term is proportional to the error. The second is proportional to the integral of the error over time, and the third is proportional to the rate of change (first derivative) of the error term. The values of  $K_p$ ,  $K_I$ , and  $K_D$  are design parameters and must be properly chosen for the control system to operate properly. The proportional term of the PID equation contributes an amount to the control output that is directly proportional to the current process error. The gain term  $K_p$  adjusts exactly how much the control output response should change in response to a given error level. The larger the value of  $K_p$ , the greater the system reaction to differences between the set-point and the actual state variable. However, if  $K_p$  is too large, the response may exhibit an undesirable degree of oscillation or even become unstable. On the other hand, if  $K_p$  is too small, the system will be slow or unresponsive. An inherent disadvantage of proportional-only control is its inability to eliminate the steady-state errors (offsets) that occur after a set-point change or a sustained load disturbance.

The integral term converts the first-order proportional controller into a second-order system capable of tracking process disturbances. It adds to the controller output a factor that takes corrective action for any changes in the load level of the system. This integral term is scaled to the sum of all previous process errors in the system. As long as there is a process error, the integral term will add more amplitude to the controller output until the sum of all previous errors is zero. Theoretically, as long as the sign of  $K_I$  is correct, any value of  $K_I$  will eliminate offset errors. But, for extremely small values of  $K_I$ , the controlled variables will return to the setpoint very slowly after a load change or

setpoint change occurs. On the other hand, if  $K_I$  is too large, it tends to produce oscillatory response of the controlled process and reduces system stability. The undesirable effects of too much integral action can be avoided by proper tuning (adjusting) of the controller or by including derivative action which that to counteract the destabilizing effects.

**Checkpoint 13.8:** What happens in a PID controller if the sign of  $K_I$  is incorrect?

The derivative action of a PID controller adds a term to the controller output scaled to the slope (rate of change) of the error term. The derivative term “anticipates” the error, providing a greater control response when the error term is changing in the wrong direction and a dampening response when the error term is changing in the correct direction. The derivative term tends to improve the dynamic response of the controlled variable by decreasing the process setting time, the time it takes the process to reach steady state. But if the process measurement is noisy, that is, if it contains high-frequency random fluctuations, then the derivative of the measured (controlled) variable will change wildly, and derivative action will amplify the noise unless the measurement is filtered.

We also can use just some of the terms. For example, a proportional/integral controller drops the derivative term. We will analyze the digital control system in the frequency domain. Let  $X(s)$  be the Laplace transform of the state variable  $x(t)$ , let  $X^*(s)$  be the Laplace transform of the desired state variable  $x^*(t)$ , and let  $E(s)$  be the Laplace transform of the error. Because the system is linear

$$E(s) = X^*(s) - X(s)$$

Let  $G(s)$  be the transfer equation of the PID linear controller. PID controllers are unique in this aspect. In other words, we cannot write a transfer equation for a bang-bang, incremental, or fuzzy logic controller, but for a PID controller we have

$$G(s) = c \left( k_P + k_D s + \frac{k_I}{s} \right)$$

Let  $H(s)$  be the transfer equation of the physical plant. If we assume the physical plant (e.g., a DC motor) has a simple single-pole behavior, then we can specify its response in the frequency domain with two parameters.  $m$  is the DC gain and  $\tau$  is its time constant. The transfer function of a single-pole plant is

$$H(s) = \frac{m}{1 + \tau \cdot s}$$

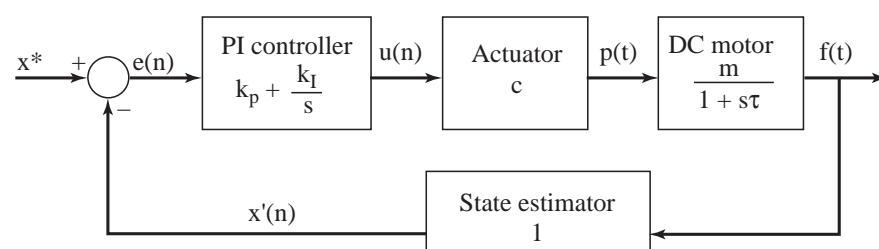
The overall gain of the control system (Figure 13.12) is

$$\frac{X(s)}{X^*(s)} = \frac{G(s)H(s)}{1 + G(s)H(s)}$$

Theoretically, we can choose controller constants  $k_P$ ,  $k_I$ , and  $k_D$  to create the desired controller response. Unfortunately, it can be difficult to estimate  $c$ ,  $m$ , and  $\tau$ . If the load to the motor varies, then  $m$  and  $\tau$  will change.

**Figure 13.12**

Block diagram of a linear control system in the frequency domain.



To simplify implementation of the PID controller, we break the controller equation into separate proportional, integral, and derivative terms. That is, let

$$U(t) = P(t) + I(t) + D(t)$$

where  $U(t)$  is the actuator output, and  $P(t)$ ,  $I(t)$ , and  $D(t)$  are the proportional, integral, and derivative components, respectively. The proportional term makes the actuator output linearly related to the error. Using a proportional term creates a control system that applies more energy to the plant when the error is large.

$$P(t) = K_p \cdot E(t)$$

To implement the proportional term we simply convert the above equation into discrete time.

$$P(n) = K_p \cdot E(n)$$

where the index  $n$  refers to the discrete time input of  $E(n)$  and output of  $P(n)$ .

**Observation:** To develop digital signal-processing equations, it is imperative that the control system be executed on a regular and periodic rate.

**Common error:** If the sampling rate varies, then controller errors will occur.

The integral term makes the actuator output related to the integral of the error. Using an integral term often will improve the steady-state error of the control system. If a small error accumulates for a long time, this term can get large. Some control systems put upper and lower bounds on this term, called *anti-reset-windup*, to prevent it from dominating the other terms:

$$I(t) = K_I \cdot \int_0^t E(\tau) d\tau$$

The implementation of the integral term requires the use of a discrete integral or sum. If  $I(n)$  is the current control output, and  $I(n-1)$  is the previous calculation, the integral term is simply

$$I(n) = K_I \cdot \sum_{i=1}^n (E(i) \cdot \Delta t) = I(n-1) + K_I \cdot E(n) \cdot \Delta t$$

where  $\Delta t$  is the sampling rate of  $E(n)$ .

The derivative term makes the actuator output related to the derivative of the error. This term is usually combined with either the proportional and/or integral term to improve the transient response of the control system. The proper value of  $K_D$  will provide for a quick response to changes in either the set point or loads on the physical plant. An incorrect value may create an overdamped (very slow response) or an underdamped (unstable oscillations) response.

$$D(t) = K_D \cdot \frac{dE}{dt}$$

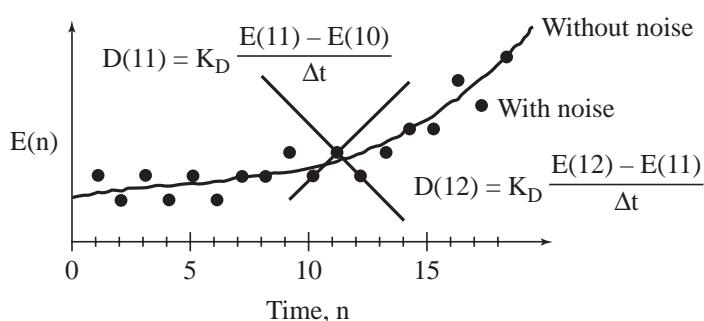
There are a couple of ways to implement the discrete time derivative. The simple approach is

$$D(n) = K_D \cdot \frac{E(n) - E(n-1)}{\Delta t}$$

In practice, this first-order equation is quite susceptible to noise. Figure 13.13 shows a sequence of  $E(n)$  with some added noise. Notice that huge errors occur when the above equation is used to calculate the derivative.

**Figure 13.13**

Illustration of the effect noise plays on the calculation of discrete derivative.



In most practical control systems, the derivative is calculated using the average of two derivatives calculated across different time spans. For example,

$$D(n) = K_D \cdot \left[ \frac{1}{2} \frac{E(n) - E(n-3)}{3\Delta t} + \frac{1}{2} \frac{E(n-1) - E(n-2)}{\Delta t} \right]$$

which simplifies to

$$D(n) = K_D \cdot \frac{E(n) + 3E(n-1) - 3E(n-2) - E(n-3)}{6\Delta t}$$

**Checkpoint 13.9:** How is the continuous integral related to the discrete integral?

**Checkpoint 13.10:** How is the continuous derivative related to the discrete derivative?

### 13.4.2 Design Process for a PID Controller

The first design step is the analysis phase, where we determine specifications such as range, accuracy, stability, and response time for our proposed control system. A data acquisition system will be used to estimate the state variables. Thus, its range, accuracy, and response time must be better than the desired specifications of the control system. We can use time-based techniques from Chapter 6, or develop an ADC-based state estimator using the techniques of Chapters 11 and 12. In addition, we need to design an actuator to manipulate the state variables. It too must have a range and response time better than the controller specifications. The **actuator resolution** is defined as the smallest reliable change in output. For example, a 100 Hz PWM output generated by a 1 μsec clock has 10,000 different outputs. For this actuator, the actuator resolution is MaxPower/10000. We wish to relate the actuator performance to the overall objective of controller accuracy. Thus, we need to map the effect on the state variable caused by a change in actuator output equal to 1 resolution. This change in state variable should be less than or equal to the desired controller accuracy.

After the state estimator and actuator are implemented, the controller settings ( $K_P$ ,  $K_I$ , and  $K_D$ ) must be adjusted so that the system performance is satisfactory. This activity is referred to as **controller tuning** or **field tuning**. If you perform controller tuning by guessing the initial setting and then adjusting them by trial and error, it can be tedious and time consuming. Thus, it is desirable to have good initial estimates of controller settings. A good first setting may be available from experience with similar control loops. Alternatively, initial estimates of controller settings can be derived from the transient response of the physical plant. A simple open-loop method, called the **process reaction curve approach**, was first proposed by Ziegler/Nichols and Cohen/Coon in 1953. In this discussion, the term “process” as defined by Ziegler/Nichols means the same thing as the “physical plant” described earlier in this chapter. This open-loop method requires only that a single step input be imposed on the process. The process reaction method is based on a single experimental

test that is made with the controller in the manual mode. A small step change,  $\Delta U$ , in the actuator output is introduced and the measured process response is recorded, as shown in Figure 13.14. To obtain parameters of the process, a tangent is drawn to the process reaction curve at its point of maximum slope (at the inflection point). This slope is  $R$ , which is called the **process reaction rate**. The intersection of this tangent line with the original base line gives an indication of  $L$ , the process lag.  $L$  is really a measure of equivalent dead time for the process. If the tangent drawn at the inflection point is extrapolated to a vertical axis drawn at the time when the step was imposed, the amount by which this value is below the horizontal base line will be represented by the product  $L \cdot R$ .  $\Delta T$  is the time step for the digital controller. It is recommended that P and PI controllers be run with  $\Delta T = 0.1L$ , and a PID controller at a rate 20 times faster ( $\Delta T = 0.05L$ ). Using these parameters, Ziegler and Nichol proposed initial controller settings as

Proportional Controller

$$K_P = \Delta U / (L \cdot R)$$

Proportional-Integral Controller

$$K_P = 0.9 \Delta U / (L \cdot R)$$

$$K_I = K_P / (3.33L)$$

Proportional-Integral-Derivative Controller

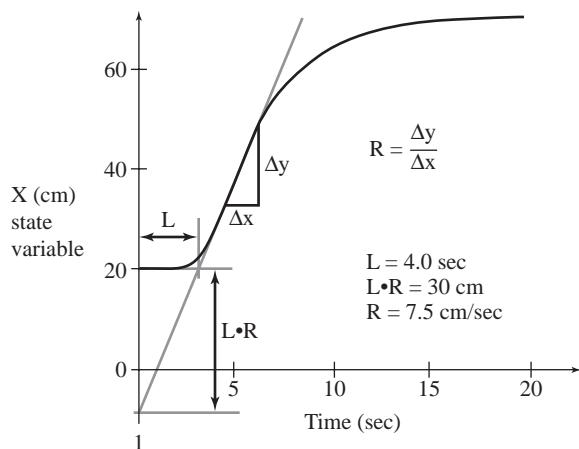
$$K_P = 1.2 \Delta U / (L \cdot R)$$

$$K_I = 0.5 K_P / L$$

$$K_D = 0.5 K_P L$$

**Checkpoint 13.11:** Are the Ziegler/Nichol equations consistent from a dimensional-analysis perspective? In other words, are the units correct?

**Figure 13.14**  
A process reaction curve used to determine controller settings.



The **response time** is the delay after  $X^*$  is changed for the system to reach a new constant state. **Steady-state controller accuracy** is defined as the average difference between  $X^*$  and  $X'$ . **Overshoot** is defined as the maximum positive error that occurs when  $X^*$  is increased. Similarly, **undershoot** is defined as the maximum negative error that occurs when  $X^*$  is decreased. During the testing phase, it is appropriate to add minimally intrusive debugging software that specifically measures performance parameters, such as response time, accuracy, overshoot, and undershoot. In addition, we can add instruments that allow

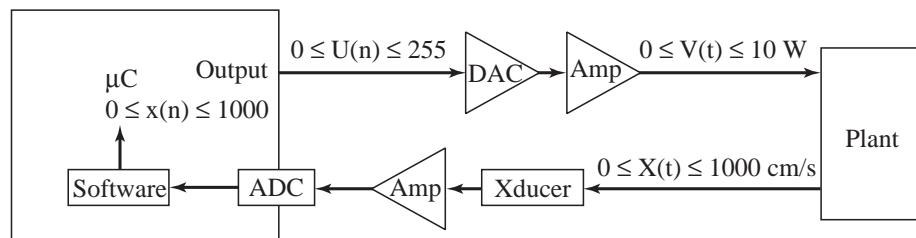
us to observe the individual  $P(t)$ ,  $I(t)$ , and  $D(t)$  components of the PID equation and their relation to controller error  $E(t)$ .

Once the initial parameters have been selected, a simple empirical method can be used to fine-tune the controller. This empirical approach starts with proportional term  $K_p$ . As the proportional term is adjusted up or down, evaluate the quickness and smoothness of the controller response to changes in setpoint and to changes in the load.  $K_p$  is too big if the actuator saturates at both the maximum and the minimum after  $X^*$  is changed. The next step is to adjust the integral term ( $K_I$ ) a little at a time to improve the steady-state controller accuracy without adversely affecting the response time. Don't change both  $K_p$  and  $K_I$  at once. Rather, you should vary them one at a time. If the response time, overshoot, undershoot, and accuracy are within acceptable limits, a PI controller is adequate. On the other hand, if accuracy and response are OK but overshoot and undershoot are unacceptable, adjust the derivative term ( $K_D$ ) to reduce the overshoots and undershoots.

### Example 13.5 Design a PID velocity controller.

**Solution** The objective of this example is to develop the fixed-point equations that implement a PID velocity controller (Figure 13.15).

**Figure 13.15**  
Interface of a PID velocity controller.



The maximum output of 255 maps linearly into a maximum applied power of 10 W to the physical plant. Similarly, a maximum speed of 1000 cm/s maps linearly into a maximum ADC result of 255. Also,  $U = 0$  converts into  $V = 0$ , and  $X = 0$  gives an ADC conversion of 0. Let  $X_{\text{star}}$  be a 16-bit unsigned integer containing the desired speed or set point in centimeters per second. Let current error be defined as

$$e(t) = X_{\text{star}} - X(t)$$

in centimeters per second. We will implement the following PID control equation

$$V(t) = 0.1e(t) + 0.5 \int_0^{\tau} e(\tau) d\tau + 0.005 \frac{de(t)}{dt}$$

where  $e(\tau)$  is in centimeters per second and  $V(\tau)$  is in watts. To simplify the problem, we will break the controller into separate proportional, integral, and derivative terms. That is, let

$$U(n) = P(n) + I(n) + D(n)$$

where  $U(n)$  is the next DAC output and  $P(n)$ ,  $I(n)$ , and  $D(n)$  are the proportional, integral, and derivative components, respectively. Let  $x(n)$  be the current 8-bit unsigned ADC sample.  $x(n)$  will be the 16-bit estimated current speed in centimeters

per second.  $x(n)$  ranges from 0 to 1000 cm/s. The fixed-point calculation that converts data into the estimated speed  $x(n)$  is

$$x(n) = \frac{1000 \cdot \text{data}}{256} = \frac{125 \cdot \text{data}}{32}$$

We define  $e(n) = X_{\text{star}} - x(n)$  as the 16-bit signed current error in centimeters per second. Let  $e(n-1)$  be the previous calculation sampled at 100 Hz. Next, we develop fixed-point equations for  $P(n)$ ,  $I(n)$ , and  $D(n)$  in terms of the current and previous  $e(n)$  calculations. The term  $I(n-1)$  refers to the previous calculation of the integral.

$$e(n) = X_{\text{star}} - x(n)$$

For the proportional term we have

$$V(t) = 0.1 \cdot e(t)$$

Therefore

$$\frac{10 \cdot P(n)}{256} = 0.1 \cdot e(n)$$

Rearranging we get

$$P(n) = \frac{256 \cdot e(n)}{100}$$

For the integral term we have

$$V(t) = 0.5 \int_0^t e(\tau) d\tau = 0.5 \Delta t \sum_{i=0}^n e(n) = 0.005 \sum_{i=0}^n e(n)$$

Therefore

$$\frac{10 \cdot I(n)}{256} = 0.005 \cdot e(n)$$

Rearranging we get

$$I(n) = \frac{16 \cdot e(n)}{125} + I(n-1)$$

For the derivative term we have

$$V(t) = 0.005 \frac{de(t)}{dt} = 0.005 \frac{e(n) - e(n-1)}{\Delta t}$$

Therefore

$$\frac{10 \cdot D(n)}{256} = 0.5 \cdot e(n) - e(n-1)$$

Rearranging, we get

$$D(n) = \frac{64 \cdot e(n) - e(n-1)}{5}$$


---



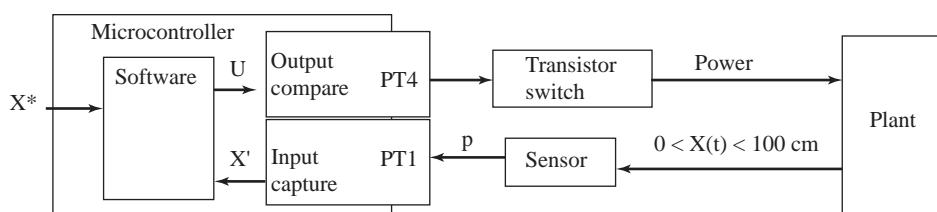
---

### Example 13.6 Design a PI position controller using a PWM actuator.

**Solution** We will design an MC9S12C32 microcomputer-based proportional-integral control system. The overall objective is to control the position of an object with an accuracy of 0.1 cm and a range of 0 to 100 cm, as shown in Figure 13.16. Let  $X^*$  be the desired state variable. In this example,  $X^*$  is a decimal fixed-point number and is set by the main program. Let  $X'$  be the estimated state variable that comes from the **state estimator**, which encodes the current position as the period of a squarewave, interfaced to **PT1**. The

**Figure 13.16**

PI controller using period measurement and pulse-width modulation.



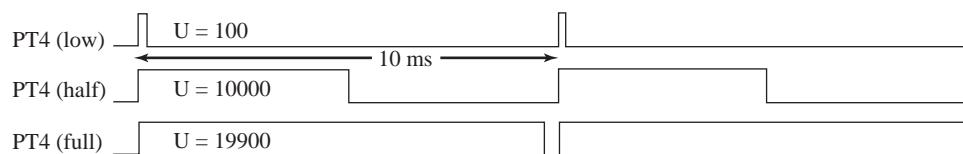
period output of the sensor is linearly related to the position  $\mathbf{X}$  with a fixed offset. The accuracy of the state estimator needs to match the 0.1 cm specification of the controller. If  $p$  is the measured period in 0.1 ms and  $\mathbf{X}'$  is the estimated position in 0.1 cm, the state estimator measures the period and calculates  $\mathbf{X}'$ .

$$\mathbf{X}' = p - 100$$

Let  $\mathbf{U}$  be the actuator control variable ( $100 \leq \mathbf{U} \leq 19900$ ). This system uses **pulse-width modulation** with a 100 Hz squarewave that applies energy to the physical plant, as shown in Figure 13.17.  $\mathbf{U}$  will be the number of E clock cycles (out of 20000) when the **PT4** output is high. There is an external gravitational force pulling down on the object. **PT4** is an output from the computer and an input to the actuator, creating an upward force.

**Figure 13.17**

Pulse width modulated actuator signals.



The process reaction curve shown previously in Figure 13.14 was measured for this system after the actuator was changed from 250 to 2000; thus,  $\Delta\mathbf{U}$  is 1750 (units of E clock cycles). From Figure 13.4, the lag  $\mathbf{L}$  is 4.0 sec and the process reaction rate  $\mathbf{R}$  is 7.5 cm/sec. The controller rate is selected to be about ten times faster than the lag  $\mathbf{L}$ , so  $\Delta T = 0.4$  sec. In this way, the controller runs at a rate faster than the physical plant. We calculate the PI controller settings using the Ziegler/Nichol equations.

$$\mathbf{K}_P = 0.9 \Delta\mathbf{U}/(\mathbf{L} \cdot \mathbf{R}) = 0.9 \cdot 1750 / (4.0 \cdot 7.5) = 52.5 \text{ cycles/cm}$$

$$\mathbf{K}_I = \mathbf{K}_P / (3.33\mathbf{L}) = 52.5 / (3.33 \cdot 4.0) = 3.94144 \text{ cycles/cm/sec}$$

We will execute the proportional control equation once every 0.4 second.  $\mathbf{X}^*$  and  $\mathbf{X}'$  are decimal fixed-point numbers with a resolution of 0.1 cm. The constant 52.5 is expressed as 105/2. The extra divide by 10 handles the decimal fixed-point representation of  $\mathbf{X}^*$  and  $\mathbf{X}'$ .

$$P(n) = K_P \cdot (X^* - X')/10 = 105 \cdot (X^* - X')/20$$

We will also execute the integral control equation once every 0.4 second. Binary fixed-point is used to approximate 1.57658 as 101/64.

$$I(n) = I(n-1) + K_I \cdot (X^* - X') \cdot \Delta T/10 = I(n-1) + 3.94144 \cdot (X^* - X') \cdot 0.4/10 = I(n-1) + 101 \cdot (X^* - X')/640$$

Program 13.6 includes the OC5 interrupt service handler, which generates the 10 kHz real time clock. The OC5 handler will establish the current **Time** in 0.1 ms. After 4000 OC5 interrupts (0.4 second), the control algorithm is implemented.

**Program 13.6**

Integral position control software.

```

unsigned short Time; // Time in 0.1 msec
short X;           // Estimated position in 0.1 cm, 0 to 1000
short Xstar;        // Desired pos in 0.1 cm, 0 to 1000
short E;           // Position error in 0.1 cm, -1000 to +1000
short U,I,P;       // Actuator duty cycle, 100 to 19900 cycles
unsigned short Cnt; // once a sec
unsigned short Told; // used to measure period
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20;      // ack C5F
    TC5 = TC5+200;    // every 0.1 ms
    Time++;           // used to measure period
    if((Cnt++)==4000){ // every 0.4 sec
        Cnt = 0;        // 0<X<100, 0<Xstar<100, 100<U<19900
        E = Xstar-X;
        P = (105*E)/20;
        I = I+(101*E)/640;
        if(I < -500) I=-500; // anti-reset windup
        if(I > 4000) I=4000;
        U = P+I;          // PI controller has two parts
        if(U < 100) U=100; // Constrain actuator output
        if(U>19900) U=19900;
    }
}
}

```

**Checkpoint 13.12:** What is the output **U** of the controller if the position **X** is much greater than the setpoint **X\***? In this situation, what does the object do?

Program 13.7 uses OC4 to generate a 100 Hz real-time clock, creating the 100 Hz variable duty-cycle squarewave connected to the actuator. The OC4 output will be high for **U** cycles and low for  $20,000 - U$  cycles. Program 13.8 uses IC1 to interrupt on each rise of the sensor squarewave, measuring the period of the sensor squarewave and estimating the current position. Program 13.9 is the ritual that initializes the global variables and arms the three interrupts. Once initialized, the controller runs in the background. The main program (not shown) is responsible for calling the ritual, and establishing the controller setpoint, **Xstar** with units of 0.1 cm.

**Program 13.7**

PWM actuator control software.

```

void interrupt 12 TC4handler(void){
    TFLG1 = 0x10;      // acknowledge C4F
    if(PTT&0x10){
        TC4 = TC4+U;    // PT4 is 1, High for the next U cycles
    }
    else{
        TC4 = TC4+20000-U; // PT4 is 1, Low for the next 20000-U cycles
    }
}

```

**Program 13.8**

Sensor measurement software.

```

// Time is incremented every 0.1 ms, by OC5
// This handler is executed on rise of PT1
void interrupt 9 TC1Handler(void){unsigned short p;
    TFLG1 = 0x02; // Acknowledge C1F
    p = Time-Told; // period in msec
    X = p-100;     // estimated position (0.1 cm)
    Told = Time;
}

```

**Program 13.9**

Initialization software.

```

void Init(void){ // PT5 output for debugging
    asm sei        // make atomic
    TIOS = 0x30;   // output compare OC5, OC4
    DDRT &= ~0x02; // PT1 is input
    DDRT |= 0x30; // PT5, PT4 are output
    TSCR1 = 0x80; // enable
    TSCR2 = 0x01; // 500 ns clock
    TCTL1 = (TCTL1&0xF0)|0x05; // toggle PT5, PT4
    TCTL4 = 0x08; // Input capture on rise of IC1
    TIE = 0x32;   // Arm OC5F+OC4F+IC1F
    U = 100;      // Initial U, low power
    Time = Told = 0; Cnt = 0;
    TFLG1 = 0x32; // clear flags
    TC5 = TCNT+50; // First OC5 in 25us
    asm cli
}

```

**Observation:** PID control will work extremely well (fast, accurate, and stable) if the physical plant can be described with a set of linear differential equations.

## 13.5 Fuzzy Logic Control

There are a number of reasons to consider the fuzzy logic approach to control. It requires less mathematics than PID systems. It will also require less memory and execute faster. In other words, an 8-bit fuzzy system may perform as well (same steady-state error and response time) as a 16-bit PID system. When complete knowledge about the physical plant is known, then a good PID controller can be developed. Since the fuzzy logic control is more robust (still works even if the parameter constants are not optimal), then the fuzzy logic approach can be used when complete knowledge about the plant is not known or can change dynamically. Choosing the proper PID parameters requires expert knowledge about the plant. The fuzzy logic approach is more intuitive, following more closely the way a human would control the system. It is easy to modify an existing fuzzy control system into a new problem. So if the framework exists, rapid prototyping is possible.

Fuzzy logic was conceived in the mid-1960s by Lotfi Zadeh while at the University of California at Berkeley. However, the first commercial application didn't come until 1987, when Matsushita Industrial Electric used it to control the temperature in a shower head. Named after the nineteenth-century mathematician George Boole, Boolean logic is an algebra in which values are either true or false. This algebra includes the operations of AND OR and NOT. Fuzzy logic is also an algebra, but conditions may exist in the continuum between true and false. While Boolean logic defines two states, 8-bit fuzzy logic consists of 256 states all the way from "not at all" (0) to "definitely true" (255). For example, "128" means halfway between true and false. The fuzzy logic algebra also includes the operations of AND, OR, and NOT. A **fuzzy membership set**, a **fuzzy variable**, and a **fuzzy set** all refer to the same entity, which is a software variable describing the level of correctness for a condition within fuzzy logic. If we have a fuzzy membership set for the condition "hungry," then as the value of hungry moves from 0 to 255, the condition "hungry" becomes more and more true.

....0.....32.....64.....96.....128.....160.....192.....224.....255  
Not at all...a little bit...somewhat...mostly...pretty much...definitely

The design process for a fuzzy logic controller solves the following eight components. These components are listed in the order we would draw a data flow graph, starting with the state variables on the left, progressing through the controller, and ending with the actuator output on the right.

- The **Physical plant** has *real state variables*.
- The **Data Acquisition System** monitors these signals, creating the *estimated state variables*.
- The **Preprocessor** may calculate relevant parameters called *crisp inputs*.
- **Fuzzification** will convert crisp inputs into *input fuzzy membership sets*.
- The **Fuzzy Logic** is a set of rules that calculate *output fuzzy membership sets*.
- **Defuzzification** converts output sets into *crisp outputs*.
- The **Postprocessor** modifies crisp outputs into a more convenient format.
- The **Actuator System** affects the physical plant based on these outputs.

We will work through the concepts of fuzzy logic by considering examples of how we as humans control things like driving a car at a constant speed. During the initial stages of the design, we study the **physical plant** and decide which state variables to consider. For example, if we wish to control speed, then speed is obviously a state variable, but it might be also useful to know other forces acting on the object such as gravity (e.g., going up and down hills), wind speed, and friction (e.g., rain and snow on the roadway). The purpose of the **data acquisition system** is to accurately measure the state variables. It is at this stage that the system converts physical signals into digital numbers to be processed by the software controller. We have seen two basic approaches in this book for this conversion: the measurement of period/frequency using input capture and the analog-to-digital conversion using an ADC. The **preprocessor** calculates **crisp inputs**, which are variables describing the input parameters in our software having units (like miles/hr). For example, if we measured speed, then some crisp inputs we might calculate would include speed error and acceleration. Just as with the PID controller, the accuracy of the data acquisition system must be better than the desired accuracy of the control system as a whole.

The next stage of the design is to consider the **actuator** and postprocessor. It is critical for the actuator to be able to induce forces on the physical plant in a precise and fast manner. The step response of the actuator itself (time from software command to the application of force on the plant) must be faster than the step response of the plant (time from the application of force to the change in state variable). Consider the case where we wish to control the temperature of a pot of water using a kitchen stove. The speed of this actuator is the time between turning the stove on and the time when heat is applied to the pot. The actuator on a gas stove is much faster than the actuator on an electric stove. The resolution of an actuator is the smallest change in output it can reliably generate. Just as with the PID controller, the resolution of the actuator (converted into equivalent units on the input) must be smaller than the desired accuracy of the control system as a whole. A **crisp output** is a software variable describing the output parameters having units (e.g., watts, Newtons, dynes/cm<sup>2</sup>). The **postprocessor** converts the crisp output into a form that can be directly output to the actuator. The postprocessor can verify that the output signals are within the valid range of the actuator. One of the advantages of fuzzy logic design is its connection to human intuition. Think carefully about how you control the actuator (gas pedal) when attempting to drive a car at a constant speed. There is no parameter in your brain specifying the exact position of the pedal (e.g., 50% pressed, 65% pressed), unless, of course, you are city taxicab driver and your brain allows two actuator states: full gas and full brake. Rather, what your brain creates as actuator commands are statements like “press the pedal a little harder” and “press the pedal a lot softer.” So, mimicking the way people think, the crisp output of fuzzy logic controller might be change in

pedal pressure  $\Delta U$ , and the postprocessor would calculate  $U = U + \Delta U$ , and then check to make sure  $U$  is within an acceptable range.

We continue the design of a fuzzy logic controller by analyzing its crisp inputs. As a design step, we create a list of true/false conditions that together describe the current state of the physical plant. In particular, we define **input fuzzy membership sets**, which are fuzzy logic variables describing conditions related to the state of the physical plant. These fuzzy variables do not need to be orthogonal. In other words, it is acceptable to have variables that are related to each other. When designing a speed controller, we could define multiple fuzzy variables referring to similar conditions, such as **WayTooFast**, **Fast**, and **LittleBitFast**. Given the scenario where we are driving too fast, there should be generous overlap in conditions, such that two or even three fuzzy sets are simultaneously partially true. On the other hand, it is important that the entire list of input fuzzy membership sets, when considered as an ensemble, form a complete description of the status of the physical plant. For example, if we are attempting to drive a car at a constant speed, then we might include input fuzzy variables **SlowingDown**, **GoingSteady**, and **SpeedingUp** describing the car's acceleration. **Fuzzification** is the mathematical step converting the crisp inputs into input fuzzy membership sets. The Freescale 9S12 defined a set of assembly language instructions to optimize the implementation of fuzzy logic controllers. The instruction **mem** performs the fuzzification process converting a crisp input into an input fuzzy membership set. When implementing fuzzy logic explicitly with C code, we will have available the full set of AND, OR, NOT fuzzy logic operations. On the other hand, if we are implementing the fuzzy logic controller using the built-in 9S12 assembly language instructions, we will have access only to the AND OR fuzzy logic operations. Therefore, if the fuzzy logic controller needs a logic parameter that is the complement of **Fast**, we can create an additional input fuzzy variable, **NotFast**, and calculate it during the fuzzification stage.

The heart of a fuzzy logic controller is the **fuzzy logic** itself, which is set of logic equations that calculate fuzzy outputs as a function of fuzzy inputs. An **output fuzzy membership set** is a fuzzy logic variable describing a condition related to the actuator. **Quickstop**, **SlowDown**, **JustRight**, **MorePower**, and **MaxPower** are examples of output fuzzy variables that might be used to describe the action to perform on the gas pedal. Like input fuzzy variables, output fuzzy variables exist in the continuum from definitely false (0) to definitely true (1). Just as with the input specification, it is also important to create a list of output membership fuzzy sets that, when considered as an ensemble, form a complete characterization of what we wish to be able to do with the actuator. We write fuzzy logic equations using AND/OR functions in a way similar to Boolean logic. The fuzzy logic AND is calculated as the minimum value of the two inputs, and the fuzzy logic OR is calculated as the maximum value of the two inputs. The 9S12 instruction **rev** will execute an entire set of fuzzy logic rules converting fuzzy inputs into fuzzy outputs. The design of the rules, like the other aspects of fuzzy control, follows the human intuition. For example, this fuzzy logic equations arises from a human thought.

*"We should slow down if we are going too fast or if we are speeding up and going a little bit too fast."*

**SlowDown = WayTooFast + SpeedingUp\*LittleBitFast**

**Checkpoint 13.13:** If **WayTooFast** is 50, **SpeedingUp** is 40, and **LittleBitFast** is 60, then what would be the calculated value for **SlowDown**?

The **defuzzification** stage of the controller converts the output fuzzy variables into crisp outputs. Although any function could be used, an effective approach is to use a weighted average. The 9S12 instructions **wav** and **ediv** together perform the defuzzification step, converting output fuzzy variables into a crisp output. Consider the case where the pedal pressure  $U$  varies from 0 to 100; thus, the crisp output  $\Delta U$  can take on values from -100

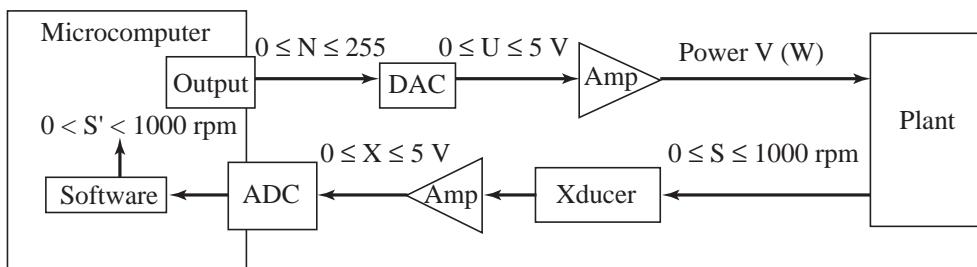
to +100. We think about what crisp output we want if just **QuickStop** were to be true. In this case, we wish to make  $\Delta U$  equal to -100. In a similar way, we then define crisp output values for **SlowDown**, **JustRight**, **MorePower**, and **MaxPower** as -10, 0, +10, and +100 respectfully. We can combine the five factors using a weighted average.

$$\Delta U = \frac{-100 * \text{QuicksStop} - 10 * \text{SlowDown} + 10 * \text{MorePower} + 100 * \text{MaxPower}}{\text{QuickStop} + \text{SlowDown} + \text{JustRight} + \text{MorePower} + \text{MaxPower}}$$

Because the fuzzy controller is modular, we begin by testing each of the modules separately. The system-level testing of a fuzzy logic controller follows a procedure similar to the PID controller tuning. Minimally intrusive debugging instruments can be added to record the crisp inputs, fuzzy inputs, fuzzy outputs, and crisp outputs during the real-time operation of the system. Fuzzification parameters are adjusted so that the status of the plant is captured in the set of values contained in the fuzzy input variables. Next, the rules are adjusted so that fuzzy output variables properly describe what we want to do with the actuator. Lastly, the defuzzification parameters are adjusted so the proper crisp outputs are created, optimizing both accuracy and response time.

**Example 13.7** Design a fuzzy logic controller using an ADC input and a DAC output. Implement the system using standard C programming.

**Solution** The objective of this example is to design a *fuzzy logic* microcomputer-based DC motor controller (Figure 13.18). The actuator is a DAC and linear power amplifier. The power to the motor is controlled by varying the 8-bit DAC output voltage. An amplifier provides power to the motor that is linearly related to the DAC output. The motor speed is estimated with a tachometer connected to an 8-bit ADC.



**Figure 13.18**

Interface of a motor controlled with fuzzy logic.

Our system has

- Two control inputs
 

$S^*$	desired motor speed, rpm
$S'$	current estimated motor speed, rpm
- One control output
 

$N$	digital value that we write to the DAC
-----	--

To utilize 8-bit mathematics, we change the units of speed to  $1000/256 = 3.90625$  rpm.

$$\begin{aligned} T^* &= (256 \cdot S^*)/1000 && \text{desired motor speed, 3.9 rpm} \\ T' &= (256 \cdot S')/1000 && \text{current estimated motor speed, 3.9 rpm} \end{aligned}$$

For example, if the desired speed is 500 rpm, then  $T^*$  will be 128. Notice that the estimated speed  $T'$  is simply the ADC conversion value. Inherent in most control systems is

the concept of periodic execution. In other words, the control system functions (estimate state variables, control equation calculations, and actuator output) are performed on a regular and periodic basis for every  $\Delta t$  time unit. This allows signal-processing techniques to be used. We will let  $T'(n)$  refer to the current measurement and  $T'(n-1)$  refer to the previous measurement (i.e., the one measured  $\Delta t$  time ago).

In the fuzzy logic approach, we begin by considering how a human would control the motor. Assume your hand were on a joystick (or your foot on a gas pedal) and consider how you would adjust the joystick to maintain a constant speed. We select crisp inputs and outputs to base our control system on. It is logical to look at the error and the change in speed when developing a control system. Our fuzzy logic system will have two crisp inputs

$$\begin{aligned} E &= T^* - T' && \text{error in motor speed, 3.9 rpm} \\ D &= T'(n) - T'(n-1) && \text{change in motor speed, 3.9 rpm/time} \end{aligned}$$

Notice that if we perform the calculations of  $D$  on periodic intervals, then  $D$  will represent the derivative of  $T$ ,  $dT'/dt$ .  $T^*$  and  $T'$  are 8-bit unsigned numbers, so the potential range of  $E$  varies from  $-255$  to  $+255$ . Errors beyond  $\pm 127$  will be adjusted to the extremes  $+ 127$  or  $-128$  without loss of information. Program 13.10 gives the calculations in C.

### Program 13.10

Subtraction with overflow/underflow checking.

```
char static Subtract(unsigned char N, unsigned char M){
/* returns N-M */
unsigned short N16,M16;
short Result16;
N16=N;           /* Promote N,M */
M16=M;
Result16=N16-M16; /* -255•Result16•+255 */
if(Result16<-128) Result16 = -128;
if(Result16>127) Result16 = 127;
return(Result16);}
```

Program 13.11 gives the global definitions of the input signals and fuzzy logic crisp input.

### Program 13.11

Inputs and crisp inputs.

```
unsigned char Ts;      /* Desired Speed in 3.9 rpm units */
unsigned char T;       /* Current Speed in 3.9 rpm units */
unsigned char Told;    /* Previous Speed in 3.9 rpm units */
char D;               /* Change in Speed in 3.9 rpm/time units */
char E;               /* Error in Speed in 3.9 rpm units */
```

**Common error:** Neglecting overflow and underflow can cause significant errors.

The need for the special Subtract function can be demonstrated with the following example:

```
E = Ts - T; // if Ts = 200 and T = 50 then E will be -106!!
```

This function can be used to calculate both  $E$  and  $D$  (Program 13.12). Now, if  $Ts = 200$  and  $T = 50$ , then  $E$  will be  $+127$ .

### Program 13.12

Calculation of crisp inputs.

```
void CrispInput(void){
E=Subtract(Ts,T);
D=Subtract(T,Told);
Told=T;} /* Set up Told for next time */
```

To control the actuator, we could simply choose a new DAC value  $N$  as the crisp output. Instead, we will select  $\Delta N$ , which is the change in  $N$ , rather than  $N$  itself because it better mimics how a human would control it. Again, think about how you control the speed of your car when driving. You do not adjust the gas pedal to a certain position but rather make small or large changes to its position to speed up or slow down. Similarly, when controlling the temperature of the water in the shower, you do not set the hot/cold controls to certain absolute positions. Again you make differential changes to affect the actuator in this control system. Our fuzzy logic system will have one crisp output:

$$\Delta N \quad \text{change in output, } N = N + \Delta N, \text{ in DAC units}$$

Next we introduce fuzzy membership sets that define the current state of the crisp inputs and outputs. Fuzzy membership sets are variables that have true/false values. The value of a fuzzy membership set ranges from definitely true (255) to definitely false (0). For example, if a fuzzy membership set has a value of 128, you are stating the condition is halfway between true and false. For each membership set, it is important to assign a meaning or significance to it. The calculation of the input membership sets is called *fuzzification*. For this simple fuzzy controller, we will define six membership sets for the crisp inputs:

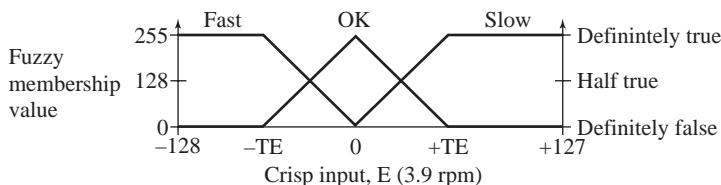
<i>Slow</i>	True if the motor is spinning too slow
<i>OK</i>	True if the motor is spinning at the proper speed
<i>Fast</i>	True if the motor is spinning too fast
<i>Up</i>	True if the motor speed is getting larger
<i>Constant</i>	True if the motor speed is remaining the same
<i>Down</i>	True if the motor speed is getting smaller

We will define three membership sets for the crisp output:

<i>Decrease</i>	True if the motor speed should be decreased
<i>Same</i>	True if the motor speed should remain the same
<i>Increase</i>	True if the motor speed should be increased

The fuzzy membership sets are usually defined graphically, but software must be written to actually calculate each. In this implementation, we will define three adjustable thresholds:  $TE$ ,  $TD$ , and  $TN$ . These are software constants and provide some fine-tuning to the control system. We will set each threshold to 20. If you build one of these fuzzy systems, try varying one threshold at a time and observe the system behavior (steady-state controller error and transient response). If the error  $E$  is  $-5$  (3.9 rpm units), the fuzzy logic will say that *Fast* is 64 (25% true), *OK* is 192 (75% true), and *Slow* is 0 (definitely false). If the error  $E$  is  $+21$  (in 3.9 rpm units), the fuzzy logic will say that *Fast* is 0 (definitely false), *OK* is 0 (definitely false), and *Slow* is 255 (definitely true).  $TE$  is defined to be the error (e.g., 20 in 3.9 rpm units is 78 rpm) above which we will definitely consider the speed to be too fast. Similarly, if the error is less than  $-TE$ , then the speed is definitely too slow (Figure 13.19).

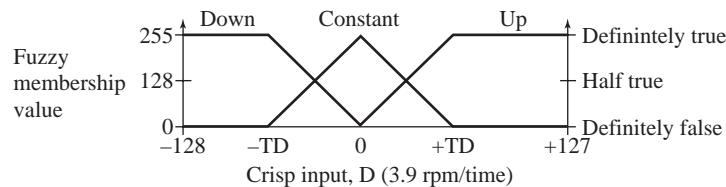
**Figure 13.19**  
Fuzzification of the error input.



In this fuzzy system, the input membership sets are continuous piecewise-linear functions. Also, the crisp input values (*Fast*, *OK*, *Slow*) sum to 255. In general, it is possible for the fuzzy membership sets to be nonlinear or discontinuous, and the

membership values do not have to sum to 255. The other three input fuzzy membership sets depend on the crisp input D. TD is defined to be the change in speed (e.g., 20 in 3.9 rpm/time units is 78 rpm/time) above which we will definitely consider the speed to be going up. Similarly, if the change in speed is less than  $-TD$ , then the speed is definitely going down (Figure 13.20).

**Figure 13.20**  
Fuzzification of the acceleration input.



In C, we could define a fuzzy function that takes the crisp inputs and calculates the fuzzy membership set values. Again TE and TD are software constants that will affect the controller error and response time (Program 13.13).

### Program 13.13

Calculation of the fuzzy membership variables in C.

```
#define TE 20
unsigned char Fast, OK, Slow, Down, Constant, Up;
#define TD 20
unsigned char Increase, Same, Decrease;
#define TN 20
void InputMembership(void){
    if(E <= -TE) { /* E≤-TE */
        Fast=255;
        OK=0;
        Slow=0;
    }
    else
        if (E < 0) { /* -TE<E<0 */
            Fast=(255*(-E))/TE;
            OK=255-Fast;
            Slow=0;
        }
        else
            if (E < TE) { /* 0<E<TE */
                Fast=0;
                Slow=(255*E)/TE;
                OK=255-Slow;
            }
            else { /* +TE≤E */
                Fast=0;
                OK=0;
                Slow=255;
            }
    if(D <= -TD) { /* D≤-TD */
        Down=255;
        Constant=0;
        Up=0;
    }
    else
        if (D < 0) { /* -TD<D<0 */
            Down=(255*(-D))/TD;
            Constant=255-Down;
            Up=0;
        }
        else
            if (D < TD) { /* 0<D<TD */
                Down=0;
                Up=(255*D)/TD;
                Constant=255-Up;
            }
            else { /* +TD≤D */
                Down=0;
                Constant=0;
                Up=255;
            }
}
```

The fuzzy rules specify the relationship between the input fuzzy membership sets and the output fuzzy membership values. It is in these rules that one builds the intuition of the controller. For example, if the error is within reasonable limits and the speed is constant, then the output should not be changed. In fuzzy logic we write

*If OK and Constant then Same*

If the error is within reasonable limits and the speed is going up, then the output should be reduced to compensate for the increase in speed. That is,

*If OK and Up then Decrease*

If the motor is spinning too fast and the speed is constant, then the output should be reduced to compensate for the error. That is,

*If Fast and Constant then Decrease*

If the motor is spinning too fast and the speed is going up, then the output should be reduced to compensate for both the error and the increase in speed. That is,

*If Fast and Up then Decrease*

If the error is within reasonable limits and the speed is going down, then the output should be increased to compensate for the drop in speed. That is,

*If OK and Down then Increase*

If the motor is spinning too slow and the speed is constant, then the output should be increased to compensate for the error. That is,

*If Slow and Constant then Increase*

These seven rules can be illustrated in table form (Figure 13.21).

**Figure 13.21**  
Fuzzy logic rules shown  
in table form.

		Down	Constant	Up
		Slow	Increase	Increase
		OK	Increase	Same
		Fast		Decrease
			Decrease	Decrease

It is not necessary to provide a rule for all situations. For example, we did not specify what to do for *Fast&Down* or for *Slow&Up*, although we could have added (but did not)

*If Fast and Down then Same*

*If Slow and Up then Same*

When more than one rule applies to an output membership set, then we can combine the rules:

*Same=(OK and Constant)*

*Decrease=(OK and Up) or (Fast and Constant) or (Fast and Up)*

*Increase=(OK and Down) or (Slow and Constant) or (Slow and Down)*

In fuzzy logic, the *and* operation is performed by taking the minimum, and the *or* operation is the maximum. Thus the C function that calculates the three output fuzzy membership sets is shown in Program 13.14.

#### Program 13.14

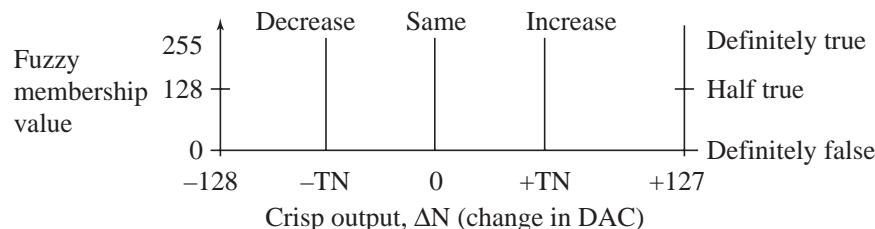
Calculation of the output fuzzy membership variables in C.

```
unsigned char static min(unsigned char u1,unsigned char u2){
    if(u1>u2) return(u2);
    else return(u1);}
unsigned char static max(unsigned char u1,unsigned char u2){
    if(u1<u2) return(u2);
    else return(u1);}
void OutputMembership(void){
    Same=min(OK,Constant);
    Decrease=min(OK,Up)
    Decrease=max(Decrease,min(Fast,Constant));
    Decrease=max(Decrease,min(Fast,Up));
    Increase=min(OK,Down)
    Increase=max(Increase,min(Slow,Constant));
    Increase=max(Increase,min(Slow,Down));}
```

The calculation of the crisp outputs is called *defuzzification*. The fuzzy membership sets for the output specifies the crisp output,  $\Delta N$ , as a function of the membership value. For example, if the membership set *Decrease* were true (255) and the other two were false (0), then the change in output should be  $-TN$  (where  $TN$  is another software constant). If the membership set *Same* were true (255) and the other two were false (0), then the change in output should be 0. If the membership set *Increase* were true (255) and the other two were false (0), then the change in output should be  $+TN$  (Figure 13.22).

**Figure 13.22**

Defuzzification of the  $\Delta N$  crisp output.



In general, we calculate the crisp output as the weighted average of the fuzzy membership sets:

$$\Delta N = (Decrease \cdot (-TN) + Same \cdot 0 + Increase \cdot TN) / (Decrease + Same + Increase)$$

The C compiler will promote the calculations to 16 bits and perform the calculation using 16-bit signed mathematics that will eliminate overflow on intermediate terms. The output  $dN$  will be bounded between  $-TN$  and  $+TN$ . Thus the C function that calculates the crisp output is shown in Program 13.15. When writing in assembly, you will need to deal with converting *Increase*, *Same*, *Decrease* from 8-bit unsigned to 16-bit

#### Program 13.15

Calculation of the crisp output in C.

```
char dN;
void CrispOutput(void){
    dN=(TN*(Increase-Decrease))/(Decrease+Same+Increase);
}
```

signed, and with overflow on intermediate terms. Just like in C, the output,  $\Delta N$ , will be bounded between  $-TN$  and  $+TN$ . If the calculation is rearranged, you can still use the 8-bit unsigned multiply. The numerator is

$$TN * Increase - TN * Decrease$$

using 8-bit unsigned multiply and 16-bit subtract. The divide calculation for  $dN$  needs to be 16-bit signed (Program 13.16).

### Program 13.16

Main program for fuzzy logic controller in C.

```
unsigned short Time;
void main(void){ short dT;
    ADC_Init(); // Program 11.13, use 8-bit mode
    Timer_Init(); // Program 2.10
    DDRT = 0xFF; // Actuator PTT
    Time = TCNT+1000; // time for first run
    N = 0;
    while(1){
        while((dT=Time-TCNT)>0){};
        Time = Time+1000; // TCNT value for next calculation
        T = ADC_In(0x80); // Sample A/D and set T, 0 to 255
        CrispInput(); // Calculate E,D and new Told
        InputMembership(); // Sets Fast,OK,Slow,Down,Constant,Up
        OutputMembership(); // Sets Increase,Same,Decrease
        CrispOutput(); // Sets dN
        N = max(0,min(N+dN,255));
        PTT = N;
    }
}
```

**Example 13.8** Design a fuzzy logic temperature controller. Implement the system with special op codes available on the 9S12.

**Solution** The objective of this section is to design a *fuzzy logic* microcomputer-based temperature controller. The desired temperature is  $T^*$ . The first step in designing a controller is choosing the input sensors and output actuators. There is only one sensor input, **Temperature**. There are two actuator outputs. **Heat** is a variable output (0 to 255) that will apply heat to the physical plant. **Fan** is a variable output (0 to 255) that will force air across the physical plant. **Fan+Heat** will warm up a very cold environment, **Heat** alone will slowly warm up the environment, and **Fan** alone will cool down a very hot environment. To conserve power, the use of the fan will be restricted.

The second step is selecting the crisp inputs and crisp outputs. Knowledge of how this physical plant works is critical in this step. What we wish to do depends on both absolute temperature and temperature error, so two crisp inputs will be employed:

**T** is the temperature (scaled to the range 0 to 255)

**E** is the temperature error (also scaled to the range 0 to 255)

```
; crisp inputs
T: ds 1 ;temperature (units 0.5 F)
E: ds 1 ;Measured-Desired (128 means no error) (units 0.125 F)
```

The crisp outputs will affect the two actuators:

**Heat** is the heater actuator control (0 to 255)

**Fan** is the fan actuator control (0 to 255)

```
;crisp outputs
Heat: ds 1 ; 0 is off and 255 is maximum heat
Fan: ds 1 ; 0 is off and 255 is maximum fan
```

The third step is choosing the fuzzy input and output membership sets. Here, we will divide each crisp input and output into three regions. Using five or seven regions would probably create a better controller, but for purposes of illustration, we will use only three. The three fuzzy membership inputs based on temperature are

<b>Cold</b>	means temperature of room is cold
<b>Normal</b>	means the temperature of room is a normal temperature
<b>Hot</b>	means temperature of room is hot

The three fuzzy membership inputs based on temperature error are

<b>TooCold</b>	means temperature of room is below the setpoint
<b>OK</b>	means temperature of room is correct
<b>TooHot</b>	means temperature of room is above the setpoint

The three fuzzy membership outputs based on the heater are

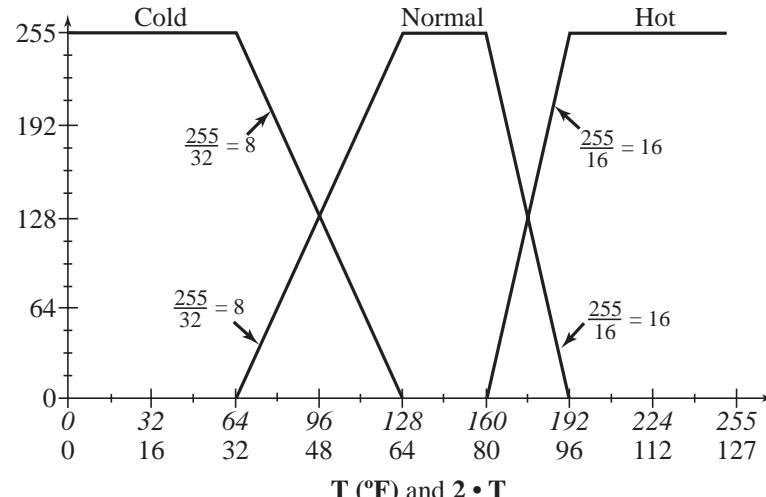
<b>NoHeat</b>	means heater should be off
<b>SomeHeat</b>	means heater should be on a little
<b>MaxHeat</b>	means heater should be on a lot

The three fuzzy membership outputs based on the fan are

<b>NoFan</b>	means fan should be off
<b>SomeFan</b>	means fan should be on a little
<b>MaxFan</b>	means fan should be on full speed

In the fourth step, we choose functions for the conversion of crisp inputs to input membership variables. The particular constants used in these functions will be used as a starting point. Once the system is built and tested, the values can be adjusted as needed (Figure 13.23 and Program 13.17).

**Figure 13.23**  
Fuzzification of the temperature input.

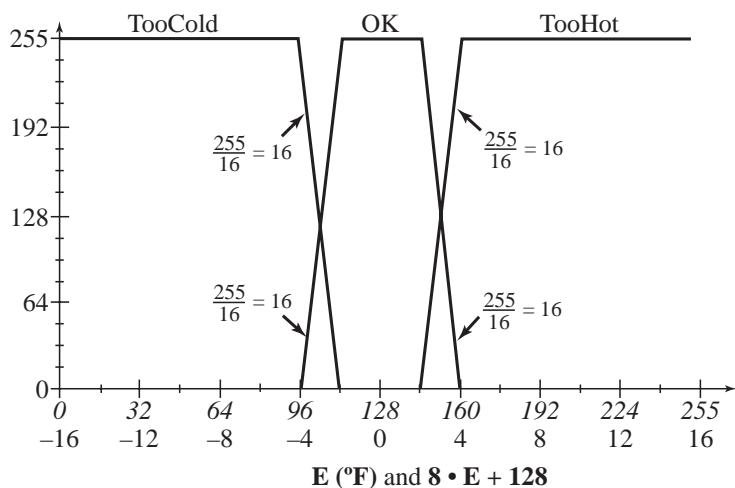


**Program 13.17**  
Fuzzification function for the temperature input in 9S12 assembly.

```
; format is Point1,Point2,Slope1,Slope2
T_tab: dc.b 160,255,16,0      ;Hot
       dc.b 64,192,8,16        ;Normal
       dc.b 0,64,0,8          ;Cold
```

We will define the error as OK if it is within  $\pm 2^{\circ}\text{F}$  of the setpoint (Figure 13.24 and Programs 13.18 and 13.19).

**Figure 13.24**  
Fuzzification of the temperature error input.



**Program 13.18**  
Fuzzification function for the temperature error input in 9S12 assembly.

```
E_tab:    dc.b 144,255,16,0      ;TooHot
          dc.b 64,192,8,16       ;OK
          dc.b 0,96,0,16         ;TooCold
```

**Program 13.19**  
Global variables in 9S12 assembly.

```
; input membership variables
fuzvar: ds 0      ; inputs
Hot:    ds 1
Normal: ds 1
Cold:   ds 1
TooHot: ds 1
OK:    ds 1
TooCold: ds 1
; output membership variables
fuzout: ds 0      ; outputs
NoHeat: ds 1      ; turn off heater
SomeHeat: ds 1    ; apply some heat
MaxHeat: ds 1     ; turn on heater
NoFan:  ds 1      ; turn off fan
SomeFan: ds 1    ; apply some fan
MaxFan: ds 1     ; turn on fan
; input membership variables relative offsets
hot:    equ 0
normal: equ 1
cold:   equ 2
toohot: equ 3
ok:    equ 4
toocold: equ 5
;output membership variables
noheat: equ 6    ; turn off heater
someheat: equ 7  ; apply some heat
maxheat: equ 8   ; turn on heater
nofan:  equ 9    ; turn off fan
somefan: equ 10  ; apply some fan
maxfan: equ 11   ; turn on fan
```

The fifth step is creating the fuzzy rules. Again, we use our intuition as a starting point. During the testing phase of the project, we develop a means to observe the values of the input membership variables and which rules apply in certain situations. The 9S12 background debug module is a convenient nonintrusive debugging tool that we can use to observe memory locations while the program is running (Program 13.20).

```
rules: dc.b hot,toohot,$FE,noheat,maxfan,$FE      ; use fan for max cooling
       dc.b hot,ok,$FE,noheat,somefan,$FE           ; use fan for some cooling
       dc.b normal,toohot,$FE,noheat,nofan,$FE        ; no fan for normal temps
       dc.b normal,ok,$FE,noheat,nofan,$FE            ; perfect
       dc.b normal,toocold,$FE,someheat,nofan,$FE     ; a little heat
       dc.b cold,ok,$FE,noheat,nofan,$FE              ; cold but perfect
       dc.b cold,toocold,$FE,maxheat,maxfan,$FE       ; fast warmup
       dc.b $FF
```

### Program 13.20

Fuzzy rules in 9S12 assembly.

The last design step is defuzzification. It is here we convert the true/false fuzzy output variables (`NoHeat`, `SomeHeat`, `MaxHeat`, `NoFan`, `SomeFan`, `MaxFan`) to crisp outputs (`Heat`, `Fan`) (Program 13.21).

```
Heatsingleton: dc.b 0,50,255
Fansingleton: dc.b 0,128,255
timehan: ldaa #$20 ;clear C5F
          staa TFLG1 ;Acknowledge
          ldd TC5
          addd #10000 ;next in 10 ms
          std TC5
          jsr MeasureTemperatire ; crisp input temperature, T
;reg A is temperature 0 to 255 (units 0.5•F)
          ldx #T_tab
          ldy #fuzvar
          mem      ; calculate Hot
          mem      ; calculate Normal
          mem      ; calculate Cold
          jsr MeasureError ; crisp input error, E
;reg A is error 0 to 255 (128 means no error) (units 0.125•F)
          ldx #E_tab
          mem      ; calculate TooHot
          mem      ; calculate OK
          mem      ; calculate TooCold
          ldab #6
cloop:  clr 1,y+ ; clear NoHeat, SomeHeat, MaxHeat, NoFan, SomeFan, MaxFan
          dbne b,cloop
          ldx #rules
          ldy #fuzvar
          ldaa #$FF
          rev
```

*continued on p. 681*

*continued from p. 680*

```

ldy #NoHeat ; pointer to NoHeat, SomeHeat, MaxHeat
ldx #Heatsingleton
ldab #3
wav
ediv
tfr y,d
stab Heat
ldy #NoFan ; pointer to NoFan, SomeFan, MaxFan
ldx #Fansingleton
ldab #3
wav
ediv
tfr y,d
stab Fan
rti

```

### Program 13.21

Fuzzy controller in 9S12 assembly.

Output fuzzy set	Singleton value
NoHeat	0
SomeHeat	50
MaxHeat	255

Output fuzzy set	Singleton value
NoFan	0
SomeFan	128
MaxFan	255

**Observation:** Fuzzy logic control will work extremely well (fast, accurate, and stable) if the designer has expert knowledge (intuition) of how the physical plant behaves.

## 13.6 Exercises

**13.1** For each term, give a definition in 32 words or less.

- |                                     |                           |
|-------------------------------------|---------------------------|
| a) State variable                   | f) Stability              |
| b) State estimator                  | g) Process reaction curve |
| c) Closed loop                      | h) Process reaction rate  |
| d) Steady-state controller accuracy | i) Anti-reset windup      |
| e) Transient response               |                           |

**13.2** For each control algorithm, give a definition in 32 words or less.

- |                |                |
|----------------|----------------|
| a) Open loop   | d) PID         |
| b) Bang-bang   | e) PI          |
| c) Incremental | f) Fuzzy logic |

**13.3** For each fuzzy logic term, give a definition in 32 words or less.

- |                         |                    |
|-------------------------|--------------------|
| a) Crisp input          | d) Fuzzy logic     |
| b) Fuzzy membership set | e) Defuzzification |
| c) Fuzzification        | f) Crisp output    |

**13.4** Which instructions are atomic and which can be suspended by an interrupt?

- |        |         |          |         |
|--------|---------|----------|---------|
| a) mem | c) revw | e) etbl  | g) wav  |
| b) rev | d) minm | f) emacs | h) ediv |

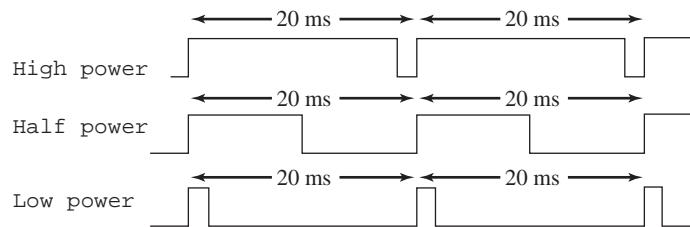
**13.5** Briefly explain why it is important to choose the proper update rate for a fuzzy logic controller. In particular, explain what happens to a fuzzy logic controller if the controller is executed too

infrequently. Similarly, explain what happens to a fuzzy logic controller if the controller is executed too frequently.

**13.6** Assume you have an 8-bit fuzzy logic system like the ones described in this chapter. Write formal descriptions for the complement and exclusive or fuzzy logic operations. Show C code implementations for these two functions.

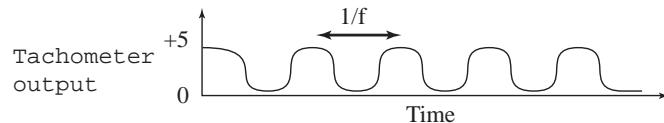
**D13.7** The objective of this exercise is to control the rotational speed of a DC motor. The +5 V DC motor has a resistance of  $10 \Omega$ . Your control system will apply a variable power to the motor by varying the duty cycle of a 50 Hz signal, as shown in Figure 13.25, for example.

**Figure 13.25**  
PWM output to actuator.



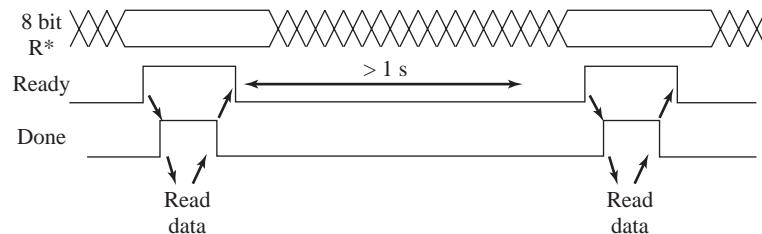
The rotation speed  $\mathbf{R}$  of the motor is measured by a tachometer. You should measure  $\mathbf{R}$  every 20 ms with a resolution of 1 rps. The rotational speed will vary from 0 to 250 rps ( $0 \leq \mathbf{R} \leq 250$  rps). The output of the tachometer is an ugly-looking digital wave with a frequency 100 times the motor speed. Thus,  $0 \leq f \leq 25$  kHz (Figure 13.26).

**Figure 13.26**  
Tachometer frequency is a function of the motor rotation speed.



The desired rotational speed ( $\mathbf{R}^*$  also in rps) comes from an 8-bit parallel input port. A new value is available on the rise of Ready. The fall of Done is an acknowledge signal back to the input device signifying that the microcomputer no longer needs the data. The timing is shown in Figure 13.27.

**Figure 13.27**  
Sensor timing.



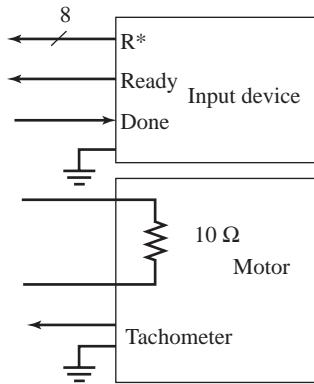
The interrupt-driven control algorithm should be implemented at 50 Hz ( $\mathbf{U}(t)$  is the duty cycle in percent):

$$\mathbf{U}(t) = \int_0^t (\mathbf{R}^* - \mathbf{R}(t)) dt \quad \text{where } \mathbf{R}^* \text{ and } \mathbf{R}(t) \text{ are in rps and } t \text{ is in seconds}$$

- a) Let  $\mathbf{u}(n)$  equal the cycle count (500 ns each) that controls the duty cycle of the output-compare variable-duty cycle 50 Hz squarewave. If the range of duty cycle is  $0 < \mathbf{U}(t) < 100$ , what is the relationship between  $\mathbf{u}(n)$  and  $\mathbf{U}(t)$ ?

- b)** Let  $\mathbf{R}(n)$  be the sampled sequence of measured rotational speed in rps. Convert the above integral control equation into discrete form. That is, determine the relationship that calculates  $\mathbf{u}(n)$  from  $\mathbf{u}(n-1), \mathbf{u}(n-2) \dots, \mathbf{R}(n), \mathbf{R}(n-1), \mathbf{R}(n-2) \dots$ , and  $\mathbf{R}^*$ .
- c)** Show the interface from the input device, the motor, and the tachometer to the microcomputer. Label chip numbers, resistors, and capacitor values (Figure 13.28).

**Figure 13.28**  
Sensor and motor interface.



- d)** Show the ritual software including data structures. The main program executes the ritual, and then performs other unrelated tasks (i.e., all processing occurs under interrupt control).
- e)** Show the interrupt software. You may poll any way you wish.

**D13.8** The objective of this exercise is to develop the fixed-point equations that implement a PID controller. You are to implement the following control system (Figure 13.29):

$X(t)$	is the state variable (V)
$X^*$	is the desired state (V)
$e(t) = X^* - X(t)$	is the error (V)
$V(t)$	is the actuator command (V)

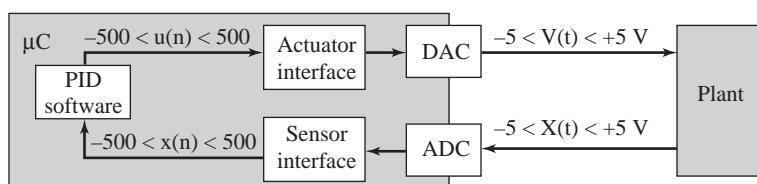
$$V(t) = K_p e(t) + K_I \int_0^t e(\tau) d\tau + K_D \frac{de(t)}{dt}$$

where  $K_p = 0.1$  (dimensionless)

$K_I = 10$  (1/s)

$K_D = 0.0001$  (s)

**Figure 13.29**  
Control system for Exercise 13.2.



The state estimator and actuator output hardware/software interfaces are given. Your PID software is given a signed 16-bit decimal fixed-point input,  $x(n)$ , that represents the current state variable. Similarly, your PID software will calculate a signed 16-bit decimal fixed-point output  $u(n)$  that will be fed to the actuator interface. For both cases, the fixed-point resolution,  $\Delta$ , is 0.01 V.

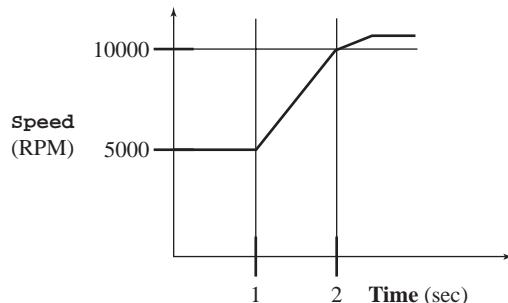
Assuming the digital controller is executed every 1 ms (1000 Hz), show the control equation to be executed in the periodic interrupt handler. In this process you will convert from floating to fixed-point numbers and convert from continuous to discrete time. No software is required; just

show the equations. Explain how you would deal with overflow/underflow. You should assume some noise exists on the sensor input.

**D13.9** Write a 9S12 assembly language program for the fuzzy logic controller described in Example 13.7 using the special fuzzy logic assembly instructions.

**D13.10** The objective of this problem is to use the Ziegler and Nichol approach to develop the PI controller equations that allow an embedded system to control a DC motor. The state variable is speed, which is measured using 16-bit input capture and has a measurement resolution of 1 rpm. The input capture device driver repeatedly updates a global variable, called **Speed**. This 16-bit unsigned variable has units of rpm and a range of 0 to 20000. The microcontroller uses pulse-width modulation to control power to the motor. The controller software writes to a global variable, called **Duty**, which ranges from 0 (0%) to 10000 (100%). The following plot shows an experimental measurement obtained when **Duty** is changed from 2500 to 5000. The desired speed is stored in the global variable, **Desired**, which has the same units as **Speed**. Design a fixed-point PI controller that takes **Speed** and **Desired** as inputs and calculates **Duty** as an output. From the response graph in Figure 13.30, estimate the **L** and **R** parameters of the Ziegler and Nichol method. How often should the controller be executed? Show just the equations (no software or hardware is required), calculating **Duty** as a function of **Speed** and **Desired**.

**Figure 13.30**  
A process reaction curve  
for the DC motor.



**D13.11** The objective of this problem is to use the Ziegler and Nichol approach to develop the PID controller equations that allow an embedded system to control the DC motor presented in Exercise D13.10 (i.e., work through the steps of Exercise 13.10 for a PID system).

**D13.12** Create a definition for a fuzzy logic complement. Let  $\sim A$  be the complement of  $A$ . Some of these logic equations are valid for fuzzy logic, and some are not. For each valid equation, present a formal proof of its correctness. For each invalid equation, give a counter example.

- |  |  |
|--|--|
| a) $A \cdot B = B \cdot A$                     | e) $(A+B) \cdot C = (A \cdot C) + (B \cdot C)$ |
| b) $A+B = B+A$                                 | f) $A + \sim A = \text{true}$                  |
| c) $(A \cdot B) \cdot C = A \cdot (B \cdot C)$ | g) $A * \sim A = \text{false}$                 |
| d) $(A+B)+C = A+(B+C)$                         | h) $(A \cdot B)+A = A$                         |

## 13.7 Lab Assignments

**Lab 13.1** The objective of this lab is to design a PID motor controller. The desired speed is received from a user interface (either a keypad or the SCI). You should use a variable-speed DC motor, paying careful attention to the voltage and current specifications of the motor. Attach onto the motor shaft a circular disk, and paint contrast lines in a circular pattern around the disk. Mount a reflectance optical sensor, such as the QRB1113 or QRB1134, pointing toward the disk, and interface it so that there is a digital squarewave with a frequency related to the speed of the shaft. Build the state estimator that measures shaft speed in real time. The first experimental measurement is to determine the current required to spin the motor at full speed. The next step is to design a PWM actuator so the software can control the delivered power to the motor. Measure the inductance of the motor ( $L$ ) and the turn-off time ( $\Delta t$ ) of the PWM switch. Use these parameters to mathematically determine the back EMF ( $V = L \cdot \Delta I / \Delta t$ ) that occurs when the motor is turned off.

Make sure the snubber diode can handle this voltage. Use the PWM in open-loop fashion and the state estimator to generate a DC response curve like that shown in Figure 13.24. Furthermore, you should measure a process reaction curve similar to that shown in Figure 13.14. Use the Ziegler and Nichol equations to design the initial PID controller, and implement it using fixed-point math. Run through an experimental fine-tuning, choosing PID parameters that minimize both controller error and response time.

**Lab 13.2** The objective of this lab is to design a fuzzy logic motor controller. The desired speed is received from a user interface (either a keypad or the SCI). You should use a variable-speed DC motor, paying careful attention to the voltage and current specifications of the motor. Attach onto the motor shaft a circular disk, and paint contrast lines in a circular pattern around the disk. Mount a reflectance optical sensor, such as the QRB1113 or QRB1134, pointing toward the disk, and interface it so that there is a digital squarewave with a frequency related to the speed of the shaft. Build the state estimator that measures both shaft speed and acceleration in real time. Measure the current required to spin the motor at full speed. Design a PWM actuator so the software can control the delivered power to the motor. It is OK if the actuator precision is greater than the 8-bit fuzzy logic numbers. Make sure the snubber diode can handle the back EMF generated when the motor is switched off. Use the PWM in open-loop fashion and the state estimator to generate a DC response curve like that shown in Figure 13.24. Furthermore, you should measure the time-constant of the physical plant (time to 0.69 of final response after a step change in input). Design and implement an initial fuzzy logic controller and add debugging instruments to measure fuzzy variables in real time. Run through an experimental fine-tuning, optimizing the fuzzy parameters in order to minimize both controller error and response time.

**Lab 13.3** The objective of this lab is to design a PID temperature controller. The goal is to control the temperature of a small object. The desired temperature is received from a user interface (either a keypad or the SCI). You can use a Peltier junction to deliver heat and a thermistor to measure temperature. Use PWM to control power to the Peltier junction, and use the ADC to measure temperature. If you use an H-bridge to control the Peltier junction, you will be able to both heat and cool the object. Be careful that your interface circuit can deliver the current required to activate the Peltier junction. Make sure there is good thermal contact between one side of the Peltier junction, the object, and the thermistor sensor. The next step is to design a PWM actuator so the software can control the delivered power to the junction. Use the PWM in open-loop fashion and the thermistor-based thermometer to generate a DC response curve like that shown in Figure 13.24. Furthermore, you should measure a process reaction curve similar to that shown in Figure 13.14. Use the Ziegler and Nichol equations to design the initial PID controller, and implement it using fixed-point math. Run through an experimental fine-tuning, choosing PID parameters that minimize both controller error and response time.

**Lab 13.4** The objective of this lab is to design a fuzzy logic temperature controller. The desired temperature is received from a user interface (either a keypad or the SCI port). The goal is to control the temperature of a small object. You can use a Peltier junction to deliver heat and a thermistor to measure temperature. Use PWM to control power to the Peltier junction, and use the ADC to measure temperature. If you use an H-bridge to control the Peltier junction, you will be able to both heat and cool the object. Be careful that your interface circuit can deliver the current required to activate the Peltier junction. Make sure there is good thermal contact between one side of the Peltier junction, the object, and the thermistor sensor. Build the state estimator that measures both temperature and temperature slope in real time. Design a PWM actuator so the software can control the delivered power to the junction. It is OK if the actuator precision is greater than the 8-bit fuzzy logic numbers. Use the PWM in open-loop fashion and the state estimator to generate a DC response curve like that shown in Figure 13.24. Furthermore, you should measure the time-constant of the physical plant (time to 0.69 of final response after a step change in input). Design and implement an initial fuzzy logic controller, and add debugging instruments to measure fuzzy variables in real time. Run through an experimental fine-tuning, optimizing the fuzzy parameters in order to minimize both controller error and response time.

# 14 Simple Networks

## Chapter 14 objectives are to:

- ❖ Introduce basic concepts of networks
- ❖ Present master/slave, ring, and multidrop networks based on the SCI interface
- ❖ Design and implement a controller area network (CAN)
- ❖ Introduce the concepts of wireless communication
- ❖ Define some of the terminology and approaches to modem communication

The goal of this chapter is to provide a brief introduction to communication systems. Communication theory is a richly developed discipline, and much of it is beyond the scope of this book. Nevertheless, the trend in embedded systems is to employ multiple intelligent devices. Consequently, their interconnection will be a strategic factor in the performance of the system. Given that various manufacturers are involved in the development of these devices; the interconnection network must be flexible, robust, and reliable. Because the emphasis of this book is on real-time embedded systems, this chapter focuses on implementing communication networks appropriate for embedded systems. The components of an embedded system typically are combined to achieve a common objective. Therefore, the nodes on the communication network must cooperate towards that shared goal. Requirements of an embedded system, in general, involve relatively low to moderate bandwidth, static configuration, and a low probability of corrupted data. In addition, reliability and low latency are important for real-time systems.

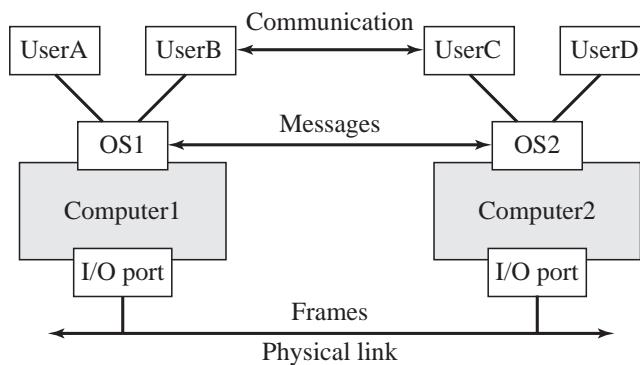
### 14.1 Introduction

In the serial interfacing chapter (Chapter 7), we considered the hardware interfaces between computers. In this chapter, we will build on those ideas and introduce the concepts of networks by investigating a couple of simple networks. A communication network includes both the physical channel (hardware) and the logical procedures (software) that allow users or software processes to communicate with each other. The network provides the transfer of information as well as the mechanisms for process synchronization. It is convenient to visualize the network in a hierarchical fashion. Figure 14.1 shows a three-layer communication system.

At the lowest level, frames are transferred between I/O ports of the two (or more) computers along the physical link or hardware channel. At the next logical level, the OS

**Figure 14.1**

A layered approach to communication systems.



of one computer sends *messages* or *packets* to the OS on the other computer. The message protocol will specify the types and formats of these messages. Typically, error detection and correction is handled at this level. Messages typically contain four fields.

#### 1. Address information field

Physical address specifying the destination/source computers

Logical address specifying the destination/source processes (e.g., users)

#### 2. Synchronization or handshake field

Physical synchronization, like shared clock, start, and stop bits

OS synchronization, like request connection or acknowledge

Process synchronization, like semaphores

#### 3. Data field

ASCII text (raw or compressed)

Binary (raw or compressed)

#### 4. Error detection and correction field

Vertical and horizontal parity

Checksum

Block correction codes (BCC)

**Observation:** Communication systems often specify bandwidth in total bits per second, but the important parameter is the information transfer rate.

**Observation:** Often the bandwidth is limited by the software and not the hardware channel.

At the highest level, we consider communication between users or processes. A *process* is a complete software task that has a well-defined goal. For example, when a file is to be printed on a network printer, the OS creates a process that

1. Establishes connection with the remote printer;
2. Reads blocks from the hard disk drive and sends the data to the printer;  
The OS printer driver may have to manipulate graphics/colors for the specific printer;  
The OS network driver will break the data into message packets;
3. Disconnects the printer.

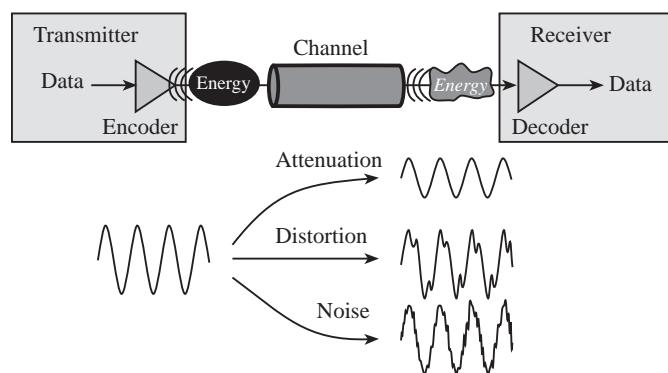
Many embedded systems require the communication of command or data information to other modules at either a near or a remote location. Because the focus of this book is on embedded systems, we will limit our discussion to communication with devices within the same room. A *full-duplex* channel allows data to transfer in both directions at the same time. In a *half-duplex* system, data can transfer in both directions but only in one direction at a time. Half-duplex is popular because it is less expensive (two wires) and allows the addition of more devices on the channel without change to the existing nodes.

Information (such as text, sound, pictures, and movies) can be encoded in digital form and transmitted across a channel, as shown in Figure 14.2. The channel will have a maximum information per second it can transmit, called *channel capacity*. In order to improve the effective bandwidth, many communication systems will compress the information at the source, transmit the compressed version, and then decompress the data at the destination. Compression essentially removes redundant information in such a way that the decompressed data is identical (*lossless*) or slightly altered but similar enough (*lossy*). For example, a 400 pixels/inch photo compressed using the JPG algorithm will be 5 to 30 times smaller than the original. A *guided medium* focuses the transmission energy into a well-defined path, such as current flowing along copper wire of a twisted pair cable or light traveling along a fiber optic cable. Conversely, an *unguided medium* has no focus, and the energy field diffuses as it propagates, such as sound or EM fields in air or water. In general, for communication to occur, the transmitter must encode information as *energy*, the channel must allow the energy to move from transmitter to receiver, and the receiver must decode the energy back into the information. In an analog communication system, energy can vary continuously in amplitude and time. A digital communication signal exists at a finite number of energy levels for discrete amounts of time. Along the way, the energy may be lost due to *attenuation*. For example, a simple  $V = I * R$  voltage drop is in actuality a loss of energy as electrical energy converted to thermal energy. A second example of attenuation is an RF cable splitter. For each splitter, there will be 50% attenuation, where half the energy goes left and the other half goes right through the splitter. Unguided media will have attenuation as the energy propagates in multiple directions. Attenuation causes the received energy to be lower in amplitude than the transmitted energy. A second problem is *distortion*. The transfer gain and phase in the channel may be function of frequency, time, or amplitude. Distortion causes the received energy to be different shape than the transmitted energy. A third problem is *noise*. The noise energy is combined with the information energy to create a new signal. White noise and EM field noise were discussed in Chapter 12. *Crosstalk* is a problem where energy in one wire causes noise in an adjacent wire. We quantify noise with the *signal-to-noise ratio* (SNR), which is the ratio of the information signal power to noise power.

$$\text{SNR(dB)} = 10 \cdot \log_{10} \left( \frac{\text{Average signal power}}{\text{Average noise power}} \right)$$

**Figure 14.2**

Information is encoded as energy, but errors can occur during transmission.



**Checkpoint 14.1:** Why do we measure SNR as power and not voltage?

We can make an interesting analogy between time and space. A communication system allows us transfer information from position A to position B. A digital storage system allows us transfer information from time A to time B. Many of the concepts (encoding/decoding information as energy, noise, error detection/correction, security, and compression) apply in an analogous manner to both types of systems.

**Checkpoint 14.2:** We measure the performance of a communication system as bandwidth in bits/sec. What is the analogous performance measure of a digital storage system?

Errors can occur when communicating through a channel with attenuation, distortion, and added noise. If the receiver detects an error, it can send a negative acknowledgement, so the transmitter will retransmit the data. The CAN and ZigBee protocols handle this detection–retransmission process automatically. Networks based on the SCI port will need to define and implement error detection. That is, we can add an additional bit to the serial frame for the purpose of detecting errors. With *even parity*, the sum of the data bits plus the parity bit will be an even number. The noise flag (NF) and framing error (FE) on the 9S12 also can be used to signify the data may be corrupted. The CAN network sends a LRC, which is the exclusive or of the bytes in the frame. The ZigBee network adds a checksum, which is the sum of all the data.

There are many ways to improve transmission in the channel, reducing the probability of errors. The first design choice is the selection of the interface driver. For example, RS422 is less likely to exhibit errors than RS232. Of course, having a driver will be more reliable than not having a driver. The second consideration is the cable. Proper shielding can improve SNR. For example, Cat6 Ethernet cables have a separator between the four pairs of twisted wire, which reduce the crosstalk between lines as compared to Cat5e cable. If we can separate or eliminate the source of added noise, the SNR will improve. Reducing the distance and reducing the bandwidth often will reduce the probability of error. If we must transmit long distances, we can use a repeater, which accepts the input and retransmits the data again.

Error correcting codes are beyond the scope of this book. However, we can present two simple error correcting codes. The first error correcting code involves sending three copies of each data. The receiver will compare the three versions received and majority vote will decide which value to use. A second error correcting code uses both parity and LRC. For example, assume we wished to send the message “Ciao.” Encoded as ASCII characters the data are \$43, \$69, \$61, and \$6F. The first step is to display the binary data in 2-D.

	Byte 0	Byte 1	Byte 2	Byte 3
Bit 7	0	0	0	0
Bit 6	1	1	1	1
Bit 5	0	1	1	1
Bit 4	0	0	0	0
Bit 3	0	1	0	1
Bit 2	0	0	0	1
Bit 1	1	0	0	1
Bit 0	1	1	1	1

The second step is to add an even parity to each byte and add a LRC at the end. Notice that the even parity is the exclusive OR of each bit in the vertical column and the LRC is the exclusive OR of each bit in the horizontal row. The parity bit for the LRC (or the LRC bit for the parity) will be the exclusive OR of all the data bits.

	Byte 0	Byte 1	Byte 2	Byte 3	LRC
Parity	1	0	1	0	0
Bit 7	0	0	0	0	0
Bit 6	1	1	1	1	0
Bit 5	0	1	1	1	1
Bit 4	0	0	0	0	0
Bit 3	0	1	0	1	0
Bit 2	0	0	0	1	1
Bit 1	1	0	0	1	0
Bit 0	1	1	1	1	0

Now, if any one bit in this 9-row by 5-column matrix is flipped, we can determine which byte is in error by the parity and which bit is in error by the LRC. Rather than asking for retransmission, we simply correct the error. These are very simple error correcting codes, but they illustrate that we can send more bits than the minimum and use those extra bits in a creative way to either detect or correct errors.

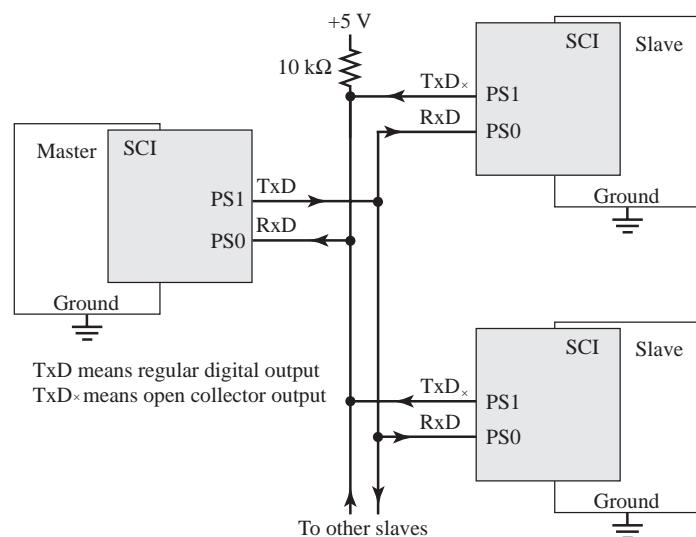
## 14.2 Communication Systems Based on the SCI Serial Port

In this section, we will present three communication networks that utilize the SCI port. If the distances are short, half-duplex can be implemented with simple *open collector* or *open-drain* TTL-level logic. Open collector logic has two output states: low and off. In the off state the output is not driven high or low, it just floats. The  $10\text{ k}\Omega$  pull-up resistor will passively make the signal high if none of the open collector outputs are low. The 9S12 can make its **TxD** serial output be open collector. This mode allows a half-duplex network to be created without any external logic (although pull-up resistors are often used). Three factors will limit the implementation of this simple half-duplex network: (1) the number nodes on the network, (2) the distance between nodes, and (3) presence of corrupting noise. In these situations, a half-duplex RS485 driver chip like the SP483 made by Sipex or Maxim can be used.

The first communication system uses a **master-slave** configuration, where the master transmit output is connected to all slave receive inputs, as shown in Figure 14.3. This provides for broadcast of commands from the master. All slave transmit outputs are connected together using wire-or open-collector logic, allowing for the slaves to respond one at a time. With the 9S12 the W0MS bit 0 in the slaves should be set to activate open collector mode on PS1. The low-level device driver for this communication system is identical to the SCI driver developed in Chapter 7. When the master performs SCI output, it is broadcast to all the slaves. There can be no conflict when the master transmits, because a single output is connected to multiple inputs. When a slave receives input, it knows it is a command from the master. A potential problem exists in the other direction, because multiple slave transmitters are connected to the same signal. If the slaves transmit only after specifically being triggered by the master, no collisions can occur.

**Figure 14.3**

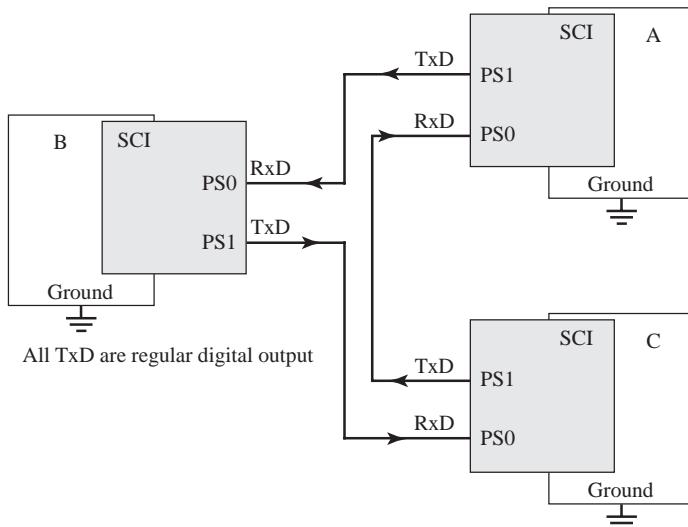
A master/slave network implemented with multiple microcomputers.



**Checkpoint 14.3:** What voltage level will the master RxD observe if two slaves simultaneously transmit, one making it a logic high and the other a logic low?

The next communication system is a **ring network**. This is the simplest distributed system to design, because it can be constructed using standard serial ports. In fact, we can build a ring network simply by chaining the transmit and receive lines together in a circle, as shown in Figure 14.4. Building a ring network is a matter as simple as soldering an RS232 cable in a circle with one DB9 connector for each node. Messages will include source address, destination address, and information. If Computer A wishes to send information to computer C, it sends the message to B. The software in Computer B receives the message, notices it is not for itself, and resends the message to C. The software in Computer C receives the message, notices it is for itself, and keeps the message. Although simple to build, this system has slow performance (response time and bandwidth), and it is difficult to add or subtract nodes.

**Figure 14.4**  
A ring network implemented with three microcomputers.



**Checkpoint 14.4:** Assume the ring network has 10 nodes, the baud rate is 100,000 bits/sec, and there are 10 bits per frame. What is the average time it takes to send a 10-byte message from one computer to another?

The third communication system implements a very common approach to distributed embedded systems, called **multi-drop**, as shown in Figure 14.5. To transmit a byte to the other computers, the software activates the SP483 driver and outputs the frame. Since it is half-duplex, the frame is also sent to the receiver of the computer that sent it. This echo can be checked to see whether a collision occurred (two devices simultaneously outputting). If more than two computers exist on the network, we usually send address information first, so that the proper device receives the data. The 9S12 SCI has a status bit in the SCISR2 register called RAF that will be true if there is an incoming frame on the RxD line. Many collisions can be avoided by checking this bit before transmitting.

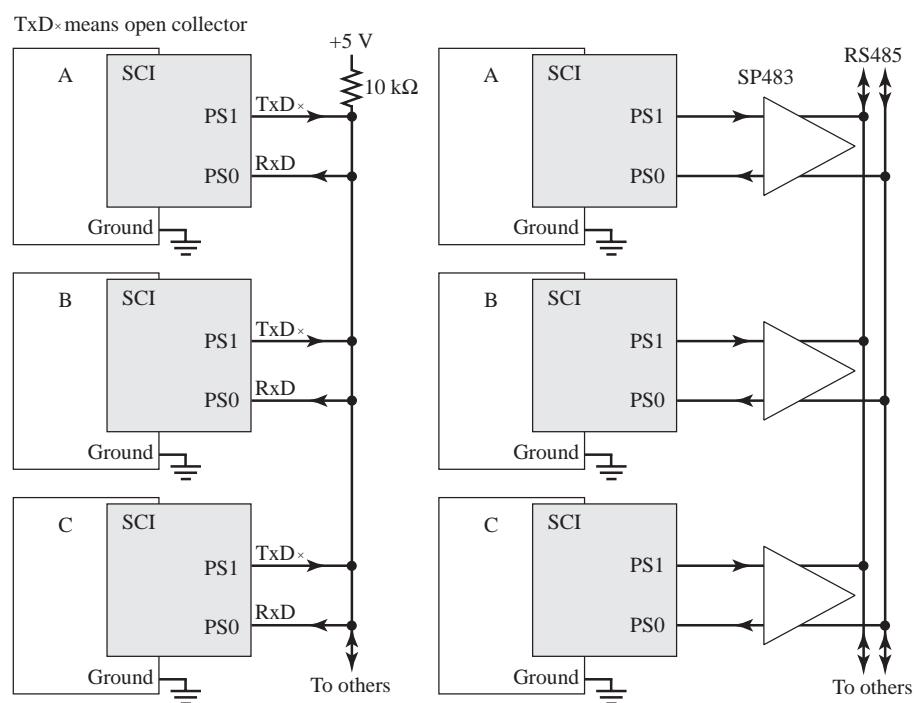
**Checkpoint 14.5:** How can the transmitter detect whether a collision has corrupted its output?

**Checkpoint 14.6:** How can the receiver detect whether a collision has corrupted its input?

There are many ways to check for transmission errors. You could use a **longitudinal redundancy check** (LRC) or horizontal even parity. The error check byte is simply the **exclusive-OR** of all the message bytes (except the LRC itself). The receiver also performs an **exclusive-OR** on the message as well as the error check byte. The result will equal zero if the block has been transmitted successfully. Another popular method is **checksum**, which is simply the modulo<sub>256</sub> (8-bit) or modulo<sub>65536</sub> (16-bit) sum of the data packet. In addition, each byte could (but doesn't have to) include even parity.

**Figure 14.5**

Two multi-drop networks implemented with three microcomputers.



There are two mechanisms that allow the transmission of variable amounts of data. Some protocols use start (STX = \$02) and stop (ETX = \$03) characters to surround a variable amount of data. The disadvantage of this “termination code” method is that binary data cannot be sent, because a data byte might match the termination character (ETX). Therefore, this protocol is appropriate for sending ASCII characters. Another possibility is to use a byte count to specify the length of a message. Many protocols use a byte count. The S19 records, for example, have a byte count in each line.

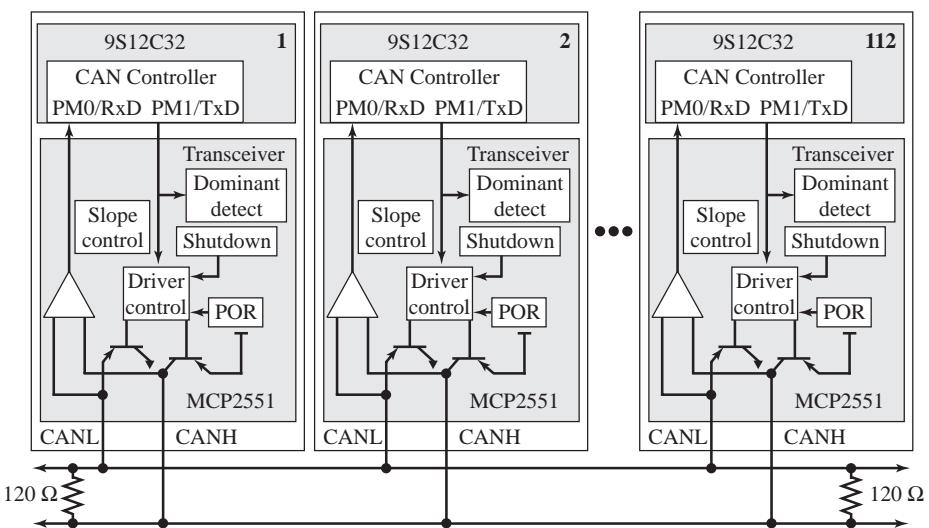
## 14.3 Design and Implementation of a Controller Area Network (CAN)

### 14.3.1 The Fundamentals of CAN

In this section, we will design and implement a Controller Area Network (CAN). A CAN is a high-integrity serial data communications bus that is used for real-time applications. It can operate at data rates of up to 1 Mbit/s, having excellent error-detection and confinement capabilities. The CAN was originally developed by the Robert Bosch Corporation for use in automobiles, and is now extensively used in industrial automation and control applications. The CAN protocol has been developed into an international standard for serial data communication, specifically the ISO 11989. Figure 14.6 shows the block diagram of a CAN system, which can have up to 112 nodes. There are four components of a CAN system. The first part is the CAN bus, consisting of two wires (CANH, CANL) with 120 Ω termination resistors on each end. The second part is the transceiver, which handles the voltage levels and interfacing the separate receive (Rx) and transmit (Tx) signals onto the CAN bus. The third part is the CAN controller, which is hardware built into the 9S12C32; it handles message timing, priority, error detection, and retransmission. The last part is software running within the 9S12C32, which handles the high-level functions of generating data to transmit and processing data received from other nodes.

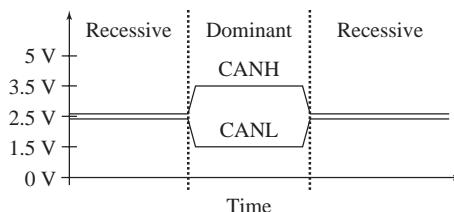
Each node consists of a 9S12C32 microcontroller (with an internal CAN controller) and a transceiver that interfaces the CAN controller to the CAN bus. A **transceiver** is a device capable of transmitting and receiving on the same channel. The CAN is based on the “broadcast communication mechanism,” which follows a message-based transmission protocol rather than an address-based protocol. The CAN provides two communication

**Figure 14.6**  
Block Diagram of a  
9S12C32-based CAN  
communication system.



services: the sending of a message (data frame transmission) and the requesting of a message (remote transmission request). All other services such as error signaling or automatic retransmission of erroneous frames are user-transparent, which implies that the CAN interface automatically performs these functions. The MC9S12C32 has an integrated CAN interface. The physical channel consists of two wires containing in differential mode one digital logic bit. Because multiple outputs are connected together, there must be a mechanism to resolve simultaneous requests for transmission. In a manner similar to open collector logic, there are **dominant** and **recessive** states on the transmitter, as shown in Figure 14.7. The outputs follow a wired-and-mechanism in such a way that if one or more nodes are sending a dominant state, it will override any nodes attempting to send a recessive state.

**Figure 14.7**  
Voltage specifications  
for the recessive and  
dominant states.



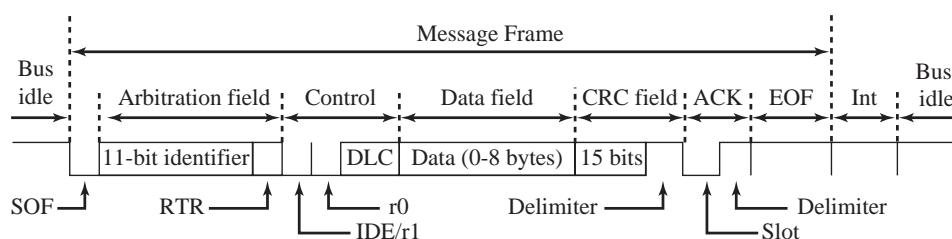
**Checkpoint 14.7:** What are the dominant and recessive states in open collector logic?

The CAN transceiver is a high-speed, fault-tolerant device that serves as the interface between a CAN protocol controller (located in the 9S12C32) and the physical bus. The transceiver is capable of driving the large current needed for the CAN bus and has electrical protection against defective stations. Typically, each CAN node must have a device to convert the digital signals generated by a CAN controller to signals suitable for transmission over the bus cabling. The transceiver also provides a buffer between the CAN controller and the high-voltage spikes than can be generated on the CAN bus by outside sources. Examples of CAN transceiver chips include the AMIS-30660 high-speed CAN transceiver, the Infineon Technologies TLE6250GV33 transceiver, the ST Microelectronics L9615 transceiver, the Philips Semiconductors AN96116 transceiver, and the Microchip MCP2551 transceiver. These transceivers have similar characteristics and are equally suitable for implementing a CAN system.

In a CAN system, messages are identified by their contents rather than by addresses. Each message sent on the bus has a unique identifier, which defines both the content and the priority of the message. This feature is especially important when several stations compete for bus access, a process called **bus arbitration**. As a result of the content-oriented

**Figure 14.8**

CAN standard format data frame.



addressing scheme, a high degree of system and configuration flexibility is achieved. It is easy to add stations to an existing CAN network.

Four message types or frames can be sent on a CAN bus. These include the **Data Frame**, the **Remote Frame**, the **Error Frame**, and the **Overload Frame**. This section will focus on the Data Frame, whose parts in standard format are shown in Figure 14.8. The **Arbitration Field** determines the priority of the message when two or more nodes are contending for the bus. For the Standard CAN 2.0A, it consists of an 11-bit identifier. For the extended CAN 2.0B, there is a 29-bit identifier. The identifier defines the type of data. The **Control Field** contains the DLC, which specifies the number of data bytes. The **Data Field** contains zero to eight bytes of data. The **CRC Field** (Cyclic Redundancy Check) contains a 15-bit checksum used for error detection. Any CAN controller that has been able to correctly receive this message sends a dominant ACK bit at the end of each message. This bit is stored in the Acknowledge slot in the CAN data frame. The transmitter checks for the presence of this bit, and if no acknowledge is received, the message is retransmitted. To transmit a message, the software must set the 11-bit identifier, set the 4-bit DLC, and give the 0 to 8 bytes of data. The receivers can define filters on the identifier field, so that only certain message types will be accepted. When a message is received, the software can read the identifier, length, and data. The **Intermission Frame Space (IFS)** separates one frame from the next. There are two factors that affect the number of bits in a CAN message frame. The ID (11 or 29 bits) and the Data fields (0, 8, 16, 24, 32, 40, 48, 56, or 64 bits) have variable length. The remaining components (36 bits) of the frame have fixed length including SOF (1), RTR (1), IDE/r1 (1), r0 (1), DLC (4), CRC (15), and ACK/EOF/intermission (13). For example, a Standard CAN 2.0A frame with two data bytes has  $11 + 16 + 36 = 63$  bits. Similarly, an Extended CAN 2.0B frame with four data bytes has  $29 + 32 + 36 = 97$  bits.

If a long sequence of 0's or a long sequence of 1's is being transferred, the data line will be devoid of edges that the receiver needs to synchronize its clock to the transmitter. In this case, measures must be taken to ensure that the maximum permissible interval between two signal edges is not exceeded. **Bit Stuffing** can be utilized by inserting a complementary bit after five bits of equal value. Some CAN systems add stuff bits, where the number of stuff bits depends on the data transmitted. Assuming  $n$  is the number of data bytes (0 to 8), CAN 2.0A may add  $3 + n$  stuff bits and a CAN 2.0B may add  $5 + n$  stuff bits. Of course, the receiver has to un-stuff these bits to obtain the original data.

The urgency of messages to be transmitted over the CAN network can vary greatly in a real-time system. Typically, there are one or two activities that require high transmission rates or quick responses. Both bandwidth and response time are affected by message priority. Low-priority messages may have to wait for the bus to be idle. There are two priorities occurring as the 9S12C32 CANs transmit messages. The first priority is the 11-bit identifier, which is used by all the CAN controllers wishing to transmit a message on the bus. Message identifiers are specified during system design and cannot be altered dynamically. The 11-bit identifier with the lowest binary number has the highest priority. In order to resolve a bus access conflict, each node in the network observes the bus level bit by bit, a process known as bit-wise arbitration. In accordance with the wired-and-mechanism, the dominant state overwrites the recessive state. All nodes with recessive transmission but

dominant observation immediately lose the competition for bus access and become receivers of the message with the higher priority. They do not attempt transmission until the bus is available again. Transmission requests are hence handled according to their importance for the system as a whole. The second priority occurs locally, within each CAN node. When a node has multiple messages ready to be sent, it will send the highest priority messages first.

### 14.3.2 Details of the 9S12C32 CAN

The 9S12C32 CAN receiver has a FIFO queue, which can hold up to five incoming messages, as shown in Figure 14.9. The 9S12C32 CAN transmitter uses a priority queue, which can hold up to three outgoing messages. To transmit a message the software writes the message into addresses \$0170 to \$017F. The software specifies the priority of the outgoing message (**CANTXTBPR** at \$017F). High-priority messages go to the front of the queue and are transmitted next. Low-priority messages go to the back of the queue and are transmitted only when no higher priority messages are ready. Once in the queue, the CAN hardware is responsible for handling priority, timing, transmitting the message, error detection, and retransmission if an error occurs. The 9S12C32 CAN receiver has a FIFO queue, which can hold up to five incoming messages. To retrieve the contents of an incoming message the software reads from addresses \$0160 to \$016F.

**Observation:** It is confusing when designing systems that use a sophisticated I/O interface such as a CAN to understand the difference between those activities automatically handled by the CAN hardware module and those your software must perform. The solution is to look at software examples to see exactly the kinds of tasks your software must perform.

**Figure 14.9**  
Data flow through the 9S12C32 CAN controller.

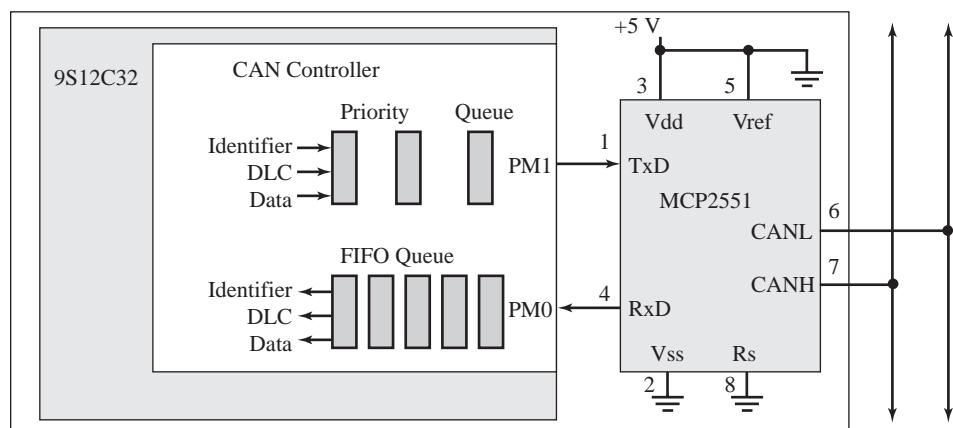


Table 14.1 shows some of the I/O ports used to program the 9S12C32 CAN. The **CANCTL0** and **CANCTL1** registers contain flags and control bits. **RXFMR** is the Received Frame Flag. It is set when a receiver has received a valid message correctly, independently of the filter configuration. Once set, it remains set until cleared by software or reset. Clearing is done by writing a “1” to the bit. **RXACT** is the Receiver Active Status flag. This read-only flag indicates find the CAN is receiving a message. **SYNCH** is the Synchronized Status flag. This read-only flag indicates whether the CAN is synchronized to the CAN bus and, as such, can participate in the communication process. **INITRQ** is the Initialization Mode Request bit. When this bit is set by the CPU, the CAN skips to Initialization Mode. Any ongoing transmission or reception is aborted, and synchronization to the bus is lost. The module indicates entry to Initialization Mode by setting **INITAK** = 1. **SLPRQ** is the Sleep Mode Request bit. This bit requests the CAN to enter Sleep Mode, which is an internal power-saving mode. The Sleep Mode request is serviced when the CAN bus is idle (i.e., the

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0140	RXFRM	RXACT	CSWAI	SYNCH	TIME	WUPE	SLPRQ	INITRQ	CANCTL0
\$0141	CANE	CLKSRC	LOOPB	LISTEN	0	WUPM	SLPAK	INITAK	CANCTL1
\$0142	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	CANBTR0
\$0143	SAMP	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10	CANBTR1
\$0144	WUPIF	CSCIF	RSTAT1	RSTAT0	TSTAT1	TSTAT0	OVRIF	RXF	CANRFLG
\$0145	WUPIE	CSCIE	RSTATE1	RSTATE0	TSTATE1	TSTATE0	OVRIE	RXFIE	CANRIER
\$0146	0	0	0	0	0	TXE2	TXE1	TXE0	CANTFLG
\$0147	0	0	0	0	0	TXEIE2	TXEIE1	TXEIE0	CANTIER
\$014A	0	0	0	0	0	TX2	TX1	TX0	CANTBSEL
\$014B	0	0	IDAM1	IDAM0	0	IDHIT2	IDHIT1	IDHIT0	CANIDAC
\$0150-	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0	CANIDAR0-
\$0153									CANIDAR3
\$0154-	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0	CANIDMR0-
\$0157									CANIDMR3
\$0158-	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0	CANIDAR4-
\$015B									CANIDAR7
\$015C-	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0	CANIDMR4
\$015F									CANIDMR7
\$0160	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	CANRXIDR0
\$0161	ID2	ID1	ID0	RTR	IDE=0	0	0	0	CANRXIDR1
\$0164-	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	CANRXDSR0-
\$016B									CANRXDSR7
\$016C	0	0	0	0	DLC3	DLC2	DLC1	DLC0	CANRXDLR
\$0170	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	CANTXIDR0
\$0171	ID2	ID1	ID0	RTR	IDE=0	0	0	0	CANTXIDR1
\$0174-	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	CANTXDSR0-
\$017B									CANTXDSR7
\$017C	0	0	0	0	DLC3	DLC2	DLC1	DLC0	CANTXDLR
\$017F	PRI07	PRI06	PRI05	PRI04	PRI03	PRI02	PRI01	PRI00	CANTXTBPR

**Table 14.1**

MC9S12C32 CAN ports.

module is not receiving a message and all transmit buffers are empty). The module indicates entry to Sleep Mode by setting **SLPAK** = 1. **CANE** is the CAN Enable bit, which we set to 1 to enable the CAN module. If it is 0, the module is disabled. **CLKSRC** is the CAN Clock Source bit, which defines the clock source for the CAN module. We set it to 1 to use the Bus Clock, and to 0 to use the Oscillator Clock. The frequency of the Oscillator Clock is equal to the frequency of the external crystal. The Bus Clock is the frequency at which data is accessed on the Bus and is a function of both the crystal and the PLL. We define the time quanta,  $T_q$ , as the period of the selected clock. **LISTEN** is the Listen Only Mode bit, which configures the CAN as a bus monitor. When the bit is set, all valid CAN messages with matching IDs are received, but no acknowledgement or error frames are sent out.

The **CANBTR0** and **CANBTR1** registers provide for bus timing control, which can be written only in initialization mode. **SJW1**, **SJW0** are the Synchronization Jump Width bits, which we will set to zero for high-speed communication. **BRP[5–0]** are Baud Rate Prescaler bits, and let  $x$  be the 6-bit number formed by these bits. The clock period used to create the individual bit timing is  $(x + 1) * T_q$ . **SAMP** is the Sampling bit, which determines the number of samples of the serial bus to be taken per bit time. If set, three samples per bit are taken the regular one (sample point) and two preceding samples using a majority rule. For higher bit rates, it is recommended that **SAMP** be cleared, which means that only one sample is taken per bit. There are three time segments for each transmitted bit. Segment 0 is exactly one

clock period, but the length of the other two periods is programmed using **CANBTR1**. The input bit is sampled at the time in between Segment 1 and Segment 2. **TSEG22-TSEG20** are the three Time Segment 2 bits, and let **y** be the 3-bit number formed by these bits. The length of Segment 2 will be **y + 1** clock periods. **TSEG13-TSEG10** are the four Time Segment 1 bits, and let **z** be the 4-bit number formed by these bits. The length of Time Segment 1 will be **z + 2** clock periods. The time for each bit includes all three segments

$$\text{Bit Time} = \mathbf{T_q} * (\mathbf{x} + 1)(3 + \mathbf{y} + \mathbf{z})$$

**Checkpoint 14.8:** What is the relationship between **y** and **z** if we wish to sample the input in the middle of the bit interval?

**CANRFLG** is the Receiver Flag Register. The **WUPIF CSCIF RSTAT1 RSTAT0 TSTAT1 TSTAT0 OVRIF** and **RXF** flags are cleared by writing a “1” to the corresponding bit position. Every flag has an associated interrupt arm bit in the **CANRIER** register. For low-power applications, we can place the system in Sleep Mode. **WUPIF** is the Wake-Up Interrupt Flag, which is used to detect bus activity while in Sleep Mode. This bit is 1 when it has detected activity on the bus and requested wake-up. **CSCIF** is the CAN Status Change Interrupt Flag. This flag is set when the CAN changes its current bus status as shown in the 4-bit (**RSTAT[1:0], TSTAT[1:0]**) status register. The coding for the bits **RSTAT1, RSTAT0** is:

00 = Rx OK:	0 " Receive Error Counter " 96
01 = Rx Warning:	96 < Receive Error Counter " 127
10 = Rx Error:	127 < Receive Error Counter " 255
11 = Bus-Off:	255 < Receive Error Counter

The coding for the bits **TSTAT1, TSTAT0** is:

00 = Tx OK:	0 " Transmit Error Counter " 96
01 = Tx Warning:	96 < Transmit Error Counter " 127
10 = Tx Error:	127 < Transmit Error Counter " 255
11 = Bus-Off:	255 < Transmit Error Counter

Excessive transmitter errors will turn off both the receiver and the transmitter. **OVRIF** is the Overrun Interrupt Flag, which is set when a data overrun condition occurs. In particular, an overrun occurs when five valid messages are in the receive FIFO and a sixth message is received. **RXF** is the Receive Buffer Full Flag, which is set by the CAN when a new message is shifted in the receiver FIFO. This flag indicates whether the shifted buffer is loaded with a correctly received message (matching identifier, matching Cyclic Redundancy Code (CRC), and no other errors detected). After the CPU has read that message from locations \$0160-\$016F, the **RXF** flag must be cleared to release the buffer. If armed (**RXFIE**), this bit will request an interrupt.

The software can configure the 9S12C32 CAN to filter incoming messages. Accepted messages will set the **RXF** flag and will be available for processing. Dropped messages will not set the **RXF** flag and will be discarded. **CANIDAR0-7** are the Identifier Acceptance Registers. **CANIDMR0-7** are corresponding the Identifier Mask Registers. These registers can be set only in initialization mode. **CANIDAC** is the Identifier Acceptance Control Register. The two bits **IDAM1 IDAM0** specify the Identifier Acceptance Mode.  $00_2$  means the eight acceptance registers are configured as two 32-bit filters.  $01_2$  means the eight acceptance registers are configured as four 16-bit filters.  $10_2$  means the eight acceptance registers are configured as eight 8-bit filters.  $11_2$  means the filter is closed, indicating that no message will be accepted and that the foreground buffer is never reloaded. On reception, each message is written into the background receive buffer. The CPU is signaled to read the message only if it passes the criteria in the identifier acceptance and identifier mask registers (accepted); otherwise, the message is overwritten by the next message (dropped). The acceptance registers of the CAN are applied on the **IDR0**

to IDR3 registers of incoming messages in a bit-by-bit manner. Mask bits **AM7-AM0** are set to 0 to specify that the corresponding bit will be filtered, and a mask bit of 1 means the corresponding bit will match (be acceptable) regardless of ID bit value. **AC7-AC0** comprise a user-defined sequence of bits with which the corresponding bits of the related identifier register (IDRn) of the receive message buffer are compared. The result of this comparison is then masked with the corresponding identifier mask register. The three bits **IDHIT2**, **IDHIT1**, and **IDHIT0** specify which filter is applied to the message currently available in the receive FIFO.

**Observation:** To enable the receiver to accept all messages set the mask registers to 0xFF.

**CANTFLG** is the Transmitter Flag Register. The flags are cleared by writing a “1” to the corresponding bit position. Every flag has an associated interrupt arm bit in the **CANTIER** register. **TXE2**, **TXE1**, and **TXE0** are the Transmitter Buffer Empty bits, which indicate that the associated transmit message buffer is empty and thus not scheduled for transmission. The CPU must clear the flag after a message is set up in the transmit buffer and is due for transmission. The CAN sets the flag after the message is sent successfully. The flag is also set by the CAN when the transmission request is successfully aborted due to a pending abort request. There are three transmit buffers in the priority, but only one is accessible at addresses \$0170-\$017F. **CANTBSEL** is the Transmit Buffer Selection register, which buffer will be accessible. In particular, **TX2**, **TX1**, and **TX0** are the Transmit Buffer Select bits. The lowest numbered bit places the respective transmit buffer in the \$0170-\$017F space (e.g., if **CANTBSEL** is 011<sub>2</sub>, transmit buffer 0 is selected). Read and write accesses to the selected transmit buffer will be blocked if the corresponding TXEx bit is cleared and the buffer is scheduled for transmission.

**IDE** is the ID Extended bit, which indicates whether the extended or standard identifier format is applied in this buffer. In the case of a receive buffer, the flag is set as received and indicates to the CPU how to process the buffer identifier registers. In the case of a transmit buffer, the flag indicates to the CAN what type of identifier to send. **IDE** = 1 means Extended format (29 bit), and **IDE** = 0 means Standard format (11 bit). **RTR** is the Remote Transmission Request bit, which reflects the status of the Remote Transmission Request bit in the CAN frame. In the case of a receive buffer, it indicates the status of the received frame and supports the transmission of an answering frame in software. In the case of a transmit buffer, this flag defines the setting of the RTR bit to be sent. **RTR** = 1 means Remote frame, and **RTR** = 0 means Data frame.

### 14.3.3 9S12C32 CAN Device Driver

The device driver for the 9S12C32-based CAN network is divided into three components: initialization, transmission, and reception. Although the 9S12C32 can handle standard and extended message formats, this software system will be configured to handle only the standard format. Program 14.1 gives the initialization code for the interface. The high-level software on all nodes of the network will call **CAN\_Open()** to initialize the CAN modules. If a node wishes to send 0 to 8 bytes of data to the other nodes, it would pass the information to **CAN\_Send()**, which will transmit the message via the CAN bus. This information would then be retrieved by the receiving nodes by calling **CAN\_Receive()**. The receiver will generate an interrupt when a new message is ready, and a FIFO queue will be used to pass the message from the background to the foreground. Each entry in the FIFO will be 11 bytes long: two bytes for the 11-bit ID, one byte for the 3-bit length, and eight bytes for the data. The CAN is enabled by setting the **CANE** bit. In order to set the configuration registers, the CAN must be in initialization mode. If the main program calls **CAN\_Open** a second time, there may be transmit or receive messages in progress. In order to prevent errors, this ritual will first request a transfer into Sleep Mode. This request will allow incoming and outgoing messages to complete before acknowledging that Sleep Mode has been entered. Once in Sleep Mode, this ritual can safely request the CAN enter Initialization Mode. The initialization

```

void CAN_Open(void){
    asm sei           // make atomic
    CANFifo_Init();   // Initialize FIFO data structure
    CANCTL1 |= 0x80;   // CANE=1, Enable CAN
    CANCTL0 |= 0x02;   // SLPRQ=1, go to sleep first
    while((CANCTL1&0x02)==0){}; // SLPACK signifies Sleep Mode
    CANCTL0 &= ~0x02;  // SLPRQ=0, leave Sleep Mode
    CANCTL0 |= 0x01;   // INITRQ=1, Enter Initialization Mode
    while((CANCTL1&0x01)==0){}; // INITAK signifies Initialization Mode
    CANCTL1 &= ~0x10;  // LISTEN=0, get out of Listen-only mode
    CANCTL1 &= ~0x40;  // CLKSRC=0, use oscillator clock
    CANIDAC = 0x10;    // four 16-bit filters
    CANIDMR0 = 0xFF;  CANIDMR1 = 0xFF; CANIDMR2 = 0xFF; CANIDMR3 = 0xFF;
    CANIDMR4 = 0xFF;  CANIDMR5 = 0xFF; CANIDMR6 = 0xFF; CANIDMR7 = 0xFF;
    CANBTR0 = 0x03;   // (x+1)=4, assume oscillator is 8 MHz
    CANBTR1 = 0x23;   // (3+y+z)=8, divide by 32 gives 250,000 bits/sec
    CANCTL0 &= ~0x01; // INITRQ=0, Leave Initialization mode
    while(CANCTL1&0x01){}; // wait for the end of initialization
    CANRIER |= 0x01;   // Arm RxF, interrupt on receive message
    asm cli           // Enable interrupts
}

```

**Program 14.1**

Initialization of the 9S12C32 CAN network.

sequence turns off Listen Mode, sets the clock, and establishes the acceptance filters. Setting all the acceptance masks to 0xFF means all messages will be accepted.

Program 14.2 shows the software used to transmit a message. It begins by waiting for an empty transmit buffer. After the first while loop, one or more bits in the **CANTFLG** register will be set. Each flag bit that is set means its corresponding buffer is free. By writing into **CANTBSEL** the CAN selects which buffer to use. In standard format, the **CANTXIDR0** and **CANTXIDR1** registers contain the 11-bit identifier. **IDE** is set to 0 to create a standard format message with 11-bit identifier. **RTR** is set to 0 to create a data frame. The message length is copied into the **CANTXDLR** register. Zero to eight bytes are copied into the data field of the message. The priority field is set to place this message into the 3-message priority queue maintained by the transmitter. If we write multiple bits into **CANTBSEL** when selecting which buffer to use, reading from it will return which buffer was selected. The last step (writing into **CANTFLG**) causes this message to be flagged as ready to transmit.

```

void CAN_Send(unsigned short id, char length, char *data, char priority) {
    char *pt=(char*)&_CANTXDSR0; // points to transmit message buffer
    while((CANTFLG&0x07)== 0){}; // Wait for transmit buffer available
    CANTBSEL = CANTFLG;          // Request selection of empty transmit buf
    CANTXIDR0 = id>>3;         // Write Identifier into ID registers
    CANTXIDR1 = id<<5;         // with RTR and IDE=0
    CANTXDLR = length;          // 0 to 8 bytes
    while(length){
        *pt++ = *data++;        // copy data into data registers
        length--;
    }
    CANTXTBPR = priority;       // set priority of this message
    CANTFLG = CANTBSEL;         // flag buffer as ready for transmission
}

```

**Program 14.2**

Transmitting a message on the 9S12C32 CAN network.

Program 14.3 shows the software used to receive a message. Interrupt synchronization is used so the main program doesn't have to be continuously checking for the presence of an incoming message. The **CANFifo** module implements a FIFO queue that puts and gets 13-byte messages. When a message has been properly received, the **RXF** flag is set and an interrupt is requested. The **CANFifo\_Put** function copies 13 bytes (the ID, Length and Data from the CAN buffer) from the CAN receive buffer into the FIFO queue. After the information has been copied, the receive buffer is released by writing a 1 to **RXF**. The high-level program can get the received message by calling **CAN\_Receive**.

```
void CAN_Receive(char msg[13]) {
    while (CANFifo_Get(msg) == 0){}; // wait for incoming message
}
interrupt 38 void CANInterruptHandler(void){
    char *msgPtr = (char*)&_CANRXIDR0;
    if(CANRFLG & RXF){
        CANCTL0 |= RXFRM;      // clear Received frame flag
        CANFifo_Put(msgPtr);
        CANRFLG |= RXF;        // clear RXF by writing a 1.
    }
}
```

### Program 14.3

Receiving a message from the 9S12C32 CAN network.

Program 14.4 shows an example main program used to send and receive messages. If it receives a message with ID equal to 50, then it will respond with a message ID of 51 and data equal to the sum of the received data.

```
void main(void){ // example foreground program
    char msg[13]; // received message
    unsigned short id; // ID of received message
    char length;
    char i;
    short sum;
    CAN_Open(); // activate CAN
    for(;;) {
        CAN_Receive(msg); // wait for incoming message
        id = (msg[0]<<3)+(msg[1]>>5); // bytes 0,1 are CANTXIDR0-1
        if(id == 50){
            length = msg[12]; // byte 12 is CANTXDLR
            i = 4;
            sum = 0;
            while(length){
                sum += msg[i++]; // bytes 4-11 are data
                length--;
            }
            CAN_Send(51,2,&sum,0);
        }
    }
}
```

### Program 14.4

Example main program that sends and receives messages on the 9S12C32 CAN network.

## 14.4 Wireless Communication

The details of how wireless communication operates are beyond the scope of this book. Nevertheless, the interfacing techniques presented in this book are sufficient to implement wireless communication by selecting a wireless module and interfacing it to the microcontroller. In general, one considers bandwidth, distance, topology and security when designing a wireless link. Bandwidth is the fundamental performance measure for a communication system. In this book, we define bandwidth of the system as the information transfer rate. However, when characterizing the physical channel, bandwidth can have many definitions. In general, the bandwidth of a channel is the range of frequencies passed by the channel (*Communication Networks* by Leon-Garcia). Let  $G_x(f)$  be the gain versus frequency of the channel. When considering EM fields transmitted across space, we can define *absolute bandwidth* as the frequency interval that contains all of the signal's frequencies. *Half-power bandwidth* is the interval between frequencies at which  $G_x(f)$  has dropped to half power (-3dB). Let  $f_c$  be the carrier frequency, and  $P_x$  be the total signal power over all frequencies. The *equivalent rectangular bandwidth* is  $P_x/G_x(f_c)$ . The null-to-null bandwidth is the frequency interval between first two nulls of  $G_x(f)$ . The FCC defines *fractional power containment bandwidth* as the bandwidth with 0.5% of signal power above and below the band. The *bounded power spectral density* is the band defined so that everywhere outside  $G_x(f)$  must have fallen to a given level. The purpose of this list is to demonstrate to the reader that, when quoting performance data, we must give both definition of the parameter and the data. If we know the channel bandwidth  $W$  in Hz and the **SNR**, we can use the *Shannon–Hartley Channel Capacity Theorem* to estimate the maximum data transfer rate  $C$  in bits/s:

$$C = W \cdot \log_2(1 + \text{SNR})$$

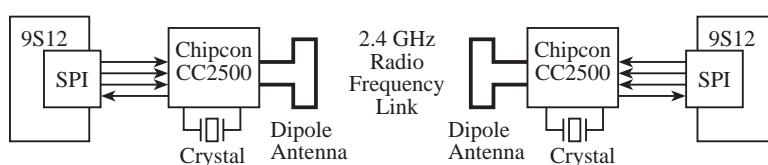
For example, consider a telephone line with a bandwidth  $W$  of 3.4 kHz and **SNR** of 38 dB. The dimensionless  $\text{SNR} = 10^{(38/10)} = 6310$ . Using the Channel Capacity Theorem, we calculate  $C = 3.4 \text{ kHz} * \log_2(1 + 6310) = 43 \text{ kbytes/s}$ .

One of the simplest modules we can use for wireless embedded systems is the Chipcon CC2500, which is a low-power 2.4-GHz RF transceiver with a SPI interface. The CC2500 is intended for 2400 to 2483.5 MHz Industrial, Scientific, and Medical (ISM) applications. The computer interface uses a SPI protocol, the clock circuit is based on an external crystal, and the antenna circuit must be tuned for the 2.4 GHz frequency. The eZ430-RF2500 from Texas Instruments is a low-cost development tool based around the MP430 and the Chipcon CC2500 low-power transceiver.

**Example 14.1** Design a system that can communicate at 1000 bytes/sec between two microcontrollers within the same room without security.

**Solution** This low bandwidth can be solved with a radio-frequency (RF) link without the complexities necessary to support BlueTooth or 802.11. This short distance is classified as a Short Range Device (SRD). There are many RF communication modules that could have been used. As illustrated in Figure 14.10, the CC2500 interface has three parts. The system

**Figure 14.10**  
Block diagram of a wireless link between two 9S12 systems.



will operate up to 500 kbytes/sec, and the chip implements dual 64-bit FIFOs for transmit and receive. Basically, the 9S12 on the left transmits data via its SPI, and the 9S12 on the right receives the data with its SPI. It is a transceiver, meaning data can flow across the link in both directions.

*ZigBee* is another low-cost wireless solution for embedded systems. The name is derived from the behavior of honey bees. Honey bees distributed across a large open field implement a mesh network in order to communicate information to their hive. They do this by message relaying. A bee distant from the hive will fly a particular zigzag pattern that represents the information. A second bee nearer the hive will repeat the pattern. The relay continues until the information reaches the hive. ZigBee is a standard that defines a set of communication protocols for low-data-rate, very low-power, short-range wireless networking. It can operate under battery power and last for years, because there are multiple types of low-power sleep mode. It is an appropriate solution for sensor networks, meter reading, industrial automation, security systems, and patient monitoring. ZigBee is an extension of the IEEE 802.15.4b standard. It is lower cost and lower performance than Bluetooth or IEEE 802.11b, as shown in Table 14.2. The range values in this table represent performance, indoors versus outdoors. ZigBee modules come in a variety of power versus performance models. In other words, you can run at lower power if you are willing to sacrifice distance and data rate.

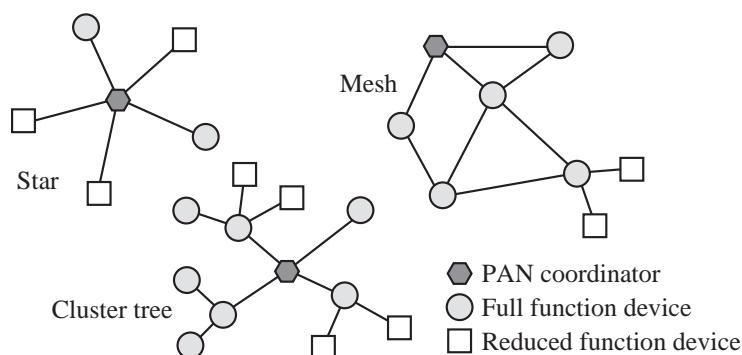
**Table 14.2**  
Comparison of wireless protocols.

	Data Rate	Range	Wireless Applications
ZigBee	20 to 250 Kbps	10–100 m	Sensor networks
Bluetooth	1 to 3 Mbps	2–10 m	Headset, mouse
IEEE 802.11b	1 to 11 Mbps	38–140 m	Internet connection
IEEE 802.11g	1 to 54 Mbps	38–140 m	Internet connection
IEEE 802.11n	1 to 72 Mbps	70–250 m	Internet connection

The ZigBee protocol is layered. The top layer is the application layer, implemented as the user program. At this layer, software in one node sends messages to another node. This section will focus on this layer because we will purchase a ZigBee module that performs the lower layers automatically. The second layer is the application support sublayer (APS). Below the APS is the network layer (NWK). Below the NWK is the media access control (MAC) layer. Below the MAC is the physical layer, which includes the RF transmitter and receiver. For more information about how ZigBee works, see <http://www.zigbee.org/>. You also could do a web search on Xbee, which is a low-cost implementation of ZigBee.

ZigBee is a personal area network (PAN), as shown in Figure 14.11. There can be a coordinator, devices that support all functions, or devices that support some functions. A typical application of ZigBee is a remote sensor network. Consider that each of the nodes

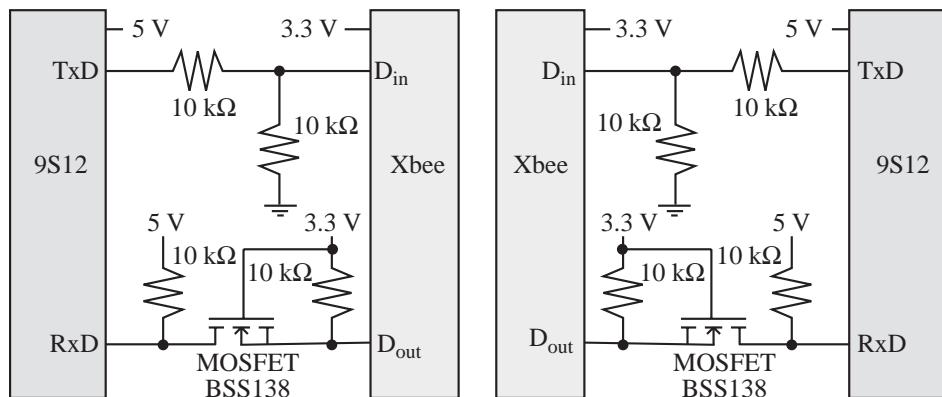
**Figure 14.11**  
ZigBee used to create a remote sensor network.



in Figure 14.11 is capable of collecting sensor data. The sensor database can be centralized at the PAN coordinator or distributed across the entire system.

One low-cost implementation of ZigBee is the Xbee module from Digi (formerly Maxstream). These modules take ZigBee and wrap it into a simple-to-use serial command set, called AT commands. These modules allow a very reliable and simple communication between microcontrollers with a serial port. Both point-to-point and multi-point networks are supported. The hardware involves interfacing 3.3-V full-duplex serial channel to the Xbee module, as shown in Figure 14.12. The 5-V to 3.3-V logic conversion was discussed previously in Section 7.8.

**Figure 14.12**  
ZigBee network  
implemented with Xbee  
modules.



The full software solution to this network will be left as Lab 14.3. In this simple configuration, there can be up to 256 nodes on the network. The initialization software must establish the baud rate at 9600 bits/sec. The system bandwidth therefore will be on the order of 960 bytes/sec. The <CR> symbol refers to the carriage return, character 13, existing as one 8-bit ASCII character. To place the Xbee in AT command mode, we execute Steps 1 to 5 over and over until we get an OK response.

1. Send a dummy character like ‘X’.
2. Wait 1.1 second (greater than the one second guard time).
3. Output “+++”.
4. Wait 1.1 second (greater than the one second guard time).
5. Wait for response, should be **OK<CR>**.

Each node must establish its address (called **my**) and the destination address. For example, if this computer is at address \$01 and wishes to send packets to the computer at address \$02, then this computer executes these AT commands. The software should wait 20 ms after each AT command. When an AT command is executed correctly, the Xbee responds with **OK<CR>**. One of the simplest modes is *Application Programming Interface* (API) mode 1, which allows the nodes to send and receive packets:

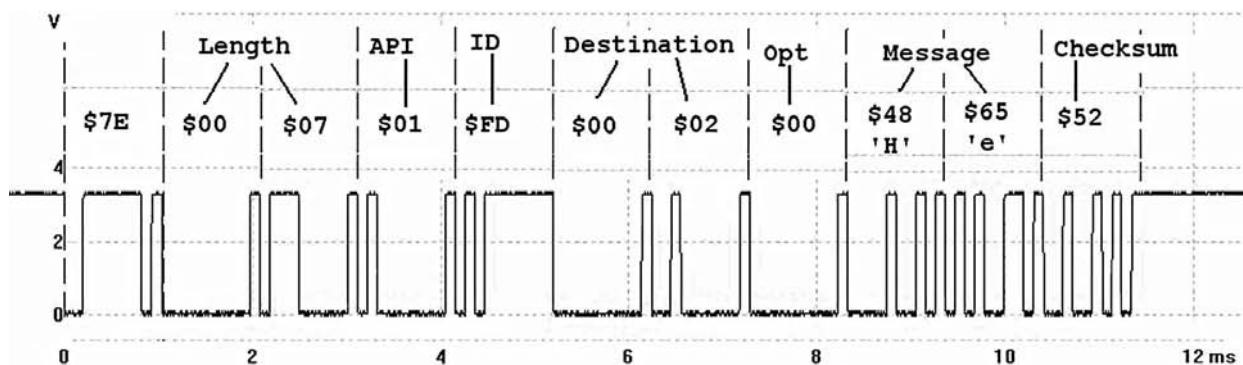
<b>ATDL02&lt;CR&gt;</b>	Sets destination address to 2 (number given in hexadecimal)
<b>ATDH0&lt;CR&gt;</b>	Sets destination high address to 0
<b>ATMY01&lt;CR&gt;</b>	Sets my address to 1 (number given in hexadecimal)
<b>ATAP1&lt;CR&gt;</b>	Sets API mode 1 (packets)
<b>ATCN&lt;CR&gt;</b>	Ends command mode

The other node in the point-to-point connection performs a similar initialization, but obviously with the **my** and **destination** addresses reversed. Using the API mode simplifies the application software, because all message routing, error detection/retransmission, and

message acknowledgement occurs at lower levels automatically. A data transmission frame has the following format:

**\$7E, LengthHi, LengthLo, \$01, ID, DestHi, DestLo, \$00, b1, ..., bn, Chksum**

All API mode 1 frames begin with \$7E. The next two bytes are the length of the message, which will be the number of bytes after the length and before the checksum. The length does not include the four bytes comprising the \$7E, which is the length itself and the checksum. The fourth byte, \$01, signifies this is a transmit data packet. The ID should be used as a message sequence number. That is, as this computer sends packets to the destination computer, the **ID** is sequenced as 0, 1, 2, ..., 255, 0, 1, .... This sequence number guarantees the packets arrive at the destination in the same order as they were sent. The two bytes **DestHi**, **DestLo** specify the destination node address. The high byte should be zero for this configuration. The next byte provides options for the frame, which should be 0. Bytes **b1** through **bn** are the data to be transmitted. Because there is a frame length, this data can be formatted however you wish. The last byte of the frame will be a checksum. Let **sum** be the 8-bit addition of all bytes not including \$7E delimiter and the length. We calculate **Chksum** as **\$FF-sum**. In this way, the receiver can add up all the bytes after the length and including the **Chksum** in order to get the result \$FF. For example, assume we wish to send the message “he” to node 2. Also assume this is the 254<sup>th</sup> frame sent, so the ID will be \$FD. The message has a length of 7. The checksum is calculated as  $\$FF - (\$01 + \$FD + \$00 + \$02 + \$00 + \$48 + \$65) = \$52$ . The oscilloscope recording for this frame is shown in Figure 14.13.



**Figure 14.13**

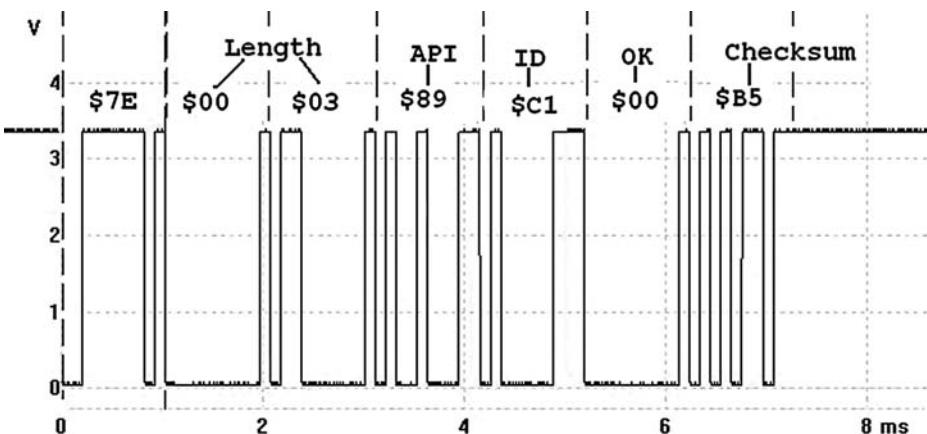
API transmit frame measured on the D<sub>in</sub> pin of the Xbee module.

When the transmitted frame is properly delivered, the Xbee sends an acknowledgement to the transmitter. The length is always three bytes. The API code is \$89. The **ID** matches the corresponding value of the transmitted frame. The next byte is a status field, and \$00 means success. A status of \$01 means no acknowledgement received, which may mean the destination node does not exist or is turned off. A status of \$02 means CCA failure, and \$03 means the message was purged. The **Chksum** byte is calculated in the same manner as all the API frames.

**\$7E, \$00, \$03, \$89, ID, \$00, Chksum**

Figure 14.14 shows a scope trace when an acknowledge frame was reported to the transmitter. The **ID** of this frame is \$C1. The checksum for this frame is  $\$FF - (\$89 + \$C1 + \$00) = \$B5$ .

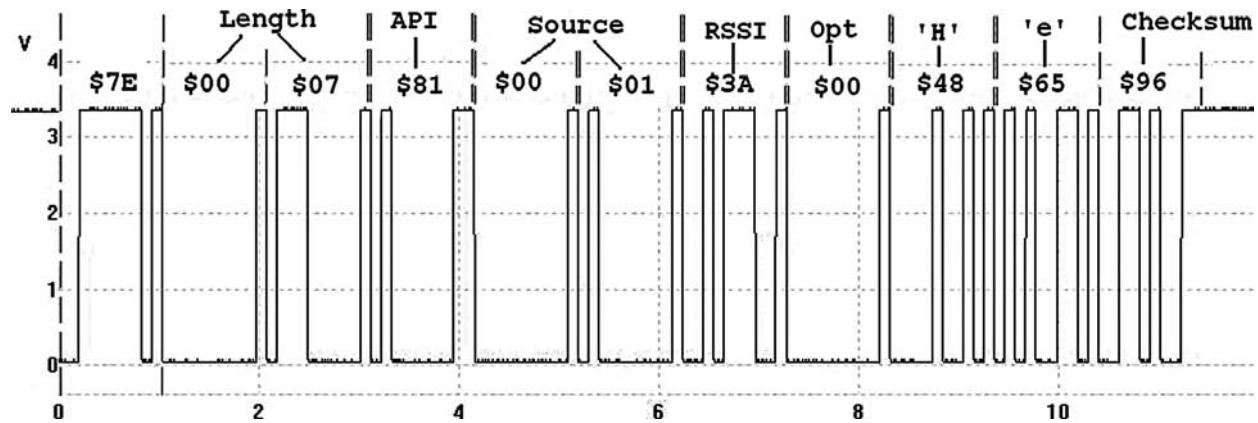
**Figure 14.14**  
API acknowledge frame measured on the D<sub>out</sub> pin of the Xbee module.



The receiver with a **my** address matching the **destination** address of the transmitted frame will be given that frame as an API packet type \$81. A data receive frame has the following format:

**\$7E, LengthHi, LengthLo, \$81, SourceHi, SourceLo, RSSI, \$00, b1, ..., bn, Chksum**

The source field identifies the node that sent the message. The **RSSI** (received signal strength indicator) is the decimal equivalent measure of the signal. For example \$3A means the received signal strength is 58 dBm. The option field should be zero. Bytes **b1** through **bn** contains the data. Figure 14.15 shows a scope trace of the received message corresponding to the transmitted message in Figure 14.13.



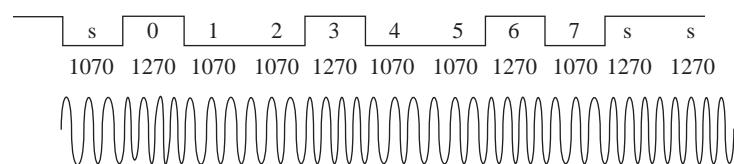
**Figure 14.15**  
API receive frame measured on the D<sub>out</sub> pin of the Xbee module.

## 14.5 Modem Communications

**14.5.1 FSK Modem** A *modem* is an electronic device that *MOdulates* and *DEModulates* a communication signal. The transmitter of a FSK modem modulates the digital signals into frequency-encoded sine waves. The receiver demodulates the sine waves back into digital signals. The transmitting computer asynchronous serial port (SCI) converts the digital data into a RS232 timing serial signal. The Bell103 or V.21 standard achieves a data rate of 300 bits/s. The

**Figure 14.16**

300 bits/sec FSK transmission of the character 'I' (originate to answer).

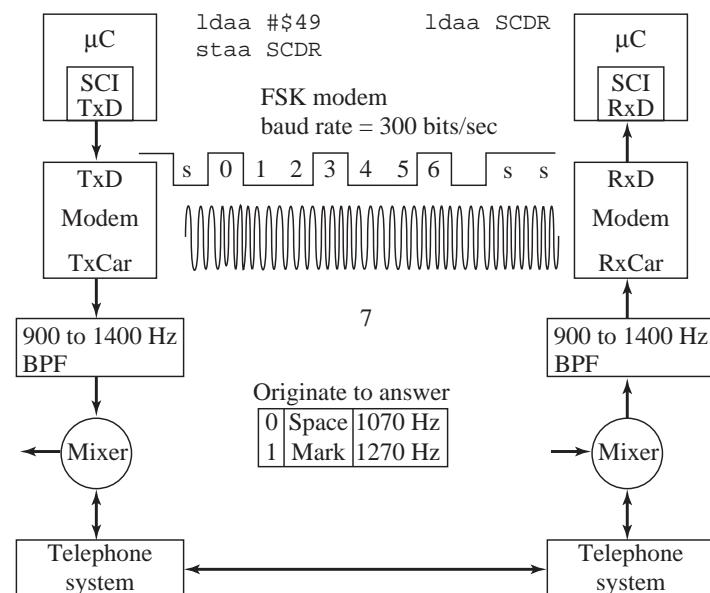


transmitting modem modulates the digital signal into 3.3-ms bursts of 1070- or 1270-Hz sounds. Figure 14.16 shows the encoded/decoded signals for the transmission of the ASCII character 'I,' which is \$49.

The transmitting bandpass filter guarantees there are no 2-kHz components in the output signal. The mixer adds the outgoing sound (1070 or 1270 Hz) with the incoming sounds on the telephone. The receiver bandpass filter will pass only the 1070/1270 Hz sounds to the demodulator. The receiver modem will demodulate the waveforms back into digital signals. If all goes well, the input to the receiving SCI, Rx<sub>D</sub>, should be the same as the output from the transmitting SCI, Tx<sub>D</sub>, only delayed in time. The software in the transmitting microcomputer should write data to the transmit serial data register (SCDR) when its TDRE = 1. Eventually, that data will arrive in the receiver's receive serial data register (SCDR), setting the receiver's RDRF (Figure 14.17).

**Figure 14.17**

Block diagram of a FSK modem communication system, showing one direction transfer.



To implement a full-duplex channel, we use a different set of frequencies to simultaneously transmit data in the other direction. We define the *originate* modem as the device that places the telephone call. It dials the telephone, originating the communication. The *answer* modem is the device that answers the phone. The *carrier frequency* is defined as the average or midvalue frequency. In the FSK protocol, the two carrier frequencies are 1170 and 2125 Hz. By convention, the modem that initiates the connection uses the lower carrier frequency. Once a connection has occurred, full-duplex communication can occur. A logical true or Boolean 1 is encoded as a *mark*, while false (0) is defined as a *space*. Table 14.3 defines the full-duplex FSK protocol.

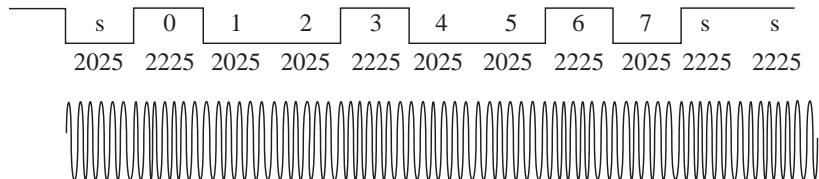
**Table 14.3**

Frequencies used in a 300 bits/s FSK modem.

Direction	Space, false, /0	Mark, true, 1
Originate to answer	1070 Hz	1270 Hz
Answer to originate	2025 Hz	2225 Hz

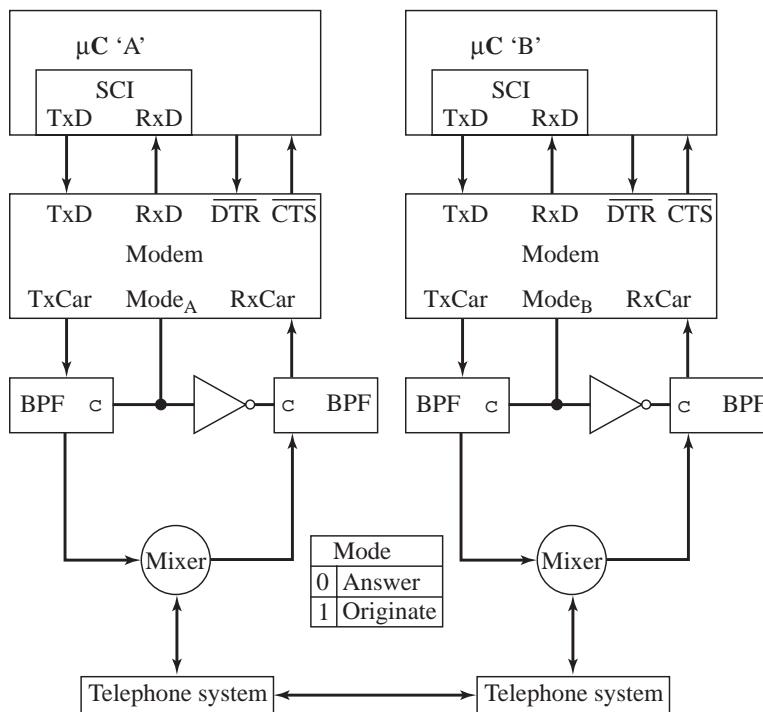
Figure 14.18 shows the encoded/decoded signals for the answer to originate transmission of a \$49.

**Figure 14.18**  
300 bits/s FSK transmission of the character 'I' (answer to originate).



An important function of the bandpass filters is to separate the two pairs of frequencies so that transmission in one direction does not interfere with transmission in the other (Figure 14.19).

**Figure 14.19**  
Block diagram of a FSK modem communication system, showing transfer in both directions.



If we develop a modem interface that can both originate and answer telephone calls, then we will need bandpass filters that have digitally controlled center frequencies. We will build a bandpass filter that has the following digital control.  $c$  is a digital input from the interface to the bandpass filter (Table 14.4). If computer A originates the call, then  $MODE_A = 1$ , and  $MODE_B = 0$ . These values are reversed if computer B originates the communication.

**Table 14.4**

Filters used in a 300 bits/s FSK modem.

c	Low frequency	High frequency
0	1900	2400
1	900	1400

The bandwidth of the FSK protocol is limited by the frequency response of the telephone system. At 300 bits/s the four sounds have 3, 4, 6, 7 cycles per bit time. These differences allow for accurate demodulation. If we wished to increase the communication bandwidth by a factor of 10, we would have to increase the carrier frequencies by this factor as well. For example, at 3000 bits/s, we would need carrier frequencies of 11 and 21 kHz to maintain the desirable 3, 4, 6, 7 cycles per bit time property. Unfortunately, the telephone was designed so that humans could comprehend audio speech, and it will not pass 21-kHz sound waves.

### 14.5.2 Phase-Encoded Modems

There are three basic components of the wave that we can use to encode the information: frequency, phase, and amplitude. Each of these characteristics can be exploited to encode information. A *phase-shift keying* (PSK) protocol encodes the information as phase changes between the sounds.

A *baud* is defined as a pulse of sound. In the early days of modem communications, the FSK protocol encoded just 1 bit/baud. To increase the communication bandwidth without increasing the carrier frequency, multiple bits are encoded into each change in sound. The *baud rate* is the number of sounds (or sound changes) transmitted per second. With a FSK protocol, the baud rate equals the data rate. On the other hand, with the phase-encoded schemes, the transmission rate will usually be higher than the baud rate. This is a very confusing point because many people improperly interchange the terms baud rate, data rate, and bandwidth. If we encode n bits of data per baud, then

$$\text{data rate} = n \cdot \text{baud rate}$$

For example, the 2400 bits/s International Telegraph and Telephone Consultative Committee (CCITT) protocol uses an 1800-Hz carrier and a 1200 bits/s baud rate. It encodes 2 bits/baud as shown in Table 14.5.

**Table 14.5**

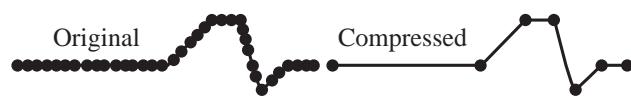
Phase shifts used in a 1200 bits/s PSK modem.

Data	CCITT phase
00	0°
01	90°
11	180°
10	270°

Encoding is the process of transmitting one or more bits of information with each baud. To improve bandwidth we can also implement data compression. Most data contain repetitive or redundant information. With data compression, the information (the message, digitally encoded sound, or photograph) is reformatted so that it occupies fewer bytes. The compressed data are transmitted across the channel and expanded on the other side. For example, a sampled waveform can be compressed into a sequence of linear line segments (Figure 14.20).

**Figure 14.20**

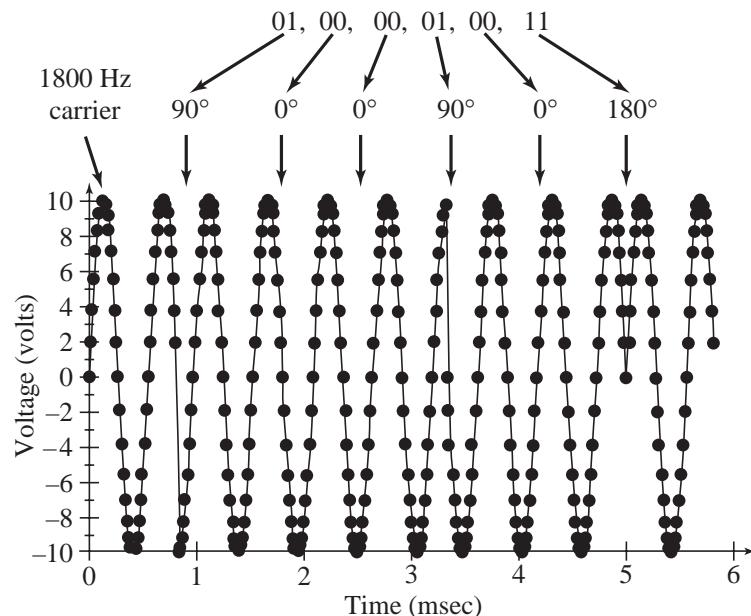
Compressed data allow information to be transferred with few data points.



For modem communications, we use Huffman or Lempel-Ziv encoding. We can evaluate a compression/decompression algorithm by three factors. The *compression ratio* is defined as the ratio of the number of original bytes to the number of compressed bytes. If an algorithm achieves a compression ratio of 4:1, then a 1000-byte message can be compressed into 250 bytes. Since only the compressed message is transmitted, the effective bandwidth of the communication system is four times faster than the physical bandwidth of the channel. The second factor to consider is the speed of the compression/decompression algorithm. For most modem applications, we require the compression/decompression functions to proceed in real time so that the operation occurs transparent to the user. The last and most important factor is accuracy. In some compression/decompression applications, such as photography or sound recordings, we can tolerate some imperfections or errors. In most modem applications however, we expect the received message to exactly equal the message transmitted.

Consider the following example of a CCITT 2400 bits/s PSK transmission. To transmit the ASCII 'A' (\$41 = %01000001) with one start bit, 8-bit data, even parity, and two stop bits, we transmit the sequence start = 0, data = 10000010, parity = 0, and stop = 11. Remember the data bits are transmitted starting with the least significant bit. The sequence is divided into 2-bit pieces: 01, 00, 00, 01, 00, 11 and encoded as phase shifts: 90°, 0°, 0°, 90°, 0°, 180°. With a carrier frequency of 1800 Hz, we transmit 1.5 cycles and then phase-shift the signal. The baud rate is 1200 because we produce 1200 sound changes per second. The data rate is 2400 bits/s, twice the baud rate, because each sound change encodes 2 bits of data (Figure 14.21).

**Figure 14.21**  
Waveform generated by  
a phase-encoded  
modem.



### 14.5.3 Quadrature Amplitude Modems

In a quadrature amplitude modem (QAM), we use both the phase and amplitude to encode up to 6 bits onto each baud. For standard QAM, only 4 bits is used to encode data. The trellis-coded quadrature amplitude modem (TCQAM or TCM) includes error-correcting signal processes so that all 6 bits can be reliably transmitted (Table 14.6).

**Table 14.6**

High-speed modems transmit many bits per sound.

<b>Standard</b>	<b>Baud rate</b>	<b>bits/baud</b>	<b>Data rate (bit/s)</b>	<b>Modulation</b>
V.21 Bell103	300	1	300	FSK
V.22 Bell212A	600	2	1,200	PSK
V.26	1200	2	2,400	PSK
V.27	1600	3	4,800	PSK
V.32	2400	4	9,600	QAM
V.33	2400	6	14,400	TCQAM
V.34	3200	10–11	33,600	TCM
V.92	8000	7	56,000	PCM

## 14.6 Exercises

**14.1** For each term, give a definition in 32 words or less.

- |                             |                               |
|-----------------------------|-------------------------------|
| <b>a)</b> Master-slave      | <b>f)</b> Crosstalk           |
| <b>b)</b> Multi-drop        | <b>g)</b> Stuff bits          |
| <b>c)</b> Channel capacity  | <b>h)</b> Frequency shift key |
| <b>d)</b> Ring network      | <b>i)</b> Phase encoding      |
| <b>e)</b> Lossy compression | <b>j)</b> Compression         |

**14.2** For each pair of terms, compare and contrast in 32 words or less.

- |   |                                   |
|---|-----------------------------------|
| <b>a)</b> Full-duplex versus half-duplex    | <b>d)</b> LRC versus checksum     |
| <b>b)</b> Lossless versus lossy compression | <b>e)</b> Originate versus answer |
| <b>c)</b> Guided versus unguided media      |                                   |

**14.3** What fundamental property is used to transmit information through a channel?

**14.4** Consider a telephone line with a channel bandwidth of 2 kHz and **SNR** of 40 dB. What is the maximum data rate possible?

**14.5** Consider a telephone line with a channel bandwidth of 4 kHz and **SNR** of 30 dB. What is the maximum data rate possible?

**14.6** Assume node \$31 wishes to send the message “Hello” to node \$32. Let ID = 0. Let RSSI = \$3A.

- a)** Give the API mode 1 frame node \$31 sends to its Xbee module. Include the entire message from \$7E up to and including the checksum.
- b)** Give the API mode 1 frame \$32 will receive from its Xbee module.

**14.7** Assume node \$41 wishes to send the message “Ciao” to node \$42. Let ID = \$12. Let RSSI = \$3A.

- a)** Give the API mode 1 frame node \$41 sends to its Xbee module. Include the entire message from \$7E up to and including the checksum.
- b)** Give the API mode 1 frame \$42 will receive from its Xbee module.

**14.8** Consider how the ACK bit is used in a CAN network.

- a)** What do the receivers do during the ACK bit?
- b)** What does it mean if the ACK bit is dominant?
- c)** What does it mean if the ACK bit is recessive?

**14.9** If the CAN channel is noisy, it is possible that some bits will be transmitted in error. Assume there are four nodes: one is transmitting, and three are receiving. What happens if a data bit is flipped in the channel due to noise being added into the channel?

**14.10** Consider a situation where two microcontrollers are connected with a CAN network. Computer 1 generates 8-bit data packets that must be sent to Computer 2, and Computer 2 generates 8-bit data packets that must be sent to Computer 1. The packets are generated at random times, and the goal is to minimize the latency between when a data packet is generated on one computer to when it is received on the other. Describe the CAN protocol you would use: 11-bit versus 29-bit ID, number of bytes of data, and bandwidth. Clearly describe what is in the ID and how the data is formatted.

**14.11** A CAN system has a baud rate of 100,000 bits/sec, 29-bit ID, and three bytes of data per frame. Assuming there is no bit-stuffing, what is the maximum bandwidth of this network, in bytes/s.

**14.12** A CAN system has a baud rate of 200,000 bits/sec, 11-bit ID, and five bytes of data per frame. Assuming there is no bit-stuffing, what is the maximum bandwidth of this network, in bytes/s.

**14.13** Consider a situation where four microcontrollers are connected together using a CAN network. Assume for this question that each frame contains 100 bits. Also assume the baud rate is 100,000 bits/sec; therefore, it takes 1 ms to send a frame. Initially, the CAN controllers are initialized (i.e., all computers have previously executed **CAN\_Open**).

At time = 0	Computer A calls <b>CAN_Send</b> with ID=1000
At time = 300 us	Computer B calls <b>CAN_Send</b> with ID=800
At time = 500 us	Computer C calls <b>CAN_Send</b> with ID=900
At time = 700 us	Computer D calls <b>CAN_Send</b> with ID=600

Specify the time sequence in which the four frames occur on the CAN network. Clearly define the begin and end times when each message is visible on the CAN network.

**14.14** In a CAN network, what is the purpose of the CRC field? That is, what is CRC used for?

**D14.15** The objective of this exercise is to design a communication network using four single-chip microcomputers. The microcomputers are connected together by fiber-optic cables. The fiber-optic channel is virtually noise-free. The asynchronous simplex serial communication channels link the four computers in a circle. Each 8-bit message consists of a 2-bit source address, a 2-bit destination address, and 4 bits of data. The message data structure has three fields. In memory, messages are stored as 3 bytes, but when transmitted they are compressed into 1 byte.

```
struct message{
    unsigned char source;           // 0,1,2,3 computer that sent the message
    unsigned char destination;     // 0,1,2,3 computer that receives the message
    unsigned char information;}; // 0-15 data part of the message
typedef struct message messageType;
```

Seven steps occur as computer 0 sends “10” to computer 2:

The main program on computer 0 creates a message and passes it to its interrupt software—for example,

```
int SayHi(void){ messagetype Hello;
    Hello.destination=2;
    Hello.information=10;
    return(Send(&Hello)); } // Send knows the source is 0
```

The interrupt software on computer 0 transmits the message to computer 1

The interrupt software on computer 1 receives the message and notices the destination address does not match its computer number (DIP switch)

The interrupt software on computer 1 retransmits the message to computer 2

The interrupt software on computer 2 receives the message and notices the destination address does match its computer number (DIP switch)

The interrupt software on computer 2 passes the message to the main program on computer 2

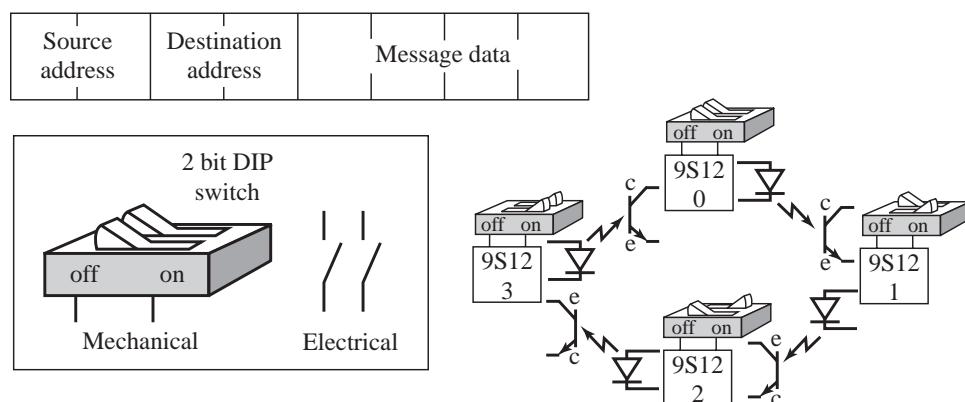
The main program on computer 2 accepts the message—for example,

```
void WaitFor(void){ messagetype msg;
    while(Accept(&msg)); } // waits for message
    printf("%d %d\n",msg.source,msg.information);}
```

Each microcomputer has a 2-bit DIP switch specifying its computer number. You may assume that each computer has a unique 2-bit address. The electric switch being on ( $0\ \Omega$ ) signifies a zero address bit, and the electric switch being off (open) signifies a one address bit (e.g., both DIP switches in computer 0 are on ( $0\ \Omega$ ) (Figure 14.22).

**Figure 14.22**

A ring network.



Each of the four fiber-optic cables contains a LED and a light-sensitive transistor. The LED operating point (shines light into the cable) is  $V = 2.5 \text{ V}$ ,  $I = 8 \text{ mA}$ . When light is in the cable, the transistor voltage  $V_{ce}$  is less than  $0.5 \text{ V}$  for any  $I_{ce}$  less than  $5 \text{ mA}$ . If the LED current is zero, then the cable is dark and the transistor is off ( $I_{ce} = 0$ ). The fiber-optic cable can handle baud rates up to  $2 \text{ Mb/s}$ . Choose the fastest baud rate for this interrupt-driven asynchronous serial communication. You will not write the main program, but you will write subroutines executed by the main program to SEND and ACCEPT messages. Your network software must be interrupt-driven.

- Show the hardware network interface for one single-chip microcomputer system. Except for the setting of the DIP switch, the network hardware/software is the same for all four computers. Please label chip numbers, but not pin numbers. Be careful how the systems are grounded to each other. If this is computer n, show the outgoing fiber-optic cable to computer n + 1 (diode) and the incoming fiber-optic cable from computer n – 1 (transistor).
- Show all the information that will be located in the RAM. The globals in RAM are unspecified on power up, so you must initialize them in the ritual in part c. Give careful thought as to the most appropriate data structures.
- Give the Ritual, Send, and Accept functions along with the interrupt handlers required. You may use FIFO queues by giving the global data structure definitions and the function prototypes without showing the FIFO function definitions. The same network communication software will be placed in each system (you can read the computer address from the DIP switches connected to an input port). The Ritual will initialize all data structures and configure the microcomputer. You will write two functions (Send, Accept) that will be called by the main program (you don't write the main program). The subroutine Send will initiate a send message communication. The prototype is shown above. Send returns right away, with the result equal to 0 if all is well so far but returns a 1 if the message cannot be sent (e.g., FIFO full). Use comments to explain how and why an error occurred. The subroutine Accept will check to see if a message has been received for this computer. The return value (RegD) is 0 if a message is ready and 1 if no incoming message is currently ready. If a message is available, then it returns by reference the source, destination, and information fields as illustrated in the above prototype. Whether or not an incoming message is available, Accept returns right away. The interrupt processes will perform the serial I/O operations. You need not poll for 0s and 1s. *Good interrupt software contains no gadfly loops.*
- What is the bandwidth of your network? Specify units and show the equation that justifies your answer.

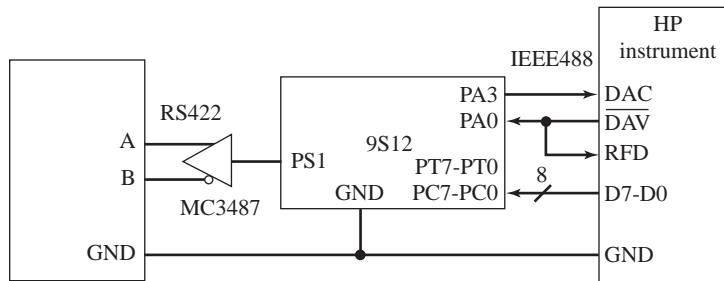
**D14.16** The objective of this exercise is to design a microcomputer-based IEEE488 to RS422 simplex converter (Figure 14.23). Your single-chip microcomputer system will perform handshaked parallel input from a IEEE488 output device, buffer it in an internal FIFO, then transmit the data on an asynchronous RS422 simplex serial channel. The serial protocol is 8-bit data, one stop, no parity, and 300 bits/s baud rate. The input bandwidth can vary from 0 to 1000 bytes/s, with an average of 10 bytes/s. The internal FIFO will allow temporarily

the input bandwidth to exceed the output bandwidth. The IEEE488 sequence of events to transmit 1 byte is:

1. Your microcomputer signals it is ready for the next data by making RFD = 1;
2. Eventually the IEEE488 output device will provide the 8-bit data and make  $\overline{DAV} = 0$ ;
3. Your microcomputer makes RFD = 0;
4. Your microcomputer should read the data, put it into the FIFO, then make DAC = 1;
5. Eventually the IEEE488 output device will make  $\overline{DAV} = 1$  and remove the data;
6. Your microcomputer makes DAC = 0.

**Figure 14.23**

An IEEE488 to RS422 interface.



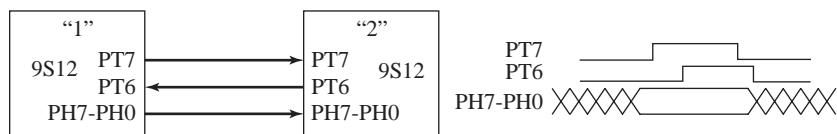
We can simplify this dedicated interface by connecting the  $\overline{DAV}$  to RFD, skipping Steps 1 and 3:

2. Eventually the IEEE488 output device will provide the 8-bit data and make  $\overline{DAV} = 0$ ;
4. Your microcomputer should read the data, put it into the FIFO, then make DAC = 1;
5. Eventually the IEEE488 output device will make  $\overline{DAV} = 1$  and remove the data;
6. Your microcomputer makes DAC = 0.
  - a) What should your microcomputer do when the FIFO is full? Why?
  - b) Show all the software including the interrupt handlers, rituals, main program, reset vector, interrupt vector, and ORG statements that place the code into the appropriate addresses on the *single-chip* microcomputer. You may use a FIFO without showing its implementation if you explain where in memory the data, pointers, and subroutines reside.

**D14.17** The objective of this exercise is to develop a message-passing facility that spans across two computers (Figure 14.24). A fully interlocked synchronization method will be implemented. The message is a simple 8-bit byte that is passed from the output port of Computer 1 to the input port of Computer 2. The following hardware connections are fixed. You may assume both computers initialize together, where Computer 1 initializes its PT7 to 0 and Computer 2 initializes its PT6 to 0.

**Figure 14.24**

A handshaking parallel port interface.



When Computer 1 wishes to transmit to Computer 2, it puts the data first on its Port H outputs. Next, Computer 1 makes a rising edge on PT7, signifying that new data are available. Computer 1 will then wait for a rising edge on PT6, signifying that the data have been accepted. Next, Computer 1 makes its PT7 low. Last, it waits for a low signal on its PT6 input line.

- a) Write the software for the transmitting Computer 1. The transmission will occur in the background using input capture interrupts. You may use a FIFO queue without writing the three routines: `InitFifo`, `PutFifo`, and `GetFifo`. You may ignore FIFO full errors. There are three routines you will write.

1. `void Ritual(void)` that clears the FIFO, makes the PT7 output low, and arms/enables the input capture interrupts as appropriate;

2. `void PutMsg(unsigned char data)` function that is called by the main program (that you do not write); if a current message is in progress, then these data are entered in the FIFO; if there is no current message in progress, then one is started with the new data;
3. `void IC6Handler(void)` interrupt handler called by the input capture on PT6.

When Computer 2 wishes to receive from Computer 1, it first waits for a rising edge on its PT7 input. Then it gets the data from its Port H inputs. Next, Computer 2 makes a rising edge on its PT6 output, signifying that the data has been accepted. Computer 2 will then wait for a falling edge on its PT7 input. Last, it makes a low signal on its PT6 line.

- b)** Write the software for the receiving Computer 2. The reception will occur in the background using input capture interrupts. You may use a FIFO queue without writing the three routines: `InitFifo`, `PutFifo`, and `GetFifo`. You may ignore FIFO-full errors. There are three routines you will write.
1. `void Ritual(void)` that clears the FIFO, makes the PT6 output low, and arms/enables the input capture interrupts as appropriate;
  2. `unsigned char data GetMsg(void)` function that is called by the main program (that you do not write); if the FIFO is empty, then this routine will call the `get()` routine over and over until data are available;
  3. `void IC7Handler(void)` interrupt handler called by the input capture on PT7.

- D14.18** Consider a CAN network that has a variable transmission rate, where the time in between messages can vary from  $t_{\min}$  to  $t_{\max}$ . Consider a particular listener on the network that will receive all messages (no filter). If the receive queue in this listener already has five unread messages, then additional CAN transmissions will stall on the network, because this receiver cannot acknowledge any more messages. In order to prevent stalling, this listener requires a real-time interface. The interface latency in this case is defined as the time between the Receive CAN Flag (RXF) being set and the execution of the CAN interrupt 38. You may assume executing each CAN ISR will quickly read and remove all messages in its receive queue (clearing RXF). What is the upper bound on the interface latency?

## 14.7 Lab Assignments

**Lab 14.1** The overall goal of this lab is to design, implement, and test a peer-to-peer communication system. Peer-to-peer means people on two computers communicate without the people on the other computers seeing the information. The system must use a ring-connected RS232 serial channel (Figure 14.4), must use interrupt-driven I/O, and must have a layered software configuration. The lowest-level software performs serial I/O. The middle-level software sends message packets from one computer to another. The highest-level software interfaces with the human operator (keypad/LCD) and provides a mechanism to create a peer-to-peer connection. In a layered system, software in one layer can call routines only within that layer or the layer immediately below it. You need a way to see who is on the network and a way to request/accept or terminate connection between two operators. Local operator input/output will occur via a keypad and LCD. You may assume all nodes on the system are willing to cooperate and will not perform malicious activity. On the other hand, it is possible that another computer on the network may not be plugged in, or the network connection may be broken.

The communication system between two or more microcomputers will be designed in three layers. The first layer, the physical layer is implemented by the SCI hardware and the interrupt-driven device driver. The second layer may consist of a simplified binary synchronous communication protocol (BSC). At this level, message packets will be transmitted between the two machines. Possible formats of the *control-code packet* and the *data packet* are shown in Figure 14.25. This *control-code packet* contains exactly six bytes.

STX—start of text: precedes text block (ASCII \$02).

Source—ASCII code of the source computer (ASCII \$31 to \$39).

Dest—ASCII code of the destination computer (ASCII \$31 to \$39).

Code—one of the following two control codes:

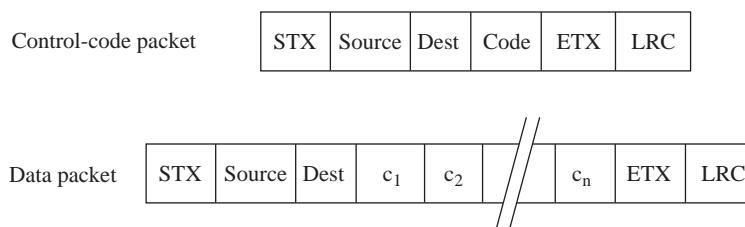
ACK—affirmative acknowledge: last block received correctly (ASCII \$06).

NAK—negative acknowledge: last block was received in error (ASCII \$15).

ETX—end of text block: ETX is followed by the error checking byte (ASCII \$03).

LRC—longitudinal redundancy check, error check byte.

**Figure 14.25**  
Control and data  
packets.



In the data packet, the data  $c_1, c_2, \dots, c_n$  are ASCII characters that constitute the information being sent from source to destination. It is OK to limit message sizes to a maximum of 20 bytes. The destination computer will respond with an ACK control packet if the message was received properly and will respond with a NAK if there are any framing, overrun, noise, or LRC errors. The transmitter will send a message and “stop and wait” for either an ACK or a NAK. If an ACK is received, then it can continue. If a NAK is received or if no response is received after a reasonable delay, then the message is retransmitted. Because there is a ring physical channel, there is no possibility of a collision. You must handle the situations when the destination computer does not exist or when the ring is broken. To solve this fault you will need some time-out mechanism to retransmit the packet if an ACK is not received in some reasonable time. You should choose an upper limit (e.g., three) on the number of times a packet is retransmitted. After three tries an error is reported to the operator.

The highest level will be a keypad interpreter and an LCD display. The LCD should show interactive feedback to the operator creates messages to be sent and should display messages received.

**Lab 14.2** The objective of this lab is to design a distributed temperature monitoring system using a CAN network. Each microcontroller node will consist of a temperature data acquisition system, an LCD display, and a CAN network interface. There are three software threads on each node. The producer thread will periodically measure temperature and send its local temperature to the other nodes on the network. The CAN identifier will specify the location (node number) and type (always temperature). The data field will contain the temperature as a decimal fixed-point number. Both data acquisition and network transmission will be performed in the background. The consumer thread, also running in the background, will accept messages from the other nodes and calculate the average and maximum temperature of the room. These data will be transferred to foreground, and the main program will display local, average, and maximum temperature. Choose a CAN bandwidth (e.g., 250,000 bps) and leave it as a constant. Start with a data acquisition and transmission rate so slow (e.g., 1/sec) that collisions will be rare. Then, increase the rate until the maximum bandwidth is reached.

**Lab 14.3** The objective of this lab is to design a point-to-point ZigBee network using the Xbee module. Each microcontroller will have a keypad and a two-line LCD display. When the operator types on the keypad, characters are displayed on one line of the LCD. When the operator hits the send key, those characters are sent and eventually displayed on the second line of the LCD of the other microcontroller. Debug the system in a bottom-up manner. Write simple main programs in the Rx and Tx systems to test the low-level functionality of the interface. Add appropriate debugging instruments, such as profiles and dumps. Connect unused pins from both devices to a single logic analyzer and record a thread profile (which program runs when) as a stream of data is passed from the Tx system to the Rx system. Write a main program to measure maximum bandwidth of your channel without using hardware flow control. Furthermore, you should determine which component limits bandwidth. Modify the transmitter so it outputs pseudo data as fast as possible. Modify the receiver so it does not display data on the LCD. Rather, the receiver will check for this pattern of characters and count the number of errors. Add minimally intrusive debugging instruments to determine if and where data is lost. Measure the maximum range of your system. In particular, find the maximum distance where the system performs reliable communication.

# 15 Digital Filters

## Chapter 15 objectives are to:

- ❖ Introduce basic principles involved in digital filtering
- ❖ Define the Z transform and use it to analyze digital filters
- ❖ Design simple low-pass, frequency-reject, and high digital filters
- ❖ Discuss the effect of non-real-time sampling on digital filter error
- ❖ Develop digital filter implementations

The goal of this chapter is to provide a brief introduction to digital signal processing (DSP). DSP includes a wide range of operations such as digital filtering, event detection, frequency spectrum analysis, and signal compression/decompression. Similar to the goal of analog filtering, a digital filter will be used to improve the S/N ratio in our data. The difference is that a digital filter is performed in software on the digital data sampled by the ADC. The particular problem addressed in several ways in this chapter is removing 60-Hz noise from the signal. Like the control systems and communication systems discussed in the last two chapters, we will provide just a brief discussion to the richly developed discipline of DSP. Again, this chapter focuses mostly on the implementation on the embedded microcomputer. Event detection is the process of identifying the presence or absence of particular patterns in our data. Examples of this type of processing include optical character readers, waveform classification, sonar echo detection, infant apnea monitors, heart arrhythmia detectors, and burglar alarms. Frequency spectrum analysis requires the calculation of the discrete Fourier transform (DFT)<sup>1</sup>. Like the regular Fourier transform, the DFT converts a time-dependent signal into the frequency domain. The difference between a regular Fourier transform and the DFT is that the DFT performs the conversion on a finite number of discrete time digital samples to give a finite number of points at discrete frequencies. Data compression and decompression are important aspects in high-speed communication systems. Although we will not specifically address the problems of event detection, DFT, and compression/decompression in this book, these DSP operations are implemented using similar techniques as the digital filters that are presented in this chapter. Our goal for this chapter is to demonstrate that fairly powerful digital signal-processing techniques can be implemented on computers of modest performance like the 9S12. The limitation

<sup>1</sup>A fast algorithm to calculate the DFT is called the fast Fourier transform (FFT).

of single-chip computers like the 9S12 is not complexity but rather execution speed. If you have a DAS or control system with unwanted 60-Hz noise, then the techniques described in this chapter provide an effective means for removing this noise.

## 15.1 Basic Principles

The objective of this section is to introduce simple digital filters. Let  $x_c(t)$  be the continuous analog signal to be digitized.  $x_c(t)$  is the analog input to the ADC. If  $f_s$  is the sample rate, then the computer samples the ADC every  $T$  seconds ( $T = 1/f_s$ ). Let  $\dots, x(n), \dots$ , be the ADC output sequence, where

$$x(n) = x_c(nT) \quad \text{with } -\infty < n < +\infty \quad (1)$$

There are two types of approximations associated with the sampling process. Because of the finite precision of the ADC, amplitude errors occur when the continuous signal  $x_c(t)$  is sampled to obtain the digital sequence  $x(n)$ . The second type of error occurs because of the finite sampling frequency. The Nyquist theorem states that the digital sequence  $x(n)$  properly represents the DC to  $(\frac{1}{2})f_s$  frequency components of the original signal  $x_c(t)$ . Two important assumptions are necessary to make when using digital signal processing:

1. We assume the signal has been sampled at a fixed and known rate  $f_s$
2. We assume aliasing has not occurred

We can guarantee the first assumption by using a hardware clock to start the ADC at a fixed and known rate. A less expensive but not as reliable method is to implement the sampling routine as a high-priority periodic interrupt process. By establishing a high priority of the interrupt handler, we can place an upper bound on the interrupt latency, guaranteeing that ADC sampling is occurring at an almost fixed and known rate.

To verify aliasing has not occurred, we can observe the ADC input with a spectrum analyzer to prove there are no significant signal components above  $(\frac{1}{2})f_s$ . “No significant signal components” is defined as having an ADC input voltage  $|Z|$  less than the ADC resolution,  $\Delta z$ ,

$$|Z| \leq \Delta z \quad \text{for all } f \geq \frac{1}{2} f_s$$

In a manner similar to the DAS design process developed in Chapter 12, we could impose a more strict constraint of  $|Z| \leq \frac{1}{2} \Delta z$ . The  $\frac{1}{2}$  in the expression  $\frac{1}{2} \Delta z$  is not the result of fundamental theorems, but rather a safety factor added during the design so that the effect of aliasing will not be present in the digital samples.

A *causal* digital filter calculates  $y(n)$  from  $y(n-1), y(n-2), \dots$  and  $x(n), x(n-1), x(n-2), \dots$ . Simply put, a causal filter cannot have a nonzero output until it is given a nonzero input. The output of a causal filter,  $y(n)$ , cannot depend on future data [e.g.,  $y(n+1), x(n+1)$ ].

A *linear* filter is constructed from a linear equation. A *nonlinear* filter is constructed from a nonlinear equation. One example of a nonlinear filter is the median. To calculate the median of three numbers, one first sorts the numbers according to magnitude, then chooses the middle value.

A *finite-impulse response* (FIR) filter relates  $y(n)$  only in terms of  $x(n), x(n-1), x(n-2), \dots$ . In the next section we will determine the gain and phase response of four

simple digital filters. These examples are presented in Equations 2 to 5. Equations 2 to 4 describe simple averaging FIR filters:

$$y(n) = \frac{x(n) + x(n - 1)}{2} \quad (2)$$

$$y(n) = \frac{x(n) + x(n - 1) + x(n - 2) + x(n - 3) + x(n - 4) + x(n - 5)}{6} \quad (3)$$

$$y(n) = \frac{x(n) + x(n - 3)}{2} \quad (4)$$

An **infinite-impulse response** (IIR) filter relates  $y(n)$  in terms of both  $x(n)$ ,  $x(n - 1)$ , ..., and  $y(n - 1)$ ,  $y(n - 2)$ . . . . Equation 5 describes a simple IIR filter:

$$y(n) = \frac{y(n - 1) + x(n)}{2} \quad (5)$$

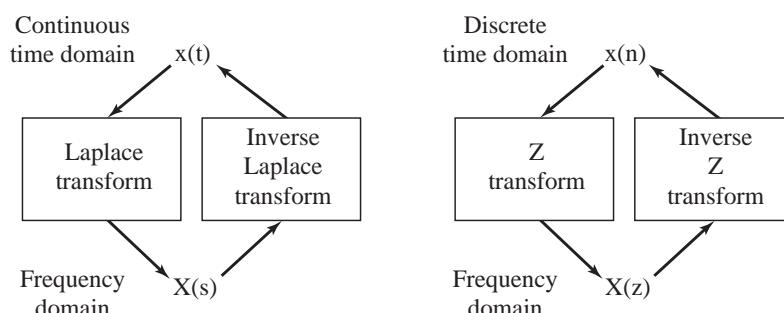
One way to analyze linear filters is the Z transform. Equation 6 gives the definition of the Z transform:

$$X(z) = Z[x(n)] \equiv \sum_{n=-\infty}^{\infty} x(n)z^{-n} \quad (6)$$

The Z transform is similar to other transforms. In particular, consider the Laplace transform, which converts a continuous time-domain signal  $x(t)$  into the frequency domain  $X(s)$ . In the same manner, the Z transform converts a discrete time sequence  $x(n)$  into the frequency domain  $X(z)$  (Figure 15.1).

**Figure 15.1**

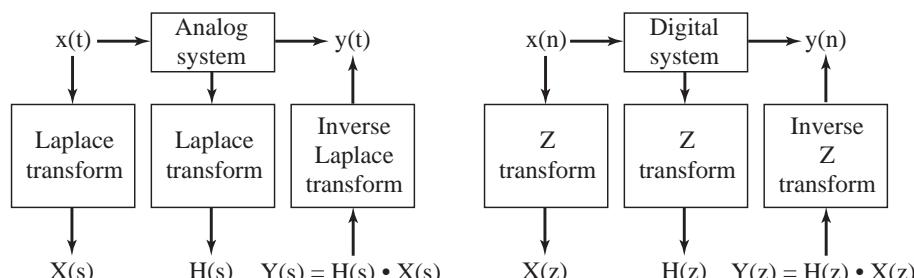
A transform is used to study a signal in the frequency domain.



The input to both the Laplace and Z transforms are infinite time signals, having values at times from  $-\infty$  to  $+\infty$ . The frequency parameters  $s$  and  $z$  are complex numbers, having a real and imaginary part. In both cases we apply the transform to study linear systems. In particular, we can describe the behavior (gain and phase) of an analog system using its transform,  $H(s) = Y(s)/X(s)$ . In this same way we will use the  $H(z)$  transform of a digital filter to determine its gain and phase response (Figure 15.2).

**Figure 15.2**

A transform also can be used to study a system in the frequency domain.



For an analog system we can calculate the gain by taking the magnitude of  $\mathbf{H}(s)$  at  $s = j2\pi f$ , for all frequencies  $f$  from  $-\infty$  to  $+\infty$ . The phase will be the angle of  $\mathbf{H}(s)$  at  $s = j2\pi f$ . If we were to plot the  $\mathbf{H}(s)$  in the  $s$  plane, the  $s = j2\pi f$  curve is the entire  $y$  axis. For a digital system we will calculate the gain and phase by taking the magnitude and angle of  $\mathbf{H}(z)$ . Because of the finite sampling interval, we will only be able to study frequencies from DC to  $(\frac{1}{2})f_s$  in our digital systems. If we were to plot the  $\mathbf{H}(z)$  in the  $z$  plane, the  $z$  curve representing the DC to  $(\frac{1}{2})f_s$  frequencies will be the unit circle.

We will begin by developing a simple, yet powerful rule that will allow us to derive the  $\mathbf{H}(z)$  transforms of most digital filters. Let  $m$  be an integer constant. One can use the definition of the Z transform to prove that

$$\begin{aligned}
 \mathbf{Z}[x(n-m)] &= \sum_{n=-\infty}^{\infty} z^{-n} x(n-m) \\
 &= \sum_{p+m=-\infty}^{p+m=+\infty} z^{-p-m} x(p) \quad p = n - m, n = p + m \\
 &= \sum_{p=-\infty}^{p=\infty} z^{-p-m} x(p) \quad m \text{ is a constant} \\
 &= z^{-m} \sum_{p=-\infty}^{p=+\infty} z^{-p} x(p) \quad m \text{ is a constant} \\
 &= z^{-m} \mathbf{Z}[x(n)] \tag{7}
 \end{aligned}$$

For example, if  $\mathbf{X}(z)$  is the Z transform of  $x(n)$ , then  $z^{-2} \cdot \mathbf{X}(z)$  is the Z transform of  $x(n-2)$ . To find the Z transform of a digital filter, we take the transform of both sides of the linear equation and solve for

$$\mathbf{H}(z) = \frac{\mathbf{Y}(z)}{\mathbf{X}(z)} \tag{8}$$

To find the response of the filter, let  $z$  be a complex number on the unit circle

$$z \equiv e^{j2\pi f/f_s} \quad \text{for } 0 \leq f < \frac{1}{2}f_s \tag{9}$$

or

$$z = \cos(2\pi f/f_s) + j \sin(2\pi f/f_s) \tag{10}$$

Let

$$\mathbf{H}(f) = \mathbf{a} + \mathbf{b}j \quad \text{where } \mathbf{a} \text{ and } \mathbf{b} \text{ are real numbers} \tag{11}$$

The gain of the filter is the complex magnitude of  $\mathbf{H}(z)$  as  $f$  varies from 0 to  $(\frac{1}{2})f_s$ .

$$\text{Gain} \equiv |\mathbf{H}(f)| = \sqrt{\mathbf{a}^2 + \mathbf{b}^2} \tag{12}$$

The phase response of the filter is the angle of  $\mathbf{H}(z)$  as  $f$  varies from 0 to  $(\frac{1}{2})f_s$ .

$$\text{Phase} \equiv \text{angle}[\mathbf{H}(f)] = \tan^{-1} \frac{\mathbf{b}}{\mathbf{a}} \tag{13}$$

## 15.2 Simple Digital Filter Examples

In this section, we perform the straightforward calculations to determine the filter response (gain and phase) given the filter equation. The gain and phase of these four filters are plotted in Figures 15.3 and 15.4. Even though the four examples are very simple, they can be

used in actual applications. The first example, Equation 2, is a simple average of the current sample with the previous sample (Program 15.1).

$$y(n) = \frac{x(n) + x(n - 1)}{2} \quad (2)$$

The first step is to take the Z transform of both sides of Equation 2. The Z transform of  $y(n)$  is  $Y(z)$ , the Z transform of  $x(n)$  is  $X(z)$ , and the Z transform of  $x(n - 1)$  is  $z^{-1}X(z)$ . Since the Z transform is a linear operator, we can write

$$Y(z) = \frac{X(z) + z^{-1}X(z)}{2}$$

The next step is to rewrite the equation in the form of  $H(z) = Y(z)/X(z)$ . We then can use Equations 9 to 13 to determine the gain and phase response of this filter.

$$\begin{aligned} H(z) &\equiv \frac{Y(z)}{X(z)} = \frac{1 + z^{-1}}{2} \\ H(f) &= \frac{1 + e^{-j2\pi f/f_s}}{2} = \frac{1 + \cos(2\pi f/f_s) - j \sin(2\pi f/f_s)}{2} \\ \text{Gain} &\equiv |H(f)| = 0.5 \sqrt{\{1 + \cos(2\pi f/f_s)\}^2 + \{\sin(2\pi f/f_s)\}^2} \\ \text{Phase} &\equiv \text{angle}[H(f)] = \tan^{-1} \left[ \frac{-\sin(2\pi f/f_s)}{1 + \cos(2\pi f/f_s)} \right] \end{aligned}$$

### Program 15.1

Real time data acquisition with a simple digital filter, Equation 2.

```
// 9S12C32 C implementation, interrupts at a frequency of fs
unsigned short x[2],y;
// x[0] is x(n) current sample
// x[1] is x(n-1) previous sample
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20;          // ack OC5F
    TC5 = TC5+PERIOD;     // Executed every 1/fs
    x[1] = x[0];           // shift MACQ data
    x[0] = ADC_In(CHANNEL); // new data
    y = (x[0]+x[1])>>1;
}
void DAS_Init(void){ // start sampling
    asm sei             // make atomic
    ADC_Init();          // Program 11.13
    TIOS |= 0x20;         // enable OC5
    TSCR1 = 0x80;         // enable, 4 MHz TCNT
    TIE |= 0x20;          // Arm output compare 5
    TFLG1 = 0x20;         // Initially clear C5F
    TC3 = TCNT+50;        // First one right away
    x[0] = x[1] = 0;
    asm cli
}
```

**Observation:** Equation 2 is a simple low-pass filter, passing DC and rejecting 0.5  $f_s$ .

**Observation:** If the ADC is only 8 bits, whether to use char or short for x,y depends on how the compiler implements the mathematical calculations.

**Checkpoint 15.1:** If TCNT increments at 4 MHz, what value should PERIOD be to make the sampling rate 1000 Hz?

**Checkpoint 15.2:** If the sampling rate is 1000 Hz, what are gains of filter Equation 2 at DC, 250 Hz and 500 Hz?

The second simple filter, Equation 3, performs the running average of the last six ADC samples (Program 15.2). It uses a faster method to implement the MACQ.

### Program 15.2

Real-time data acquisition with a simple digital filter, Equation 3.

```
// 9S12C32 C implementation, interrupts at 360Hz
unsigned short n,x[12],y;
// there are two copies of X(n) data
// x[n] is x(n) current sample
// x[n-1] is x(n-1) sample 1/360 sec ago
// x[n-2] is x(n-2) sample 2/360 sec ago
// x[n-3] is x(n-3) sample 3/360 sec ago
// x[n-4] is x(n-4) sample 4/360 sec ago
// x[n-5] is x(n-5) sample 5/360 sec ago
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20; // ack OC5F
    TC5 = TC5+11111; // Executed at 360 Hz
    n++; if(n==12) n=6;
    x[n] = x[n-6] = ADC_In(CHANNEL); // new data
    y = (x[n]+x[n-1]+x[n-2]+x[n-3]+x[n-4]+x[n-5])/6;
}
void DAS_Init(void){ // start sampling
    asm sei // make atomic
    ADC_Init(); // Program 11.13
    n = 6; // rotates through 6,7,8,9,10,11
    TIOS |= 0x20; // enable OC5
    TSCR1 = 0x80; // enable, 4 MHz TCNT
    TIE |= 0x20; // Arm output compare 5
    TFLG1 = 0x20; // Initially clear C5F
    TC3 = TCNT+50; // First one right away
    asm cli
}
```

$$y(n) = \frac{x(n) + x(n-1) + x(n-2) + x(n-3) + x(n-4) + x(n-5)}{6} \quad (3)$$

Just like the last example, first we take the Z transform of both sides of Equation 3:

$$Y(z) = \frac{X(z) + z^{-1}X(z) + z^{-2}X(z) + z^{-3}X(z) + z^{-4}X(z) + z^{-5}X(z)}{6}$$

The next step is to rewrite the equation in the form of  $H(z) = Y(z)/X(z)$ .

$$H(z) \equiv \frac{Y(z)}{X(z)} = \frac{1 + z^{-1} + z^{-2} + z^{-3} + z^{-4} + z^{-5}}{6}$$

We then can use Equations 9 to 13 to determine the gain and phase response of this filter.

$$\begin{aligned} H(f) &= \frac{1 + e^{-j2\pi f/f_s} + e^{-j4\pi f/f_s} + e^{-j6\pi f/f_s} + e^{-j8\pi f/f_s} + e^{-j10\pi f/f_s}}{6} \\ &= [1 + \cos(2\pi f/f_s) + \cos(4\pi f/f_s) + \cos(6\pi f/f_s) + \cos(8\pi f/f_s) + \cos(10\pi f/f_s) \\ &\quad - j\{\sin(2\pi f/f_s) + \sin(4\pi f/f_s) + \sin(6\pi f/f_s) + \sin(8\pi f/f_s) + \sin(10\pi f/f_s)\}]/6 \end{aligned}$$

$$\begin{aligned}\text{Gain} &= |\mathbf{H}(f)| \\ &= 1/6 \cdot [\{1 + \cos(2\pi f/f_s) + \cos(4\pi f/f_s) + \cos(6\pi f/f_s) + \cos(8\pi f/f_s) + \cos(10\pi f/f_s)\}^2 \\ &\quad + \{\sin(2\pi f/f_s) + \sin(4\pi f/f_s) + \sin(6\pi f/f_s) + \sin(8\pi f/f_s) + \sin(10\pi f/f_s)\}^2]\end{aligned}$$

$$\begin{aligned}\text{Phase} &= \text{angle}[\mathbf{H}(f)] \\ &= \frac{\tan^{-1}[-\{\sin(2\pi f/f_s) + \sin(4\pi f/f_s) + \sin(6\pi f/f_s) + \sin(8\pi f/f_s) + \sin(10\pi f/f_s)\}]}{\{1 + \cos(2\pi f/f_s) + \cos(4\pi f/f_s) + \cos(6\pi f/f_s) + \cos(8\pi f/f_s) + \cos(10\pi f/f_s)\}}\end{aligned}$$

**Observation:** This implementation of the MACQ uses twice as much memory but does not require the data to be shifted for each sample.

**Observation:** When comparing the execution speed of two potential approaches, we need to measure it directly or observe the assembly listing generated by the compiler.

**Checkpoint 15.3:** If the sampling rate of Equation 3 is 360 Hz, use the Z transform to prove that the 60 Hz gain is zero.

**Observation:** Equation 3 is a multiple-notch filter rejecting  $(\frac{1}{6})f_s$ ,  $(\frac{1}{3})f_s$ , and  $(\frac{1}{2})f_s$ . In particular, if  $f_s$  is 360 Hz, then the digital filter in Equation 3 rejects 60, 120, and 180 Hz.

The third simple filter, Equation 4, performs the average of the current 8-bit ADC sample with the sample collected three sampling periods ago. Although this filter appears to be simple, we can use it to implement a low-Q 60-Hz notch digital filter (Program 15.3).

$$y(n) = \frac{x(n) + x(n - 3)}{2} \quad (4)$$

### Program 15.3

Real-time data acquisition with a simple digital filter, Equation 4.

```
// 9S12C32 C implementation, interrupts at 360Hz
unsigned char x[4],y; // 8-bit ADC
// x[0] is x(n) current sample
// x[1] is x(n-1) sample 1/360 sec ago
// x[2] is x(n-2) sample 2/360 sec ago
// x[3] is x(n-3) sample 3/360 sec ago
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20; // ack OC5F
    TC5 = TC5+11111; // Executed at 360 Hz
    x[3] = x[2]; // shift MACQ data
    x[2] = x[1];
    x[1] = x[0];
    x[0] = ADC_In(CHANNEL); // new 8-bit data
    y = (x[0]+x[3])>>1;
}
void DAS_Init(void){ // start sampling
    asm sei // make atomic
    ADC_Init(); // Program 11.13
    TIOS |= 0x20; // enable OC5
    TSCR1 = 0x80; // enable, 4 MHz TCNT
    TIE |= 0x20; // Arm output compare 5
    TFLG1 = 0x20; // Initially clear C5F
    TC3 = TCNT+50; // First one right away
    asm cli
}
```

Again we take the Z transform of both sides of Equation 4:

$$Y(z) = \frac{X(z) + z^{-3}X(z)}{2}$$

Next we rewrite the equation in the form  $H(z) = Y(z)/X(z)$ .

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1 + z^{-3}}{2}$$

We then can use Equations 9 to 13 to determine the gain and phase response of this filter.

$$H(f) = \frac{1 + e^{-j6\pi f/f_s}}{2} = \frac{1 + \cos(6\pi f/f_s) - j \sin(6\pi f/f_s)}{2}$$

$$\text{Gain} \equiv |H(f)| = 0.5\sqrt{\{1 + \cos(6\pi f/f_s)\}^2 + \{\sin(6\pi f/f_s)\}^2}$$

$$\text{Phase} \equiv \text{angle}[H(f)] = \tan^{-1}\left[\frac{-\sin(6\pi f/f_s)}{1 + \cos(6\pi f/f_s)}\right]$$

**Checkpoint 15.4:** If the sampling rate of Equation 4 is 360 Hz, use the Z transform to prove that the 60 Hz gain is zero.

**Observation:** Equation 4 is a double-notch filter rejecting  $(\frac{1}{6})f_s$  and  $(\frac{1}{2})f_s$ . In particular, if  $f_s$  is 360 Hz, then the digital filter in Equation 4 rejects 60 and 180 Hz.

When writing digital filters it is important to consider the precision of the both the variables and the calculations. In particular, a serious error will occur if overflow or underflow occurs during the calculations. Consider the calculation  $x[0]+x[3]$  performed in Program 15.4. If simple 8-bit addition were to be used, then an error will occur when intermediate calculation  $x[0]+x[3]$  goes above 255. Observing the gain versus frequency plot in Figure 15.3, we are confident that the final calculation  $(x[0]+x[3])>>1$  will fit into an 8-bit variable. To verify program correctness, we must look at the output of the compiler. The two listings in Program 15.4 were generated by the ImageCraft ICC11 and Metrowerks CodeWarrior compilers. These are proper implementations because these compilers first promote the 8-bit data into 16 bits, perform the addition and shift using 16-bit operations, then demote the result back to 8 bits when storing into  $y$ . The comments were added to clarify the operations.

<pre>; MC68HC11, Imagecraft ICC11 ; y=(x[0]+x[3])&gt;&gt;1; ldy #1 pshy ldab _x+3 ; 8-bit x[3] clra ; promote into RegD pshb ; save on stack psha ldab _x ; 8 bit x[0] clra ; promote into RegD tsy addd 0,y ; 16-bit x[0]+x[3] puly puly jsr __asrd ; 16-bit shift stab _y ; demote to 8 bit</pre>	<pre>; 9S12C32, Metrowerks CodeWarrior ; y=(x[0]+x[3])&gt;&gt;1; ldab x:3 ; 8-bit x[3] clra ; promote into RegD tfr D,X ; save in RegX ldab x ; 8-bit x[0] leax D,X ; 16-bit x[0]+x[3] asra rolb ; 16-bit shift stab y ; demote to 8 bit</pre>
---	--

#### Program 15.4

Compiler-generated assembly listings for the filter implementation.

The fourth simple filter, Equation 5, is an example of an infinite-impulse response filter. It performs the average of the current ADC sample with the last filter output. For this filter we follow the same procedure to determine its gain and frequency response. Notice in Figure 15.3 that the gain is larger than 1 for some frequencies. Therefore it will be important when implementing this filter to use a number system with more bits than the ADC precision. In other words, even if the ADC in Program 15.5 were to have only 8 bits, we will perform the filter using 16-bit precision to avoid overflow.

### Program 15.5

Real-time data acquisition with a simple IIR digital filter, Equation 5.

```
// 9S12C32 C implementation, interrupts at a frequency of fs
unsigned short x,y[2];
// y[0] is y(n) current filter output
// y[1] is y(n-1) filter output 1 sample ago
void interrupt 13 TC5handler(void){
    TFLG1 = 0x20;           // ack OC5F
    TC5 = TC5+PERIOD;      // Executed every 1/fs
    y[1] = y[0];            // shift MACQ data
    x = ADC_In(CHANNEL);   // new data
    y[0] = (x+y[1])/2;
}
void DAS_Init(void){ // start sampling
    asm sei          // make atomic
    ADC_Init();       // Program 11.13
    TIOS |= 0x20;     // enable OC5
    TSCR1 = 0x80;     // enable, 4 MHz TCNT
    TIE |= 0x20;      // Arm output compare 5
    TFLG1 = 0x20;     // Initially clear C5F
    TC3 = TCNT+50;   // First one right away
    asm cli
}
```

$$y(n) = \frac{y(n-1) + x(n)}{2} \quad (5)$$

$$Y(z) = \frac{z^{-1}Y(z) + X(z)}{2}$$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{2 - z^{-1}}$$

$$\begin{aligned} H(f) &= \frac{1}{2 - e^{-j2\pi f/f_s}} = \frac{1}{2 - \cos(2\pi f/f_s) + j \sin(2\pi f/f_s)} \\ &= \frac{2 - \cos(2\pi f/f_s) - j \sin(2\pi f/f_s)}{[2 - \cos(2\pi f/f_s)]^2 - [\sin(2\pi f/f_s)]^2} \end{aligned}$$

$$\begin{aligned} \text{Gain} &\equiv |H(f)| = \left| \frac{2 - \cos(2\pi f/f_s) - j \sin(2\pi f/f_s)}{[2 - \cos(2\pi f/f_s)]^2 - [\sin(2\pi f/f_s)]^2} \right| \\ &= \frac{\sqrt{(2 - \cos(2\pi f/f_s))^2 + (\sin(2\pi f/f_s))^2}}{[2 - \cos(2\pi f/f_s)]^2 - [\sin(2\pi f/f_s)]^2} \end{aligned}$$

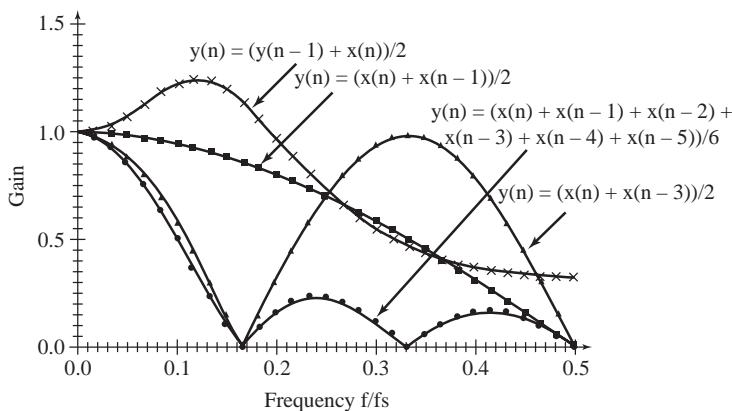
$$\text{Phase} \equiv \text{angle}[H(f)] = \tan^{-1} \left[ \frac{-\sin(2\pi f/f_s)}{2 - \cos(2\pi f/f_s)} \right]$$

**Checkpoint 15.5:** For some frequencies, the filter in Equation 5 has a gain larger than 1. What does that mean?

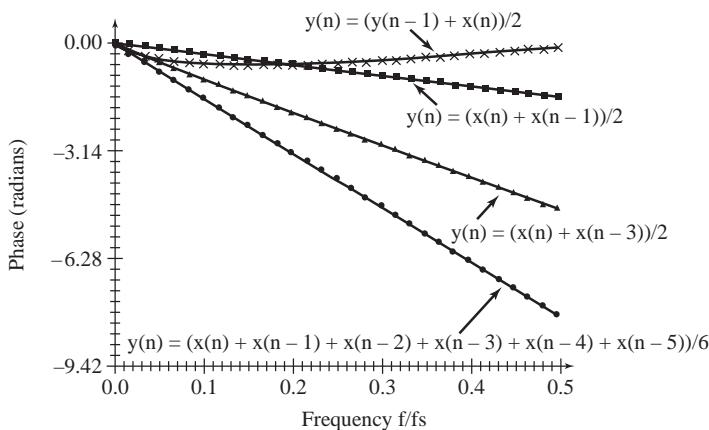
The response of the four linear digital filters is plotted in Figure 15.3; the phase response of the four linear digital filters is plotted in Figure 15.4.

**Figure 15.3**

Gain versus frequency response for four simple digital filters.

**Figure 15.4**

Phase versus frequency response for four simple digital filters.



A linear phase versus frequency response is desirable because a linear phase causes minimal waveform distortion. Conversely, a nonlinear phase response will distort the shape or morphology of the signal.

In general, if  $f_s$  is  $2 \cdot k \cdot f_c$  Hz (where  $k$  is any integer  $k \geq 2$ ), then the following is a  $f_c$  notch filter:

$$y(n) = \frac{x(n) + x(n - k)}{2} \quad (14)$$

Averaging the last  $k$  samples will perform a low-pass filter with notches. Let  $f_c$  be the frequency we wish to reject. We will choose the sampling at a multiple of this notch. That is, we choose  $f_s$  to be  $k \cdot f_c$  Hz (where  $k$  is any integer  $k \geq 2$ ), then the  $k$ -sample average filter will reject  $f_c$  and its harmonics:  $2f_c, 3f_c, \dots$ . If the number of terms  $k$  is large, the straightforward implementation of average will run slowly. Fortunately, this averaging filter can be rewritten as a function of the current sample  $x(n)$ , the sample  $k$  times ago  $x(n - k)$ , and the last filter output  $y(n - 1)$ .

$$y(n) = \frac{1}{k} \sum_{i=0}^{k-1} x(n - i) = \frac{x(n) - x(n - k)}{k} + y(n - 1) \quad (15)$$

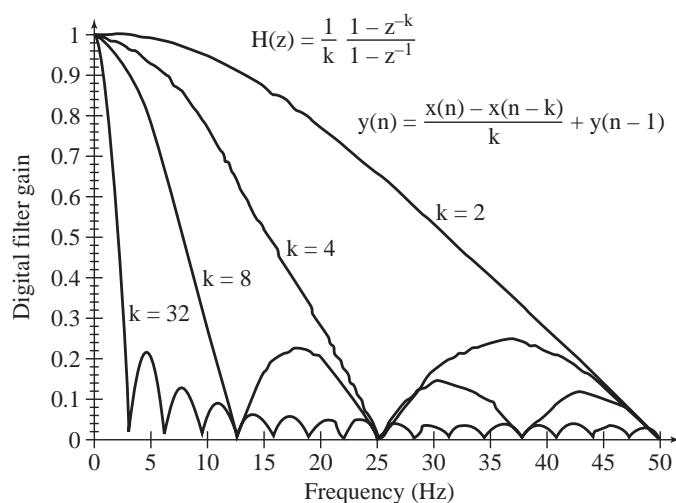
The second formulation looks like an IIR filter, but it is a FIR filter. The  $H(z)$  transfer function for this  $k$ -term averaging filter is

$$H(z) = \frac{1}{k} \frac{1 - z^{-k}}{1 - z^{-1}} \quad (16)$$

This class of digital low-pass filters can be implemented with a  $k + 1$  multiple access circular queue and a simple calculation. The gain of this class of filter is shown in Figure 15.5 for a sampling rate of 100 Hz

**Figure 15.5**

Gain versus frequency plot of four averaging low-pass filters.



### 15.3 Impulse Response

Another way to analyze digital filters is to consider the filter response to particular input sequences. Two typical test sequences are the step and the impulse.

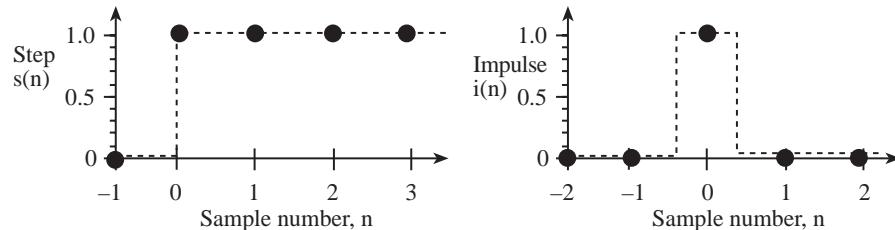
$$\begin{array}{ll} \text{Step} & \dots, 0, 0, 0, 1, 1, 1, 1, \dots \\ \text{Impulse} & \dots, 0, 0, 0, 1, 0, 0, 0, \dots \end{array}$$

The step is defined as (Figure 15.6)

$$\begin{aligned} s(n) &\equiv 0 && \text{for } n < 0 \\ &1 && \text{for } n \geq 0 \end{aligned}$$

**Figure 15.6**

Step and impulse inputs.



The step signal represents a sharp change (like an edge in a photograph).

Consider the following three digital filters:

$$\text{FIR} \quad y(n) = \frac{x(n) + x(n-1)}{2} \quad (2)$$

$$\text{IIR} \quad y(n) = \frac{y(n-1) + x(n)}{2} \quad (5)$$

$$\text{Median} \quad y(n) = \text{median}[x(n), x(n-1), x(n-2)] \quad (17)$$

The median can be performed on any odd number of data points by sorting the data and selecting the middle value. The median filter can be performed recursively or nonrecursively. For a nonrecursive median filter, the original data points are not modified. That is, the median is calculated each time without regard to the previous filter outputs. For example, a five-wide nonrecursive median filter takes as the filter output the median of  $\{x(n),$

$\mathbf{x(n-1), x(n-2), x(n-3), x(n-4)}$ } On the other hand, a recursive median filter replaces the sample point with the filter output. For example, a five-wide recursive median filter takes as the filter output the median of  $\{x(n), y(n-1), y(n-2), y(n-3), y(n-4)\}$ , where  $y(n-1), y(n-2) \dots$  are the previous filter outputs. A nonrecursive three-wide median filter is implemented in Program 15.6.

**Observation:** A median filter can be applied in systems that have impulse or speckle noise. For example, if the noise occasionally causes one sample to be very different from the rest (like a speck on a piece of paper), then the median filter will completely eliminate the noise.

### Program 15.6

The median filter is an example of a nonlinear filter.

```
unsigned char median(unsigned char u1,
unsigned char u2,unsigned char u3){
unsigned char result;
if(u1>u2)
    if(u2>u3)      result = u2; // u1>u2,u2>u3          u1>u2>u3
    else
        if(u1>u3) result = u3; // u1>u2,u3>u2,u1>u3 u1>u3>u2
        else      result = u1; // u1>u2,u3>u2,u3>u1 u3>u1>u2
    else
        if(u3>u2)      result = u2; // u2>u1,u3>u2          u3>u2>u1
        else
            if(u1>u3) result = u1; // u2>u1,u2>u3,u1>u3 u2>u1>u3
            else      result = u3; // u2>u1,u2>u3,u3>u1 u2>u3>u1
    return(result);
}
unsigned char x[3],y;
// x[0] is x(n) the current sample
// x[1] is x(n-1) the sample 1/fs ago
// x[2] is x(n-2) the sample 2/fs ago
void sample(void){
    x[2] = x[1];           // shift MACQ data
    x[1] = x[0];
    x[0] = ADC_In(0); // new data from channel 0
    y = median(x[0],x[1],x[2]);
}
```

The step responses of the three filters are:

FIR	..., 0, 0, 0, 0.5, 1, 1, 1 ...
IIR	..., 0, 0, 0, 0.5, 0.75, 0.88, 0.94, 0.97, 0.98, 0.99 ...
Median	..., 0, 0, 0, 0, 1, 1, 1, 1 ...

Except for the delay, the median filter passes a step without error (Figure 15.7).

The impulse represents a noise spike (like spots on a Xerox copy). The definition of impulse is (Figure 15.6).

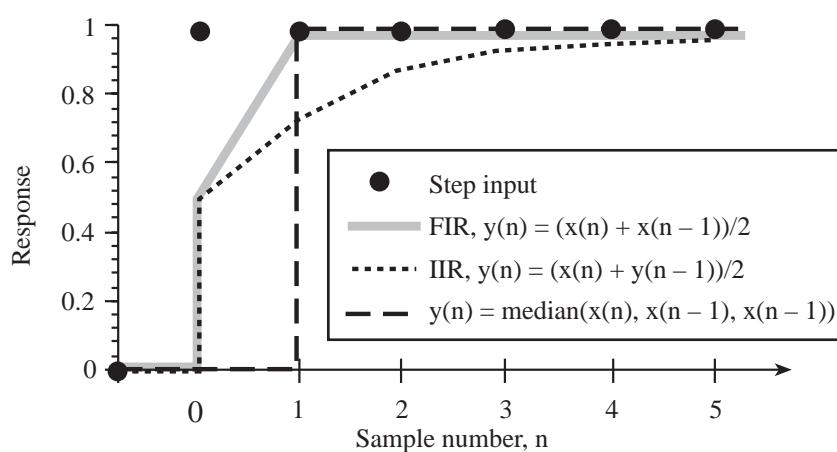
$$\begin{aligned} i(n) &\equiv 0 & \text{for } n \neq 0 \\ &1 & \text{for } n = 0 \end{aligned}$$

The impulse response of a filter is defined as  $\mathbf{h(n)}$ . The impulse responses of the three filters are:

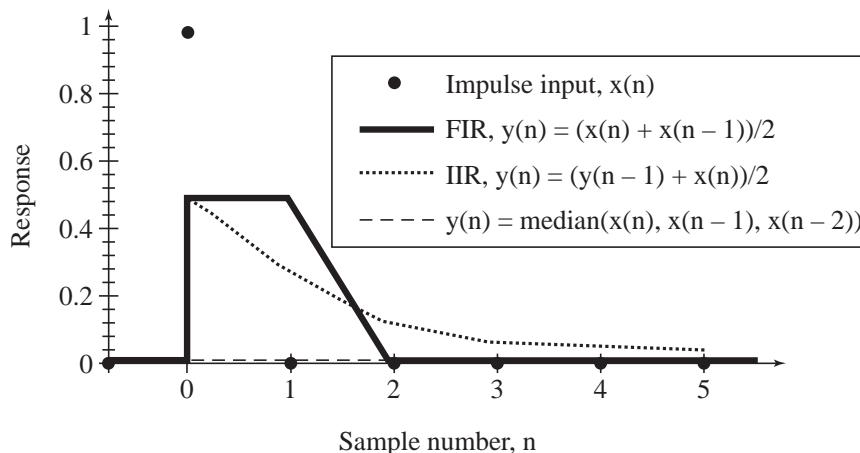
FIR	..., 0, 0, 0, 0.5, 0.5, 0, 0, 0 ...
IIR	..., 0, 0, 0, 0.5, 0.25, 0.13, 0.06, 0.03, 0.02, 0.01 ...
Median	..., 0, 0, 0, 0, 0, 0, 0, 0 ...

**Figure 15.7**

Step response of three simple digital filters.

**Figure 15.8**

Impulse response of three simple digital filters.



The impulse response  $\mathbf{h}(\mathbf{n})$  can also be used as an alternative to the transfer function  $\mathbf{H}(\mathbf{z})$ .  $\mathbf{h}(\mathbf{n})$  is sometimes called the *direct form*. A causal filter has  $\mathbf{h}(\mathbf{n}) = 0$  for  $\mathbf{n} < 0$ . For a causal filter,

$$\mathbf{H}(\mathbf{z}) = \sum_{\mathbf{n}=0}^{\infty} \mathbf{h}(\mathbf{n}) \mathbf{z}^{-\mathbf{n}} \quad (18)$$

For a FIR filter,  $\mathbf{h}(\mathbf{n}) = 0$  for  $\mathbf{n} \geq \mathbf{N}$  for some  $\mathbf{N}$ . Thus,

$$\mathbf{H}(\mathbf{z}) = \sum_{\mathbf{n}=0}^{\mathbf{N}-1} \mathbf{h}(\mathbf{n}) \mathbf{z}^{-\mathbf{n}} \quad (19)$$

The output of a filter can be calculated by convolving the input sequence  $\mathbf{x}(\mathbf{n})$  with  $\mathbf{h}(\mathbf{n})$ . For an IIR filter,

$$\mathbf{y}(\mathbf{n}) = \sum_{\mathbf{k}=0}^{\infty} \mathbf{h}(\mathbf{k}) \mathbf{x}(\mathbf{n} - \mathbf{k}) \quad (20)$$

For a FIR filter:

$$y(n) = \sum_{k=0}^{N-1} h(k) x(n-k) \quad (21)$$

## 15.4 High-Q 60-Hz Digital Notch Filter

There are two objectives for this example. The first goal is to show an example of a digital notch filter, and the second goal is to demonstrate the use of fixed-point mathematics. The 60-Hz noise is a significant problem in most DASs. The 60-Hz noise reduction can be accomplished by:

1. Reducing the noise source (e.g., shut off large motors)
2. Shielding the transducer, cables, and instrument
3. Implementing a 60-Hz analog notch filter
4. Implementing a 60-Hz digital notch filter

Consider again the analogy between the Laplace and Z transforms. When the  $H(s)$  transform is plotted in the s plane, we look for peaks (places where the amplitude  $H(s)$  is high) and valleys (places where the amplitude is low). In particular, we usually can identify zeros [ $H(s) = 0$ ] and poles [ $H(s) = \infty$ ]. In the same way we can plot the  $H(z)$  in the z plane and identify the poles and zeros. The analogies in Table 15.1 apply:

Analog condition	Digital condition	Consequence
Zero near $s = j2\pi f$ line	Zero near $z = e^{j2\pi f/f_s}$ circle	Low gain at the $f$ near the zero
Pole near $s = j2\pi f$ line	Pole near $z = e^{j2\pi f/f_s}$ circle	High gain at the $f$ near the pole
Zeros in complex conjugate pairs	Zeros in complex conjugate pairs	Output $y(t)$ is real
Poles in complex conjugate pairs	Poles in complex conjugate pairs	Output $y(t)$ is real
Poles in left half plane	Poles inside unit circle	Stable system
Poles in right half plane	Poles outside unit circle	Unstable system
Pole near a zero	Pole near a zero	High-Q response

**Table 15.1**

Analogy between the analog and digital filter design rules.

It is the 60-Hz digital notch filter that will be implemented in this example. The signal is sampled at  $f_s = 244.14$  Hz. We wish to place the zeros (gain = 0) at 60 Hz, thus (Figure 15.9):

$$\Theta = \pm 2\pi \cdot \frac{60}{f_s} = \pm 1.54416$$

The zeros are located on the unit circle at 60 Hz:

$$z_1 = \cos(\theta) + j \sin(\theta) \quad z_2 = \cos(\theta) - j \sin(\theta)$$

or

$$z_1 = 0.02663 + j 0.99965 \quad z_2 = 0.02663 - j 0.99965$$

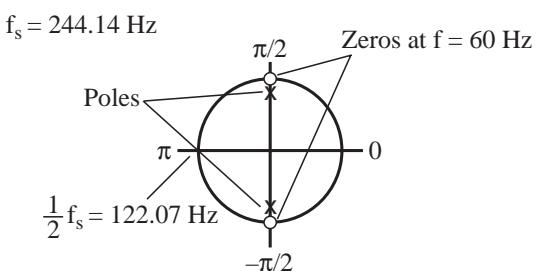
To implement a flat passband away from 60 Hz, the poles are placed next to the zeros, just inside the unit circle. Let  $\alpha$  define the closeness of the poles, where  $0 < \alpha < 1$ .

$$p_1 = \alpha z_1$$

$$p_2 = \alpha z_2$$

**Figure 15.9**

Pole-zero plot of a 60-Hz digital notch filter.



for  $\alpha = 0.95$

$$p_1 = 0.02530 + j 0.94966 \quad p_2 = 0.02530 - j 0.94966$$

The transfer function is

$$H(z) = \prod_{i=1}^k \frac{z - z_i}{z - p_i} = \frac{(z - z_1)(z - z_2)}{(z - p_1)(z - p_2)} \quad (22)$$

which can be put in standard form (i.e., with terms  $1, z^{-1}, z^{-2} \dots$ )

$$H(z) = \frac{1 - 2 \cos(\theta)z^{-1} + z^{-2}}{1 - 2\alpha \cos(\theta)z^{-1} + \alpha^2 z^{-2}}$$

or

$$H(z) = \frac{1 - 0.0532672z^{-1} + z^{-2}}{1 - 0.0506038z^{-1} + 0.9025z^{-2}}$$

The digital filter can be derived by taking the inverse Z transform of the  $H(z)$  equation

$$y(n) = x(n) - 2 \cos(\theta)x(n-1) + x(n-2) + 2\alpha \cos(\theta)y(n-1) - \alpha^2 y(n-2)$$

or

$$y_1(n) = x(n) - 0.0532672x(n-1) + x(n-2) + 0.0506038y_1(n-1) - 0.9025y_1(n-2)$$

To implement this filter in real time without floating-point hardware we rewrite Equation 50 using fixed-point mathematics:

$$y_2(n) = x(n) + x(n-2) + \frac{-14x(n-1) - 13y_2(n-1) - 231y_2(n-2)}{256}$$

It will be important to implement the software such that the intermediate calculations are performed in 16-bit arithmetic. Also, since this filter has a gain larger than 1 for some frequencies, the filter outputs must also be implemented with 16-bit variables.

Table 15.2 shows that the fixed-point calculations do not produce significant filter errors. There are two factors in choosing the fixed value (e.g., 256) in the fixed-point implementation. We choose a value large enough so that the fixed-point value closely approximates the floating-point value (e.g.,  $13/256 \approx 0.0506038$ ). The larger fixed-point value we choose, the better the approximation:

$$y_3(n) = x(n) + x(n-2) + \frac{-533x(n-1) + 506y_3(n-1) - 9025y_3(n-2)}{10000}$$

On the other hand, larger fixed-point values will require higher-precision integer mathematics. For an 8-bit microcomputer we would prefer to do 8-bit by 8-bit into 16-bit multiplies, 16-bit addition/subtraction, and 16-bit divided by 8-bit divides. For a 16-bit microcomputer

**Table 15.2**

Gain versus frequency responses for the floating point and fixed implementations.

$f$ (Hz)	$ Y_1(z)/X(z) $	$ Y_2(z)/X(z) $	$ Y_1(z)/X(z)  -  Y_2(z)/X(z) $
0	1.052	1.049	0.003
10	1.060	1.060	0.000
20	1.048	1.048	0.000
30	1.052	1.052	0.000
40	1.048	1.048	0.000
50	1.040	1.036	0.004
55	0.984	0.980	0.004
56	0.944	0.944	0.000
57	0.888	0.888	0.000
58	0.744	0.740	0.004
59	0.464	0.464	0.000
60	0.012	0.016	-0.004
61	0.396	0.404	-0.008
62	0.752	0.752	0.000
63	0.880	0.880	0.000
64	0.944	0.944	0.000
65	0.980	0.980	0.000
70	1.036	1.036	0.000
80	1.048	1.048	0.000
90	1.052	1.052	0.000
100	1.056	1.060	-0.004
110	1.056	1.056	0.000
120	1.052	1.052	0.000

we would prefer to do 16-bit by 16-bit into 32-bit multiplies, 32-bit addition/subtraction, and 32-bit divided by 16-bit divides. Notice that the multiply by  $2^n$  and divide by  $2^n$  operations are very simple to implement using the arithmetic shift operations.

For  $\alpha = 0.90$ , the filter becomes

$$y_4(n) = x(n) + x(n-2) + \frac{-14x(n-1) + 12y_4(n-1) - 207y_4(n-2)}{256}$$

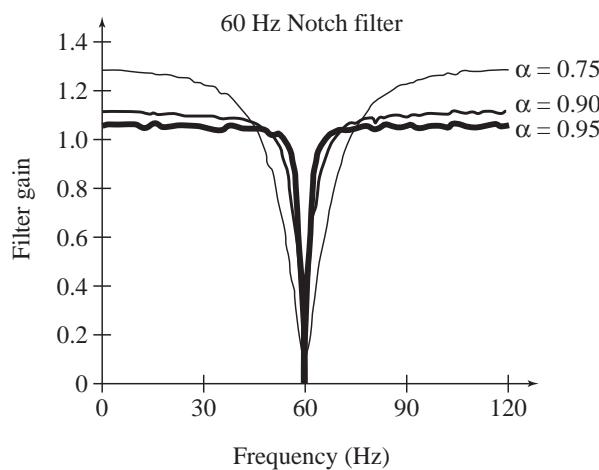
For  $\alpha = 0.75$ , the filter becomes

$$y_5(n) = x(n) + x(n-2) + \frac{-7x(n-1) + 5y_5(n-1) - 72y_5(n-2)}{128}$$

The gain of the three filters is plotted in Figure 15.10, showing the effect of  $\alpha$  on filter Q.

**Figure 15.10**

Gain versus frequency response of three 60-Hz digital notch filters.



Sometimes we can choose  $f_s$  and/or  $\alpha$  to simplify the digital filter equation. For example, if we choose  $f_s = 240$  Hz, then the  $\cos(\theta)$  terms become zero. If we choose  $\alpha = 7/8$  then the fixed-point digital filter becomes

$$y_6(n) = x(n) + x(n-2) - \frac{49 \cdot y_6(n-2)}{64}$$

Another consideration for this type of filter is the fact that the gain in the passbands is greater than 1. The DC gain can be determined two ways. The first method is to use the  $H(z)$  transfer equation and set  $z = 1$ . The  $H(z)$  transfer equation for the filter is

$$H(z) = \frac{1 + z^{-2}}{1 + (49/64)z^{-2}}$$

At  $z = 1$  this reduces to

$$\text{DC gain} = \frac{2}{1 + 49/64} = \frac{128}{64 + 49} = \frac{128}{113}$$

The second method to calculate DC gain operates on the filter equation directly. In the first step, we set all the  $x(n - k)$  terms in the filter to a single variable  $x$  and all the  $y(n - k)$  terms in the filter to a single variable  $y$ .

Next we solve for the DC gain, which is  $y/x$ .

$$y = x + x - \frac{49y}{64}$$

This method also calculates the DC gain to be 128/113. We can adjust the digital filter so that the DC gain is exactly 1 by prescaling the input terms [ $x(n)$ ,  $x(n - 1)$ ,  $x(n - 2)$  . . .] by 113/128 (Program 15.7).

$$y_7(n) = \frac{113 \cdot x(n) + 113 \cdot x(n - 2) - 98 \cdot y_7(n - 2)}{128}$$

<pre> ; 9S12C32 assembly     org RAM XN    ds 2 ; x(n) current XN1   ds 2 ; x(n-1) previous XN2   ds 2 ; x(n-2) 2 samples ago YN1   ds 2 ; y(n-1) previous YN2   ds 2 ; y(n-2) 2 samples ago YN    ds 2 ; y(n) current acc   ds 4 ; temporary 32-bit     org ROM COEF  dc.w 113,0,113,0,-98 TC5Handler     movb #\$20,TFLG1 ; ack     ldd TC5     addd #16667 ; 240Hz     std TC5     movw YN1,YN2 ; shift MACQ     movw YN,YN1     movw XN1,XN2     movw XN,XN1     jsr ADC_In     std XN ; new data     ldx #XN ; data     ldy #COEF ; coef </pre>	<pre> // 9S12C32 C implementation unsigned short x[3],y[3]; // x[0] is x(n) current sample // x[1] is x(n-1) 1 sample ago // x[2] is x(n-2) 2 samples ago // y[0] is y(n) current filter output // y[1] is y(n-1) filter out 1 ago // y[2] is y(n-2) filter out 2 ago void interrupt 13 TC5handler(void){     TFLG1 = 0x20;           // ack C5F     TC5 = TC5+16667;        // 240Hz     y[2]=y[1]; y[1]=y[0]; // shift MACQ     x[2]=x[1]; x[1]=x[0];     x[0] = ADC_In(CHANNEL); // new data     y[0]=(113*(x[0]+x[2])-98*y[2])&gt;&gt;7; } </pre>
--	---

*continued on p. 733*

*continued from p. 732*

```

ldd    #0
std   acc ; clear temporary
std   acc+2
ldaa #5      ; number of terms
loop  emacs acc ;acc=acc+{X}*{Y}
leax  2,x
leay  2,y
dbne  A,loop
ldy   acc
ldd   acc+2
;Y:D=113*(x[0]+x[2])-98*y[2]
ldx   #128
edivs
sty   YN
rti

```

### Program 15.7

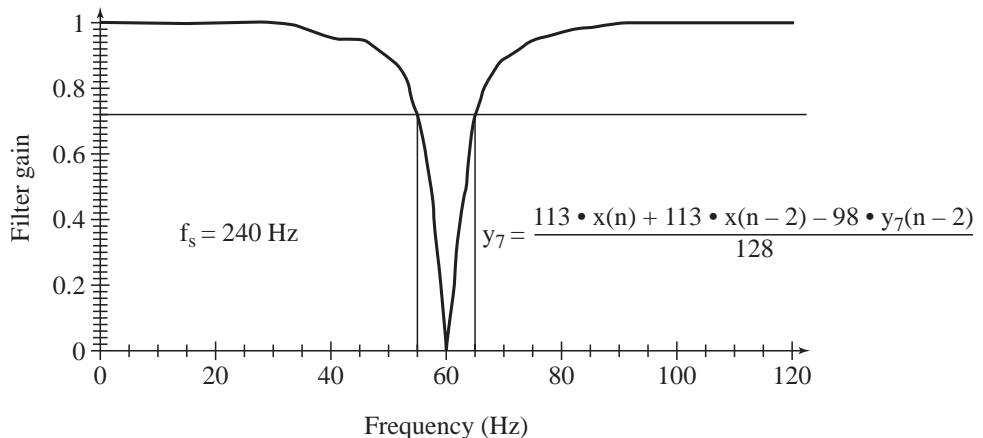
Real-time data acquisition with a 60-Hz digital notch filter.

**Checkpoint 15.6:** For the C language implementation, how large can the addition terms (before the divide by 128) get so that overflow does not occur?

**Checkpoint 15.7:** For the assembly language implementation, how large can the addition terms (before the divide by 128) get so that overflow does not occur?

Since the gain of this filter is always less than or equal to 1, the filter outputs will fit into the same precision as the filter input. The gain of this filter is shown in Figure 15.11.

**Figure 15.11**  
Gain versus frequency response of a 60-Hz digital notch filter scaled to have a DC gain of 1.



The Q of a digital notch filter is defined to be

$$Q = \frac{f_c}{\Delta f} \quad (23)$$

where  $f_c$  is the center or notch frequency and  $\Delta f$  is the frequency range, where its gain is below 0.707 of the DC gain. For the filter in Figure 15.11 the gains at 55 and 65 Hz are about 0.707, so its Q is 6.

## 15.5 Effect of Time Jitter on Digital Filters

The purpose of this analysis is to study the importance of “real time” in a real-time operating system. We define a “hard real-time” system as one that can guarantee that a process will complete a critical task within a certain specified range. On the other hand, we define a “soft real-time” system as one that assigns higher priority to critical functions. We will, in this analysis, derive specific consequences that result when the operating system delays service to a time-critical real-time process. In general, we can define **time-jitter**,  $\delta t$ , as the difference between when a periodic task is supposed to be run and when it is actually run. Let  $t_n$  be the time the software task is actually run, and let  $n\Delta t$  be the time it was supposed to be run; then the time-jitter at sample  $n$  is

$$\delta t_n = t_n - n\Delta t$$

For a real-time system with periodic tasks, we must be able to place an upper bound,  $k$ , on the time-jitter.

$$-k \leq t_n \leq +k \text{ for all } n$$

For a DAS it is often more important to control the time difference between periodic events rather than the absolute time itself. Let  $\Delta t_n$  be the actual time difference between two executions of a software task (e.g., starting the ADC). The desired time difference is  $1/f_s$ . For a DAS, we define the time-jitter at sample  $n$  to be

$$\delta t_n = \Delta t_n - 1/f_s$$

Again, we must be able to place an upper bound,  $k$ , on the time-jitter.

$$-k \leq \delta t_n \leq +k \text{ for all } n$$

If the analog input,  $V(t)$ , is a slowly varying signal, then time jitter causes little error. In general, the magnitude of the voltage error,  $\Delta V$ , caused by time jitter,  $\delta t$ , is a function of the slew rate of the input.

$$\Delta V = \frac{dV}{dt} * \delta t \quad (24)$$

Consider a data acquisition system that samples a signal at  $f_s$  and performs a 60 Hz digital notch filter to remove noise. To analyze the effect of time jitter we will assume the actual signal is composed of a constant signal,  $V_s$ , and a 60 Hz noise signal,  $V_n$ .

$$V(t) = V_s + V_n \sin(2\pi 60t)$$

The signal-to-noise ratio is  $V_s/V_n$ . Assuming the voltage range is between  $-5$  to  $+5$  volts, the  $m$  bit A/D converter will create an infinite digital sample sequence,  $x(1) x(2), \dots, x(n), \dots$ . The approximate digital values are

$$x(n) = \frac{2^{m-1}}{5} V\left(\frac{n}{f_s}\right)$$

If the sampling rate,  $f_s$ , is  $120 \cdot k$  Hz where  $k$  is an integer constant greater than 1, a simple 60 Hz digital notch filter is

$$y(n) = \frac{x(n) + x(n-k)}{2}$$

First we will consider a DAS that samples the ADC and calculates the digital filter and displays the results. In a real-time operating system this process will request service  $f_s$  times a second at exact intervals. Unfortunately, time latency occurs between the process request and

the process service. What happens to the filter performance if this delay is *always* exactly “ $\mathbf{d}$ ?” Assume the delay,  $\mathbf{d}$ , is smaller than  $1/f_s$ , so that the sample is simply delayed and not skipped altogether. In this situation, we ask how much noise passes the filter?

$$\begin{aligned} y(n) &= \frac{x(n) + x(n-k)}{2} = \frac{\frac{2^{m-1}}{5} \cdot V(t_n + d) + \frac{2^{m-1}}{5} \cdot V\left(t_n - \frac{-k}{f_s} + d\right)}{2} \\ &= \frac{2^{m-1}}{10} \left( V_s + V_n \sin\left(2\pi 60 \cdot \left(\frac{n}{120 \cdot k} + d\right)\right) + V_s + V_n \sin\left(2\pi 60 \cdot \left(\frac{n-k}{120 \cdot k} + d\right)\right) \right) \\ &= \frac{2^{m-1}}{10} \left( 2V_s + V_n \sin\left(\frac{\pi n}{k} + 120\pi d\right) + V_n \sin\left(\frac{\pi(n-k)}{k} + 120\pi d\right) \right) \\ &= \frac{2^{m-1}}{5} V_s \quad \text{independent of } d \text{ and } V_n \end{aligned}$$

For the second consideration, we will look at what happens if *one* ADC sample is delayed but the others occur on time. For example, the request may have come at a time when the OS is performing a higher priority job, and the service is delayed. Let  $\mathbf{k} = 2$ ,  $f_s = 240$ , and  $t_n$  be exactly  $n/240$  for all  $n$  except at  $n = 50$  where  $t_{50} = 50/240 + d$ . Assume the delay,  $\mathbf{d}$ , is smaller than  $1/240$ , so that the sample is simply delayed and not skipped altogether. We can derive an expression for the error in  $y(50)$  as a function of the delay  $\mathbf{d}$ , the ADC precision  $\mathbf{m}$ , and the 60 Hz noise  $V_n$ .

$$\begin{aligned} y(n) &= \frac{x(50) + x(48)}{2} = \frac{\frac{2^{m-1}}{5} \cdot V(t_{50} + d) + \frac{2^{m-1}}{5} \cdot V\left(t_{48} - \frac{-1}{120}\right)}{2} \\ &= \frac{2^{m-1}}{10} \left( V_s + V_n \sin\left(\frac{2\pi 60 \cdot 50}{240} + 120\pi d\right) + V_s + V_n \sin\left(\frac{2\pi 60 \cdot 48}{240}\right) \right) \\ &= \frac{2^{m-1}}{10} (2V_s + V_n \sin(\pi + 120\pi d)) \\ &= \frac{2^{m-1}}{10} (2V_s - V_n \sin(120\pi d)) \\ \text{error (50)} &= -V_n \frac{2^{m-2}}{5} \sin(120\pi d) \approx -V_n \frac{2^{m-2}}{5} 120\pi d \quad \text{for } d \ll \frac{1}{60} \end{aligned}$$

We could have derived this same result using the general equation (in this case  $\mathbf{d} = \delta t$ ).

$$\Delta V = \frac{dV}{dt} * \delta t = \frac{2^{m-1}}{5} * \delta t * \frac{d}{dt} (V_s + V_n \sin(2\pi 60t))$$

The largest slope occurs when  $\sin()$  is zero; the magnitude of the error in  $x(50)$  is thus bounded by

$$\Delta V \leq \frac{2^{m-1}}{5} * \delta t * 120\pi V_n$$

Because the filter divides  $x(50)$  by 2, the error in  $y(50)$  is  $1/2$  of the preceding equation, yielding the same result.

**Checkpoint 15.8:** What is the largest source of time-jitter when sampling the ADC using a background interrupt service routine?

## 15.6 Discrete Fourier Transform

The *Discrete Fourier Transform* (DFT) converts data in the time domain to data in the frequency domain. We can use the DFT to measure SNR, to identify noise type, and to design FIR digital filters. In fact, the spectrum analyzer is simply a high-speed data-acquisition system followed by a DFT. The Fast Fourier Transform (FFT) is a technique to calculate the DFT with fewer additions and multiplications. There are four important parameters when employing the DFT. The first parameter is sampling rate,  $f_s$ . While the DFT deals only with samples and bins with no concept of volts, seconds, and Hz when applying it to real data, we assume the samples have units, are bound by physical limits, and are evenly spaced at time intervals  $T = 1/f_s$ . The second parameter is sequence length,  $N$ . The other two parameters are input resolution and range. In real systems, input data come from the ADC or input capture, and the output data go to the DAC or PWM. Therefore, the performance of the DFT will be affected by the range and resolution of the input. The input to the DFT will be  $N$  samples versus time, and the output will be  $N$  points in the frequency domain.

$$\begin{aligned} \text{Input: } \{a_n\} &= \{a_0, a_1, a_2, \dots, a_{N-1}\} \\ \text{Output: } \{A_k\} &= \{A_0, A_1, A_2, \dots, A_{N-1}\} \end{aligned}$$

The definition of the DFT is

$$A_k = \sum_{n=0}^{N-1} a_n W_N^{kn} \quad \text{where } W_N = e^{-j2\pi/N} \quad \text{and } k = 0, 1, 2, \dots, N-1 \quad (25)$$

The DFT output  $A_k$  at index  $k$  represents the amplitude and phase of the input at frequency  $k*f_s/N$  (in Hz). The DFT resolution in Hz/bin is the reciprocal of the total time spent gathering time samples. That is,  $1/(N*T)$ . The *Inverse Discrete Fourier Transform* (IDFT) converts data in the frequency domain to data in the time domain. The input to the IDFT will be  $N$  points in the frequency domain, and the output will be  $N$  samples in the time domain.

$$\begin{aligned} \text{Input: } \{A_k\} &= \{A_0, A_1, A_2, \dots, A_{N-1}\} \\ \text{Output: } \{a_n\} &= \{a_0, a_1, a_2, \dots, a_{N-1}\} \end{aligned}$$

The definition of the IDFT is

$$a_n = \frac{1}{N} \sum_{k=0}^{N-1} A_k W_N^{-kn} \quad \text{where } W_N = e^{-j2\pi/N} \quad \text{and } n = 0, 1, 2, \dots, N-1 \quad (26)$$

When presenting frequency data, we can use a log scale, making it easier to visualize frequency components with widely varying amplitudes. Because the system has physical limits, we use those limits to define full scale. Assume the audio system in Chapter 12 samples sound as a voltage  $2.5 \pm 2$  V. For this system, we would define full scale  $V_{FS}$  as 2 V. In particular, if  $V$  is a DFT output in volts, we can convert it to *dB full scale* using

$$dB_{FS} = 20 * \log_{10}(V/V_{FS})$$

## 15.7 FIR Filter Design

In this section, we will use the DFT as a general tool to design FIR filters. We begin by *choosing the sampling rate*, which must be larger than two times the largest signal frequency we wish to process. After we have chosen the sampling rate (e.g., 10 kHz), we will *choose a FIR filter length* (e.g.,  $N = 51$ ). The ratio  $f_s/N$  (e.g.,  $10 \text{ kHz}/51 = 196 \text{ Hz}$ ) will determine the frequency resolution of the FIR filter design. Next, we plot or print the desired gain/phase versus frequency response. The magnitude of  $H(k)$  is selected to implement the *desired gain versus frequency response*. That is,  $|H(k)|$  will be the filter gain at  $k*f_s/N$ . The angle of  $H(k)$  is selected to implement the *desired phase versus frequency*

response. That is,  $\text{angle}[\mathbf{H}(\mathbf{k})]$  will be the filter phase at  $\mathbf{k} * \mathbf{f}_s / \mathbf{N}$ . For frequencies above  $\frac{1}{2} f_s$ , we must make  $\mathbf{H}(\mathbf{k})$  be the complex conjugate of the  $\mathbf{N} - \mathbf{k}$  term. This will guarantee that the inverse DFT of  $\mathbf{H}(\mathbf{k})$  will yield real results. Let  $x(n)$  be the input (read from the ADC) and  $X(k)$  be the input in the frequency domain. Let  $y(n)$  be the FIR filter output, and let  $Y(k)$  be the FIR filter output in the frequency domain.

$$\begin{aligned} Y(k) &= H(k) X(k) \\ y(n) &= \text{IDFT} \{ H(z) \text{DFT}\{x(t)\} \} \end{aligned}$$

We take IDFT of the  $H(k)$  to get  $N$  FIR filter coefficients. Multiplication in the frequency domain is equivalent to *convolution* in the time domain. The FIR filter is the convolution of the data with the inverse transform of the desired filter.

$$\begin{aligned} y(n) &= h(n) * x(n) = x(n) * h(n) && (* \text{ means convolution here}) \\ y(n) &= \sum [h(i) \cdot x(n-i)] \text{ as } i \text{ goes from } -\infty \text{ to } +\infty && (\cdot \text{ means multiplication here}) \end{aligned}$$

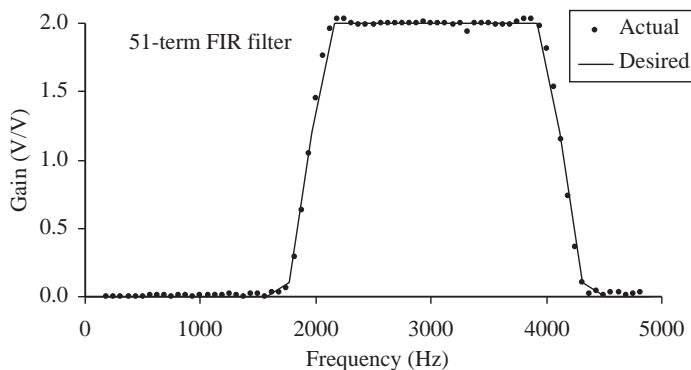
Because there are a finite number of  $h(n)$  terms, the convolution is a finite sum

$$y(n) = \sum [h(i) \cdot x(n-i)] \text{ as } i \text{ goes from } 0 \text{ to } N-1 \quad (\cdot \text{ means multiplication here})$$

**Example 15.1** Design a digital filter that passes frequencies from 2 to 4 kHz with a gain of 2. The input is audio with frequency components from 100 Hz to 5 kHz.

**Solution** We choose the sampling rate at twice the maximum frequency of the input or  $f_s = 10$  kHz. Next we choose a filter size. The larger  $N$ , the better the actual filter will match our desired response, but the slower it will execute. For this solution, we could have chosen any size from 32 to 64 and obtained similar results. In order to preserve the shape of the audio signals, we will implement linear phase. The desired filter gain is shown as Figure 15.12 and Table 15.3. The lines in the figure are the desired filter gain, and the dots will be the actual gain as implemented by the fixed-point math in Program 15.8.

**Figure 15.12**  
Desired and actual filter responses. This is  $H$ .



The  $H(N-k)$  values must be the complex conjugates of  $H(k)$ . Because the negative frequencies in Table 15.3 are complex conjugates of the positive frequencies,  $h(n)$  will be real. Next, we scale the  $h(n)$  values to make 51 fixed-point coefficients  $h[51]$ . For example, the first term  $h(0)$  is 0.003620115, which will be approximated in fixed-point as 4/1024. In summary, the  $h[51]$  coefficients are the IDFT of the values in Table 15.3 multiplied by 1024 and rounded to an integer.

```
const short h[51]={ 4, 3, -7, -2, -13, 22, 3, 2, -14, -31, 43, 1, 26,
-44, -51, 65, 0, 86, -117, -78, 82, 24, 316, -496, -277, 908, -277,
-496, 316, 24, 82, -78, -117, 86, 0, 65, -51, -44, 26, 1, 43, -31,
-14, 2, 3, 22, -13, -2, -7, 3, 4};
```

**Table 15.3**

Desired filter response.  
This is H.

<b>k</b>	<b>f (Hz)</b>	<b>Mag(H(k))</b>	<b>Angle(H(k))</b>
0	0.00	0.00	0.00
1	196.08	0.00	-3.08
2	392.16	0.00	-6.16
3	588.24	0.00	-9.24
4	784.31	0.00	-12.32
5	980.39	0.00	-15.40
6	1176.47	0.00	-18.48
7	1372.55	0.00	-21.56
8	1568.63	0.00	-24.64
9	1764.71	0.10	-27.72
10	1960.78	1.20	-30.80
11	2156.86	2.00	-33.88
12	2352.94	2.00	-36.96
13	2549.02	2.00	-40.04
14	2745.10	2.00	-43.12
15	2941.18	2.00	-46.20
16	3137.25	2.00	-49.28
17	3333.33	2.00	-52.36
18	3529.41	2.00	-55.44
19	3725.49	2.00	-58.52
20	3921.57	2.00	-61.60
21	4117.65	1.20	-64.68
22	4313.73	0.10	-67.76
23	4509.80	0.00	-70.84
24	4705.88	0.00	-73.92
25	4901.96	0.00	-77.00

Program 15.8 shows two implementations of this FIR filter. There are 100  $\mu$ s for each sample (ADC, filter, and DAC). We will implement the MACQ using two copies of the data—similar to Program 15.2. We could add this filter to the audio system developed in Example 12.4. In order to allow this FIR run in real time, we must use the assembly language version. The C version is included simply to explain how it works. In particular, the **emac** instruction provides an efficient means to execute digital filters. The assembly FIR filter with 51 terms has an execution time of  $77 + 50 \times 20 = 1070$  cycles, which is 45  $\mu$ s assuming a 24-MHz E clock. With a sampling rate of 10 kHz, this represents about 50% of the available CPU time. The C version of this FIR filter requires over 3000 cycles to execute.

<pre>         org \$0800 ; Assembly Pt    rmb 2 sum   rmb 4      ;32-bit intermediate Data  rmb 204   ;51 entries, 2 copies         org \$4000 ; input in RegD, output in RegD Filter_Calc         ldx Pt      ; if(Pt == &amp;Data[0]){         cpx #Data         bne nowrap wrap   ldx #Data+102 ; Pt = &amp;Data[50];         bra setPt   ; } else{ nowrap dex          ; 2 bytes each         dex          ; Pt--; setPt  stx Pt      ;RegX is Pt         std 0,x </pre>	<pre> short Data[102]; // two copies short *Pt; // pointer to current void Filter_Init(void){ short *pt;     for(pt=Data; pt&lt;&amp;Data[102]; pt++){         *pt = 0; // fill buffer with zeros     }     Pt = &amp;Data[0]; } // calculate one filter output // called at sampling rate // Input: new ADC data // Output: filter output, DAC data short Filter_Calc(short newdata){     int i; long sum; short *pt,*apt;     if(Pt == &amp;Data[0]){         Pt = &amp;Data[50]; // wrap     }     for(i=0; i&lt;51; i++){         sum += newdata * Data[i];     }     *Pt = sum; } </pre>
--	---

*continued on p. 739*

continued from p. 738

```

std    104,x    ;*Pt=*(Pt+51)=newdata;
ldy    #h          ; apt = h;
movw  #0,sum      ; clear temporary
movw  #0,sum+2
loop   emacs sum  ; sum += (*pt)*(*apt);
leax   2,x        ; pt++;
leay   2,y        ; apt++;
dbne   A,loop
ldy    sum        ; most sign 16 bits
ldd    sum+2      ; least sign 16 bits
ldx    #1024      ; fixed point
edivs
tfr    y,d
rts
} else{
    Pt--;           // make room for data
}
*Pt = *(Pt+51) = newdata; // two copies
pt = Pt; // copy of data pointer
apt = h; // pointer to coefficients
sum = 0;
for(i=51; i; i--){
    sum += (*pt)*(*apt);
    apt++;
    pt++;
}
return sum/1024;
}

```

### Program 15.8

51-term FIR filter.

**Checkpoint 15.9:** How can we prove the software in Program 15.8 cannot overflow?

**Checkpoint 15.10:** Can you think of a way to reduce the number of multiplies in Program 15.8 while still performing the exact same filter?

## 15.8 Direct-Form Implementations

The general form for the transfer function for an IIR filter is

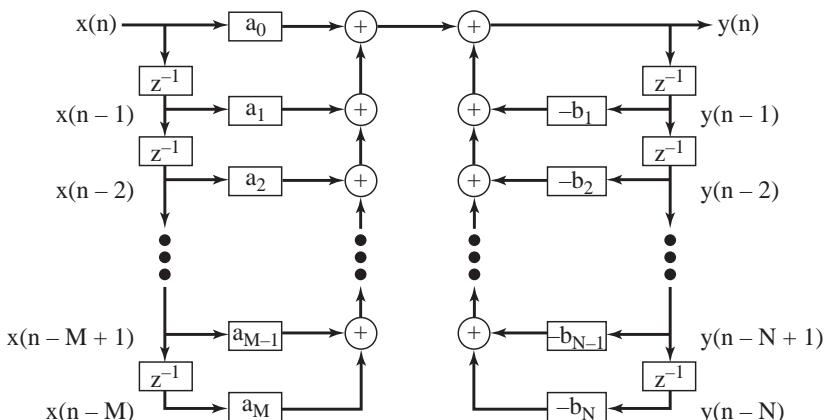
$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 + a_1z^{-1} + a_2z^{-2} + \dots + a_Mz^{-M}}{1 + b_1z^{-1} + b_2z^{-2} + \dots + b_Nz^{-N}}$$

This converts to the standard difference equation

$$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + \dots + a_Mx(n-M) - b_1y(n-1) - b_2y(n-2) - \dots - b_Ny(n-N)$$

The direct-form calculation of this filter requires two MACQs with lengths M and N. There are  $(M + N - 1)$  multiplies and  $(M + N - 2)$  additions. The data flow picture in Figure 15.13 illustrates the standard implementation. The  $z^{-1}$  boxes are time delays (MACQ). The other boxes are multiplications. The + circles are additions.

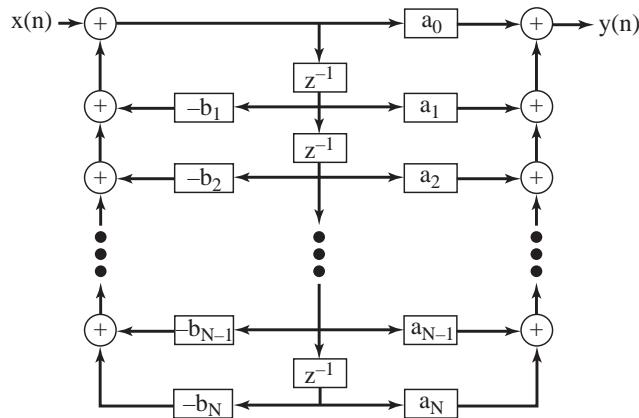
**Figure 15.13**  
General filter design  
using a direct-form  
calculation.



For the next implementation we specify the filter with  $N = M$ . We can do this without loss of generality by letting some of the coefficients be zero. An alternative implementation, called the *direct-form II realization*, requires only one MACQ of length  $N$ . There are still  $(2N - 1)$  multiplies and  $(2N - 2)$  additions. The data flow picture of Figure 15.14 illustrates the implementation.

**Figure 15.14**

General filter design using a direct-form II calculation.



## 15.9 Exercises

- 15.1** For each term, give a definition in 32 words or less.

- |  |                             |
|--|-----------------------------|
| <b>a)</b> Aliasing                             | <b>d)</b> Complex conjugate |
| <b>b)</b> Filter Q                             | <b>e)</b> MACQ              |
| <b>c)</b> Impulse response of a digital filter | <b>f)</b> Overflow          |

- 15.2** For each term, give the equation definition.

- |                       |  |
|-----------------------|--|
| <b>a)</b> Z-transform | <b>d)</b> Relationship between time jitter and voltage error |
| <b>b)</b> DFT         | <b>e)</b> Filter gain given input frequency f                |
| <b>c)</b> IDFT        | <b>f)</b> Convolution between x and h                        |

- 15.3** Consider the use of the Z transform in the design and analysis of digital filters.

- a)** State the definition of the Z transform.
- b)** Why can't we use the Z transform on a median filter?
- c)** Use the Z transform to determine the DC gain and phase of the following digital filter:

$$y(n) = x(n) - x(n - 2) + y(n - 1)$$

- 15.4** List the four parameters we need to decide when implementing a DFT.

- 15.5** For each pair of terms, compare and contrast in 32 words or less.

- |  |  |
|--|--|
| <b>a)</b> Causal versus noncausal filter | <b>d)</b> Laplace transform versus the Z-transform |
| <b>b)</b> Linear versus nonlinear filter | <b>e)</b> A pole versus a zero                     |
| <b>c)</b> FIR versus IIR filter          | <b>f)</b> A complex versus an imaginary number     |

- 15.6** 256 data points are sampled at 10 Hz with a 10-bit ADC. The ADC range is 0 to 3.3 V. A DFT is performed on the data. What is the frequency resolution? What range of frequencies is represented in the DFT output?

- 15.7** For each situation, specify whether you expect the gain at frequency f to increase, decrease, or not change much at all.

- a)** A zero is moved closer to frequency f on the z-plane.
- b)** A pole is moved closer to frequency f on the z-plane.
- c)** A zero already near frequency f on the z-plane is replaced with a double zero.
- d)** A pole already near frequency f on the z-plane is replaced with a double pole.
- e)** A pole currently near frequency f on the z-plane is moved to the origin.
- f)** A pole currently near frequency f on the z-plane is outside the unit circle.

**15.8** For each filter, specify whether it is linear or nonlinear. If it is linear, specify whether it is FIR or IIR.

- a)  $y(n) = x(n)^2 + 2x(n) + 1$
- b)  $y(n) = x(n)/4 + y(n-1) - x(n-4)/4$
- c)  $y(n) = \min\{x(n), x(n-1)\}$
- d)  $y(n) = (x(n+1) + x(n-1))/2$

**D15.9** Let the input be the sum of two sine waves:  $x(t) = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t)$ . Assume the digital filter will pass both these frequencies with a gain of 1. This filter implements a linear phase response. What can you say about the output of the filter? That is, derive an equation describing the output as a function of time.

**D15.10** Consider the following digital filter:  $y(n) = (x(n) - x(n-2))/2$ .

- a) Using the Z transform, derive general expressions for the gain and phase of the filter.
- b) Using the general expressions from part a, calculate the gain and phase of the filter at DC and 60 Hz if the sampling rate is 240 Hz.

**D15.11** Design a 10-Hz digital low-pass filter with a sampling rate of 1000 Hz. Make the gain at DC equal to 1, and the gain at 10 Hz equal to 0.707.

- a) Show the pole-zero plot of your filter.
- b) Show the  $H(z)$  transform.
- c) Show the floating-point version of the digital filter.
- d) Show the fixed-point version of the digital filter.

**D15.12** Design a digital filter that rejects both 60 Hz and 120Hz, assuming the sampling rate is 480 Hz. Apply gain scaling so the DC gain is 1. Give the filter in a form that can be implemented with fixed-point math.

**D15.13** Consider the simple sliding average filter for a general sampling rate of 1000 Hz. This filter is a low-pass filter, as shown in Figure 15.5:

$$y(n) = \frac{1}{k} \sum_{i=0}^{i=k-1} X(n-i)$$

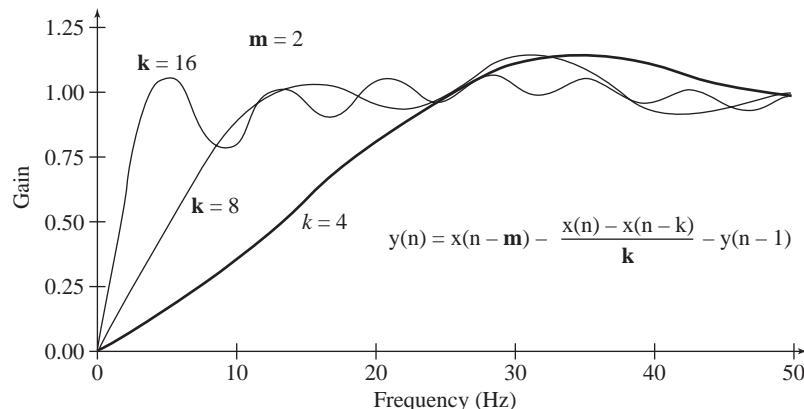
What value of  $k$  should we use to make a gain of about 0.7 at 10 Hz?

**D15.14** Consider this digital HPF, where  $m$  and  $k$  are constants. Figure 15.15 shows the gain versus frequency of this filter for  $f_s = 100$  Hz,  $m = 2$ , and  $k = 4, 8$ , and  $16$ .

$$y(n) = x(n-m) - \frac{x(n) - x(n-k)}{k} - y(n-1)$$

- a) Is this a FIR or IIR filter? Explain.
- b) Derive the  $H(z)$  transform of this filter.
- c) Let  $f_s = 100$  Hz,  $m = 8$ , and  $k = 16$ . Use  $H(z)$  to plot gain versus frequency from 0 to 50 Hz.

**Figure 15.15**  
Gain versus frequency of  
a high-pass filter.



**D15.15** We defined time-jitter,  $\delta t$ , as the difference between when a periodic task is supposed to be run, and when it is actually run. The goal of a real-time DAS is to start the ADC at a periodic rate,  $\Delta t$ . Let  $t_n$  be the nth time the ADC is started. In particular, the goal to make  $t_n - t_{n-1} = \Delta t$ . The jitter is defined as the constant,  $\delta t$ , such that

$$\Delta t - \delta t < t_i - t_{i-1} < \Delta t + \delta t \quad \text{for all } i$$

Assume the input to the ADC can be described as  $V(t) = A + B\sin(2\pi ft)$ , where  $A$ ,  $B$ ,  $f$  are constants.

a) Derive an estimate of the maximum voltage error,  $\delta V$ , caused by time-jitter. Basically, solve for the largest possible value of  $\delta V$  as a function of  $\delta t$ ,  $A$ ,  $B$ , and  $f$ .

b) Consider the situation where this time-jitter is unacceptably large. Which modification to the system will reduce the error the most? Justify your selection.

- A) Run the ADC in continuous mode
- B) Convert from spinlock semaphores to blocking semaphores
- C) Change from round robin to priority thread scheduling
- D) Reduce the amount of time the system runs with interrupts disabled.
- E) Increase the size of the DataFifo

## 15.10 Lab Assignments

**Lab 15.1** You will design, implement, and test a real-time data acquisition system with a sampling rate of 1000 Hz. This system will include a sliding average filter (data  $x(n)$  is sampled at 1000 Hz, and the equation  $y(n)$  is also calculated 1000 times/sec).

$$y(n) = (x(n) + x(n - 999))/2$$

The trick to this problem is to implement an MACQ that does not require shifting the data on each sample. You should develop LCD display device driver functions, `LCD_Init` and `LCD_Display`, using them to display your results. Using the filtered data, the main program will calculate the RMS, minimum and maximum values over a 1 sec interval. Once a second display these results. Write debugging code to measure the worst-case time-jitter in the sampling process. Use an external sinusoidal source to experimentally measure the gain-versus-frequency response of the digital filter.

**Lab 15.2** You will design, implement, and test a real-time data acquisition system with a sampling rate of 1000 Hz. This system will include a sliding average filter (data  $x(n)$  is sampled at 1000 Hz, and the equation  $y(n)$  is also calculated 1000 times/sec).

$$y(n) = \frac{1}{1000} \sum_{i=0}^{i=999} x(x - i)$$

The trick to this problem is to implement an MACQ that does not require shifting the data on each sample and to implement the sum without having to perform 1000 additions for each calculation. You should develop LCD display device driver functions, `LCD_Init` and `LCD_Display`, using them to display your results. Using the filtered data, the main program will calculate the RMS, minimum and maximum values over a 1 sec interval. Once a second display these results. Write debugging code to measure the worst-case time-jitter in the sampling process. Use an external sinusoidal source to experimentally measure the gain-versus-frequency response of the digital filter.

**Lab 15.3** The objective in this problem is to design a modular and flexible software implementation of a high-Q digital notch IIR filter (Program 15.9). The sampling frequency and notch frequency will be set dynamically (at run time). These parameters will be established when the user calls your `Filter_Init()` function. You may implement any Q you wish, but it should be similar to the high-Q 60 Hz notch IIR filter developed in Section 15.4. Adjust the filter gain so that the gain at DC is 1.0. The data will be passed through the notch filter by calls to your `Filter()` function. You may assume without checking that the cutoff frequency,  $f_c$ , is strictly greater than 0 and strictly less than one-half the sampling rate,  $0 < f_c < 1/2 f_s$ . Your software handles the establishment of the actual sampling rate. You should use only fixed-point integer calculations. For an example software system that allows

**Program 15.9**

Example application of the real-time DAS and filter.

```
#include "FILTER.H"      // prototype file for your filter
void Collect(void){ short data;
    data = ADC_In(0);    // sample
    Fifo_Put(data);
}
void main(void){ short x,y;      // filter input/output
    LCD_Init();
    Filter_Init(100,1000,&Collect); // notch 100 Hz, sample at 1KHz
    for(;;){
        while(Fifo_Get(&x)==0){} // wait for input
        y = Filter(x);          // execute filter
        LCD_Display(y);         // show results
    }
}
```

for runtime binding of functions to execute, see the OC example on the CD. Use an external sinusoidal source to experimentally measure the gain-versus-frequency response of the digital filter. The following main program illustrates how a user might apply your filter software.

**Lab 15.4** Design, implement, and test a digital hearing aid. Perform a web search to learn about hearing loss and hearing tests. In particular, you will solve the classic problem of reduced sensitivity that many elderly people suffer. The input sound is sampled using a microphone like Example 12.4. A FIR filter is implemented like Example 15.1. The output sound is created either with a DAC like Example 12.4 or a PWM like Example 11.3. Your system will drive headphones, so a regular op amp can be used in place of a high-current speaker amplifier. In general, the patient would first have a hearing test, and then you would use the results to generate the desired gain versus frequency for the hearing aid. However in this lab, pass frequencies below 1 kHz with gain 0.25 and pass frequencies from 1 to 5 kHz with a gain of 5. This 20 to 1 ratio will accentuate high frequencies. Implement linear phase. Test the gain versus frequency response of the input amp, the digital filter, and the output amp.

# Appendix 1 Glossary

**1/f noise** A fundamental noise in resistive devices arising from fluctuating conductivity. Same as pink noise.

**2's complement** (*see* two's complement).

**60 Hz noise** An added noise from electromagnetic fields caused by either magnetic field induction or capacitive coupling.

**accumulator** High-speed memory located in the processor, used to perform arithmetic or logical functions. The accumulators on the 9S12 are A and B.

**accuracy** A measure of how close our instrument measures the desired parameter referred to the NIST.

**acknowledge** Clearing the interrupt flag bit that requested the interrupt.

**active thread** A thread that is in the ready-to-run circular linked list. It is either running or ready to run.

**actuator** Electro-mechanical or electro-chemical device that allows computer commands to affect the external world.

**ADC** Analog-to-digital converter, an electronic device that converts analog signals (e.g., voltage) into digital form (i.e., integers).

**address bus** A set of digital signals that connect the CPU, memory, and I/O devices, specifying the location to read or write for each bus cycle. *See also* control bus and data bus.

**address decoder** A digital circuit having the address lines as input and a select line as output (*see* select signal).

**aging** A technique used in priority schedulers that temporarily increases the priority of low priority threads so they are run occasionally (*see* starvation).

**aliasing** When digital values sampled at  $f_s$  contain frequency components above  $\frac{1}{2} f_s$ , then the apparent frequency of the data is shifted into the 0 to  $\frac{1}{2} f_s$  range. *See* Nyquist theory.

**alternatives** The total number of possibilities. For example, an 8-bit number scheme can represent 256 different numbers. An 8-bit digital-to-analog converter (DAC) can generate 256 different analog outputs.

**anode** The positive side of a diode. Current enters the anode side of a diode. Contrast to cathode.

**answer modem** The device that receives the telephone call.

**anti-reset-windup** Establishing an upper bound on the magnitude of the integral term in a PID controller, so this term will not dominate, when the errors are large.

**arithmetic logic unit (ALU)** Component of the processor that performs arithmetic and logic operations.

**arm** Activate so that interrupts are requested. On the 9S12, most flags that can request interrupts will have a corresponding arm bit to allow or disallow that flag to request interrupts. Contrast to enable.

**armature** The moving structure in a relay; the part that moves when the relay is activated. Contrast to frame.

**ASCII** American Standard Code for Information Interchange, a code for representing characters, symbols, and synchronization messages as 7-bit, 8-bit or 16-bit binary values.

**assembler** System software that converts an assembly language program (human-readable format) into object code (machine-readable format).

**assembly directive** Operations included in the program that are not executed by the computer at run time, but rather are interpreted by the assembler during the assembly process. Same as pseudo-op.

**assembly listing** Information generated by the assembler in human-readable format, typically showing the object code, the original source code, assembly errors, and the symbol table.

**asynchronous bus** A communication protocol without a central clock whereby the data is transferred using two or three control lines implementing a handshake interaction between the memory and the computer.

**asynchronous communications interface adapter (ACIA)** Device to transmit data with asynchronous serial communication protocol. Same as UART and SCI.

**asynchronous protocol** A protocol whereby the two devices have separate and distinct clocks.

**atomic** Software execution that can not be divided or interrupted. Once started, an atomic operation will run to its completion without interruption. On most computers the assembly language instructions are atomic.

**autoinitialization** The process of automatically reloading the address registers and block size counters at the end of a previous block transfer, so that DMA transfer can occur indefinitely without software interaction.

**availability** The proportion of the total time that the system is working. MTBF is the mean time between failures, MTTR is the mean time to repair, and availability is  $\text{MTBF}/(\text{MTBF} + \text{MTTR})$ .

**background mode** A 9S12 mode with the background debug module (BDM) active.

**bandwidth** In communication systems, the information transfer rate, the amount of data transferred per second. Same as throughput. In analog circuits, the frequency at which the gain drops to 0.707 of the normal value. For a low-pass system, the frequency response ranges from 0 to a maximum value. For a high-pass system, the frequency response ranges from a minimum value to infinity. For a bandpass system, the frequency response ranges from a minimum to a maximum value. Compare to frequency response.

**bandwidth coupling** Module A is connected to Module B, because data flows from A to B.

**bang-bang** A control system where the actuator has only two states, and the system “bangs” all the way in one direction or “bangs” all the way in the other; same as binary controller.

**bank-switched memory** A memory module with two banks that interfaces to two separate address/data buses. At any given time one memory bank is attached to one address/data bus and the other bank is attached to the other bus. However, this attachment can be switched.

**basis** Subset from which linear combinations can be used to reconstruct the entire set.

**baud rate** In general, the baud rate is the total number of bits (information, overhead, and idle) per time that are transmitted. In a modem application it is the total number of sounds per time that are transmitted.

**bi-directional** Digital signals that can be either input or output.

**biendian** The ability to process numbers in both big and little endian formats.

**big endian** Mechanism for storing multiple byte numbers such that the most significant byte exists first (in the smallest memory address). *See also* little endian.

**binary** A system that has two states, on and off.

**binary controller** Same as bang-bang.

**binary semaphore** A semaphore that can have two values. The value = 1 means OK, and the value = 0 means busy. Compare to counting semaphore.

**bipolar transistor** Either an NPN or a PNP transistor.

**bipolar stepper motor** A stepper motor in which the current flows in both directions (in/out) along the interface wires; a stepper with four interface wires. Contrast to unipolar stepper motor.

**bit** Basic unit of digital information taking on the value of either 0 or 1.

**bit rate** The information transfer rate, given in bits per second. Same as bandwidth and throughput.

**bit time** The basic unit of time used in serial communication. With serial channel, bit time is 1/baud rate.

**blind cycle** A software/hardware synchronization method in which the software waits a specified amount of time for the hardware operation to complete. The software has no direct information (blind) about the status of the hardware.

**block correction code (BCC)** A code (e.g., horizontal parity) attached to the end of a message used to detect and correct transmission errors.

**blocked thread** A thread that is not scheduled for running because it is waiting on an external event.

**blocking semaphore** A semaphore whereby the threads will block (so other threads can perform useful functions) when they execute wait on a busy semaphore. Contrast to spinlock semaphore.

**Board Support Package (BSP)** A set of software routines that abstract the I/O hardware such that the same high level code can run on multiple computers.

**borrow** During subtraction, if the difference is too small, then we use a borrow to pass the excess information into the next higher place. For example, in decimal subtraction 36-27 requires a borrow from the ones to tens place, because 6-7 is too small to fit into the 0 to 9 range of decimal numbers.

**bounded waiting** The condition where once a thread begins to wait on a resource, there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed.

**break-before-make** In a double-throw relay or double-throw switch, there is one common contact and two separate contacts. Break-before-make means as the common contact moves from one of separate contacts to another, it will break off (finish bouncing and no longer touch) the first contact before it makes (begins to bounce and starts to touch) the other contact. A *form C* relay has a *break-before-make* operation.

**break or trap** A break or a trap is a debugging instrument that halts the processor. The **TEXAS** application will halt both software and hardware simulation when a specific address is encountered. With a resident debugger, the break is created by replacing specific opcode with a software interrupt instruction. When encountered it will stop your program and jump into the debugger. Therefore, a break halts the software. The condition of being in this state is also referred to as a break.

**breakdown** A transducer that stops functioning when its input goes above a maximum value or below a minimum value. Contrast to dead zone.

**breakpoint** The place where a break is inserted, the time when a break is encountered, or the time period when a break is active.

**brushed DC motor** A motor where the current reversals are produced with brushes between the stator and rotor. They are less expensive than brushless DC motors.

**brushless DC motor (BLDC)** A motor where the current reversals are produced with shaft sensors and an electronic controller. They are faster and more reliable than brushed DC motors.

- buffered I/O** A FIFO queue is placed in between the hardware and software in an attempt to increase bandwidth by allowing both hardware and software to run in parallel.
- burn** The process of programming a ROM, PROM, or EEPROM.
- burst DMA** An I/O synchronization scheme that transfers an entire block of data all at once directly from an input device into memory, or directly from memory to an output device.
- bus** A set of digital signals that connect the CPU, memory, and I/O devices, consisting of address signals, data signals, and control signals. *See also* address bus, control bus and data bus.
- bus bandwidth** The number of bytes transferred per second between the processor and memory.
- bus interface unit (BIU)** Component of the processor that reads and writes data from the bus. The BIU drives the address and control buses.
- busy-waiting** A software/hardware synchronization method whereby the software continuously reads the hardware status waiting for the hardware operation to complete. The software usually performs no work while waiting for the hardware. Same as gadfly.
- byte** Digital information containing eight bits.
- carrier frequency** The average or midvalue sound frequency in the modem.
- carry** During addition, if the sum is too large, then we use a carry to pass the excess information into the next higher place. For example, in decimal addition  $36 + 27$  requires a carry from the ones to tens place because  $6 + 7$  is too big to fit into the 0 to 9 range of decimal numbers.
- cathode** The negative side of a diode. Current exits the cathode side of a diode. Contrast to anode.
- cathode ray tube (CRT) terminal** An I/O device used to input data from a keyboard and output character data to a screen. The electrical interface is usually asynchronous serial.
- causal** The property whereby the output depends on the present and past inputs, but not on any future inputs.
- ceiling** Establishing an upper bound on the result of an operation. *See also* floor.
- certification** A process where a governing body (e.g., FDA, NASA, FCC, DOD etc.) gives approval for the use of the device. It usually involves demonstrating the device meets or exceeds safety and performance criteria.
- channel** The hardware that allows communication to occur.
- checksum** The simple sum of the data, usually in finite precision (e.g., 8, 16, 24 bits).
- closed loop control system** A control system that includes sensors to measure the current state variables. These inputs are used to drive the system to the desired state.
- CMOS** A digital logic system called complementary metal oxide semiconductor. It has properties of low power and small size. Its power is a function of the number of transitions per second. Its speed is often limited by capacitive loading.
- cohesion** A cohesive module is one such that all parts of the module are related to each other to satisfy a common objective.
- command signals** The lines that specify general information about the current cycle; signals that specify whether or not to activate during this cycle; the specific times for the rise and fall edges are uncertain. Contrast to timing signals.
- common anode LED display** A display with multiple LEDs, configured with all of the LED anodes connected together; there are separate connections to the cathodes (current flows in the common anode and out the individual cathodes).

- common cathode LED display** A display with multiple LEDs, configured with all of the LED cathodes connected together; there are separate connections to the anodes (current flows in the individual anodes and out the common cathode).
- common mode** For a system with differential inputs, the common mode properties are defined as signals applied to both inputs simultaneously. Contrast to differential mode.
- common mode input impedance** Common mode input voltage divided by common mode input current.
- common mode rejection ratio** For a differential amplifier, CMRR is the ratio of the common mode gain divided by the differential mode gain. A perfect CMRR would be zero.
- compiler** System software that converts a high-level language program (human-readable format) into object code (machine-readable format).
- compression ratio** The ratio of the number of original bytes to the number of compressed bytes.
- concurrent programming** A software system that supports two tasks to be active at the same time. A computer with interrupts implements concurrent programming.
- condition code register (CCR)** Register in the processor that contains the status of the previous ALU operation, as well as some operating mode flags such as the interrupt enable bit.
- constraint** A condition defining how the system will be developed, generally restricting the range of solutions from which the system will be built.
- control bus** A set of digital signals that connect the CPU, memory, and I/O devices, specifying when to read or write for each bus cycle. *See also* address bus and data bus.
- control coupling** Module A is connected to Module B, because actions in A affect the control path in B.
- control unit (CU)** Component of the processor that determines the sequence of operations.
- cooperative multi-tasking** A scheduler that can not suspend execution of a thread without the thread's permission. The thread must cooperate and suspend itself. Same as nonpreemptive scheduler.
- counting semaphore** A semaphore that can have any signed integer value. The value  $> 0$  means OK, and the value  $\leq 0$  means busy. Compare to binary semaphore.
- CPU bound** A situation where the input or output device is faster than the software. In other words it takes less time for the I/O device to process data than for the software to process data. Contrast to I/O bound.
- CPU cycle** A memory bus cycle where the address and R/W are controlled by the processor. On microcontrollers without DMA, all cycles are CPU cycles. Contrast to DMA cycle.
- crisp input** An input parameter to the fuzzy logic system, usually with units such as cm, cm/sec, °C, and the like.
- crisp output** An output parameter from the fuzzy logic system, usually with units such as dynes, watts, and the like.
- critical section** Locations within a software module at which, if an interrupt were to occur at one of these locations, then an error could occur (e.g., data lost, corrupted data, program crash). Same as vulnerable window.
- cross-assembler** An assembler that runs on one computer but creates object code for a different computer.
- cross-compiler** A compiler that runs on one computer but creates object code for a different computer.
- cycle steal DMA** An I/O synchronization scheme that transfers data one byte at a time directly from an input device into memory, or directly from memory to an output device.

**cycle stretch** The action whereby some memory cycles are longer, allowing time for communication with slower memories; sometimes the memory itself requests the additional time, and sometimes the computer has a preprogrammed cycle stretch for certain memory addresses.

**DAC** Digital-to-analog converter, an electronic device that converts digital signals (i.e., integers) to analog form (e.g., voltage).

**data acquisition system** A system that collects information, same as instrument.

**data bus** A set of digital signals that connect the CPU, memory, and I/O devices, specifying the value that is being read or written for each bus cycle. *See also* address bus and control bus.

**data communication equipment (DCE)** A modem or printer connected a serial communication network.

**data terminal equipment (DTE)** A computer or a terminal connected to a serial communication network.

**dead zone** A condition of a transducer when a large change in the input causes little or no change in the output. Contrast to breakdown.

**deadlock** A scenario that occurs when two or more threads are all blocked, each waiting for the other with no hope of recovery.

**defuzzification** Conversion from the fuzzy logic output variables to the crisp outputs.

**desk checking** or **dry run** We perform a desk check (or dry run) by determining in advance, either by analytical algorithm or explicit calculations, the expected outputs of strategic intermediate stages and final results for a set of typical inputs. We then run our program to compare the actual outputs with this template of expected results.

**device driver** A collection of software routines that perform I/O functions.

**differential mode** For a system with differential inputs, the differential mode properties are defined as signals applied as a difference between the two inputs. Contrast to common mode.

**differential mode input impedance** Differential mode input voltage divided by differential mode input current.

**digital signal processing** Processing of data with digital hardware or software after the signal has been sampled by the ADC (e.g., filters, detection and compression/decompression).

**direct** An addressing mode in which the data or address value for the instruction is located in memory at address \$0000 to \$00FF.

**direct memory access (DMA)** The ability to transfer data between two modules on the bus; this transfer is usually initiated by the hardware (device needs service), and the software configures the communication, but the data is transferred without explicit software action for each individual piece of data.

**direction register** Specifies whether a bidirectional I/O pin is an input or an output. We set a direction register bit to 0 (or 1) to specify the corresponding I/O pin to be input (or output).

**disarm** Deactivate so that interrupts are not requested, performed by clearing the arm bit.

**Discrete Fourier Transform (DFT)** A technique to convert data in the time domain to data in the frequency domain. N data points are sampled at  $f_s$ . The resulting frequency resolution is  $f_s/N$ .

**DMA** Direct Memory Access is a software/hardware synchronization method whereby the hardware itself causes a data transfer between the I/O device and memory at the appropriate time when data needs to be transferred. The software usually can perform other work while waiting for the hardware. No software action is required for each individual byte.

**DMA cycle** A memory bus cycle in which the address and R/W are controlled by the DMA controller. Contrast to CPU cycle.

**double byte** Two bytes containing 16 bits. Same as word.

**double-pole relay** Two separate and complete relays, which are activated together. Contrast to single pole.

**double-throw relay** A relay with three contact connections, one common and two throws. The common will be connected to exactly one of the two throws (*see* single throw).

**double-throw switch** A switch with three contact connections. The center contact will be connected exactly to one of the other two contacts. Contrast with single-throw.

**download** The process of transferring object code from the host (e.g., the PC) to the target microcomputer (e.g., the 9S12).

**dropout** An error that occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. For example,  $I = 100*(N/51)$  can only result in the values 0, 100, or 200, whereas  $I = (100*N)/51$  properly calculates the desired result.

**dual address DMA** DMA that requires two bus cycles to transfer data from an input device into memory, or from memory to an output device.

**dual-port memory** A memory module that interfaces to two separate address/data buses, and allows both systems read/write access the data.

**duty cycle** For a periodic digital wave, the percentage of time the signal is high. When an LED display is scanned, it is the percentage of time each LED is active. A motor interfaced using pulse width modulation allows the computer to control delivered power by adjusting the duty cycle.

**dynamic allocation** Data structures such as the TCB that are created at runtime by calling `malloc()` and exist until the software releases the memory block back to the heap by calling `free()`. Contrast to static allocation.

**dynamic RAM** Volatile read/write storage built from a capacitor and a single transistor having a low cost, but requiring refresh. Contrast with static RAM.

**EEPROM** Electrically erasable programmable read only memory that is nonvolatile and easy to reprogram. Typically, EEPROM can be erased and reprogrammed 10,000 times.

**effective address register (EAR)** A register that contains the address for the current memory cycle.

**embedded computer system** A system that performs a specific dedicated operation where the computer is hidden or embedded inside the machine.

**emulator** An in-circuit emulator is an expensive debugging hardware tool that mimics the processor pin outs. To debug with an emulator, you would remove the processor chip and attach the emulator cable into the processor socket. The emulator would sense the processor input signals and recreate the processor outputs signals on the socket as if an actual processor chip were actually there, running at full speed. Inside the emulator you have internal read/write access to the registers and processor state. Most emulators allow you to visualize and record strategic information in real time without halting the program execution. You can also remove ROM chips and insert the connector of a ROM-emulator. This type of emulator is less expensive, and it allows you to debug ROM-based software systems.

**EPROM** Same as PROM. Electrically programmable read only memory that is nonvolatile and requires external devices to erase and reprogram. It is usually erased using UV light.

**erase** The process of clearing the information in a PROM or EEPROM, using electricity or UV light. The information bits are usually all set to logic 1.

**EVB** Evaluation Board, a Freescale product used to develop microcomputer software.

**even parity** A communication protocol whereby the number of ones in the data plus a parity bit is an even number. Contrast with odd parity.

**expanded mode** The mode in which some of the I/O ports are used to create an external data bus (control, address, data) allowing external memory to be connected.

**extended mode** More than the usual 16-bit address. The MC9S12C32 allows up to 20 address lines with its paged memory modes.

**external fragmentation** A condition when the largest file or memory block that can be allocated is less than the total amount of free space on the disk or memory.

**fan out** The number of inputs that a single output can drive if the devices are all in the same logic family.

**fast clear** A 9S12 timer mode in which the associated flag is automatically cleared when the timer register is accessed.

**Fast Fourier Transform (FFT)** A fast technique to convert data in the time domain to data in the frequency domain. N data points are sampled at  $f_s$ . The resulting frequency resolution is  $f_s/N$ . Mathematically, the FFT is the same as the DFT, just faster.

**FET** Field effect transistor, also JFET.

**filter** In the debugging context, a filter is a Boolean function or conditional test used to make run-time decisions. For example, if we print information only if two variables x,y are equal, then the conditional ( $x==y$ ) is a filter. Filters can involve hardware status as well. For example, if we halt when the serial port has an overrun error, then `(SCSR&0x08)` is the filter, and `if (SCSR&0x08) asm("swi");` would be the entire instrument.

**finite impulse response filter (FIR)** A digital filter in which the output is a function of a finite number of current and past data samples, but not a function of previous filter outputs.

**Finite State Machine (FSM)** An abstract design method to build a machine with inputs and outputs. The machine can be in one of a finite number of states. Which state the system is in represents memory of previous inputs. The output and next state are a function of the input. There may be time delays as well.

**fixed-point** A technique whereby calculations involving nonintegers are performed using a sequence of integer operations. For example,  $0.123*x$  is performed in decimal fixed-point as  $(123*x)/1000$  or in binary fixed-point as  $(126*x) \gg 10$ .

**flash EEPROM** Electrically erasable programmable read only memory that is nonvolatile and easy to reprogram. Flash EEPROMs are typically larger than regular EEPROM, and have fewer erase-reprogram cycles.

**floating** A logic state in which the output device does not drive high or pull low. The outputs of open collector and tristate devices can be in the floating state. Same as HiZ.

**floor** Establishing a lower bound on the result of an operation. *See also ceiling.*

**follower** An analog circuit with gain equal to 1, large input impedance and small output impedance. Same as voltage follower.

**frame** A complete and distinct packet of bits occurring in a serial communication channel.

**frame** The fixed structure in a relay or transducer. Contrast to armature.

**framing error** An error when the receiver expects a stop bit (1) and the input is 0.

**frequency response** The frequency at which the gain drops to 0.707 of the normal value. For a low-pass system, the frequency response ranges from 0 to a maximum value. For a high pass system, the frequency response ranges from a minimum value to infinity. For a bandpass system, the frequency response ranges from a minimum to a maximum value. Same as bandwidth.

**frequency shift key (FSK)** A modem that modulates the digital signals into frequency encoded sine waves.

**friendly** Friendly software modifies just the bits that need to be modified, leaving the other bits unchanged. Making it easier to combine modules.

**full-duplex channel** Hardware that allows bits (information, error checking, synchronization or overhead) to transfer simultaneously in both directions. Contrast with simplex and half-duplex channels.

**full-duplex communication** A system that allows information (data, characters) to transfer simultaneously in both directions.

**functional debugging** The process of detecting, locating, or correcting functional and logical errors in a program, typically not involving time. The process of instrumenting a program for such purposes is called functional debugging or often simply debugging.

**fuzzification** Conversion from the crisp inputs to the fuzzy logic input variables.

**fuzzy logic** Boolean logic (true/false) that can take on a range of values from true (255) to false (0). Fuzzy logic **and** is calculated as the minimum. Fuzzy logic **or** is the maximum.

**gadfly** A software/hardware synchronization method whereby the software continuously reads the hardware status waiting for the hardware operation to complete. The software usually performs no work while waiting for the hardware. Same as busy-waiting.

**gage factor** The sensitivity of a strain gage transducer (i.e., slope of the resistance versus displacement response).

**gibibyte (GiB)**  $2^{30}$  or 1,073,741,824 bytes.

**half-duplex channel** Hardware that allows bits (information, error checking, synchronization, or overhead) to transfer in both directions, but in only one direction at a time. Contrast with simplex and full-duplex channels.

**half-duplex communication** A system that allows information to transfer in both directions, but in only one direction at a time.

**handshake** A software/hardware synchronization method whereby control and status signals go both directions between the transmitter and receiver. The communication is interlocked meaning each device will wait for the other.

**hard real time** A system that can guarantee that a process will complete a critical task within a certain specified range. In data acquisition systems, hard real time means there is an upper bound on the latency between when a sample is supposed to be taken (every 1/fs) and when the ADC converter is actually started. Hard real time also implies that no ADC samples are missed.

**heartbeat** A debugging monitor, such as a flashing LED, we add for the purpose of seeing if our program is running.

**hexadecimal** A number system that uses base 16.

**hiZ** A logic state in which the output device does not drive high or pull low. The outputs of open collector and tristate devices can be in the floating state. Same as floating.

**hold time** When latching data into a device with a rising or falling edge of a clock, the hold time is the time after the active edge of the clock during which the data must continue to be valid. *See setup time.*

**hook** An indirect function call added to a software system that allows the user to attach their programs to run at strategic times. These attachments are created at run time and do not require recompiling the entire system.

**horizontal parity** A parity calculated across the entire message on a bit by bit basis (e.g., the horizontal parity bit 0 is the parity calculated on all the bit 0's of the entire message, can be even or odd parity).

**hysteresis** A condition when the output of a system depends not only on the input, but also on the previous outputs e.g. (a transducer that follows a different response curve when the input is increasing than when the input is decreasing).

**I/O bound** A situation where the input or output device is slower than the software. In other words, it takes longer for the I/O device to process data than for the software to process data. Contrast to CPU bound.

**I/O device** Hardware and software components capable of bringing information from the external environment into the computer (input device), or sending data out from the computer to the external environment (output device).

**I/O port** A hardware device that connects the internal software with external hardware.

**IEEE488** A medium-speed handshaking parallel I/O standard used for desktop instruments.

**I<sub>H</sub>** Input current when the signal is high.

**I<sub>L</sub>** Input current when the signal is low.

**immediate** An addressing mode in which the operand is a fixed data or address value.

**impedance loading** A condition when the input of stage  $n+1$  of an analog system affects the output of stage  $n$ , because the input impedance of stage  $n+1$  is too small and the output impedance of stage  $n$  is too large.

**impedance** The ratio of the effort (voltage, force, pressure) divided by the motion (current, velocity, flow).

**incremental control system** A control system where the actuator has many possible states, and the system increments or decrements the actuator value depending on whether an error is positive or negative.

**indexed** An addressing mode in which the data or address value for the instruction is located in memory pointed to by an index register.

**infinite impulse response filter (IIR)** A digital filter whose output is a function of an infinite number of past data samples, usually by making the filter output a function of previous filter outputs.

**inherent** An addressing mode in which there is no operand or the operand is implied (not explicitly stated).

**input bias current** Difference between currents of the two op amp inputs.

**input capture** A mechanism to set a flag and capture the current time (TCNT value) on the rising, falling, or rising and falling edge of an external signal. The input capture event can also request an interrupt.

**input impedance** Input voltage divided by input current.

**input noise current** Current noise referred to the op amp inputs.

**input noise voltage** Voltage noise referred to the op amp inputs.

**input offset current** Average current into the two op amp inputs.

**input offset voltage** Voltage difference between the two op amp inputs that makes the output zero.

**instruction register (IR)** Register in the control unit that contains the op code for the current instruction.

**instrument** An instrument is the code injected into a program for debugging or profiling. This code is usually extraneous to the normal function of a program and may be temporary or permanent. Instruments injected during interactive sessions are considered to be temporary, because these instruments can be removed simply by terminating a session. Instruments injected in source code are considered to be permanent, because removal requires editing and recompiling the source. An example of a temporary instrument occurs when the debugger replaces a regular op code with the swi instruction. This temporary instrument can be removed dynamically by restoring the original op code. A print statement added to your source code is an example of a permanent instrument, because removal requires editing and recompiling. An embedded system that collects information, same as data acquisition system.

**instrumentation** The debugging process of injecting or inserting an instrument.

**instrumentation amp** A differential amplifier analog circuit, which can have large gain, large input impedance, small output impedance, and a good common mode rejection ratio.

**internal fragmentation** Storage that is allocated for the convenience of the operating system but contains no information. This space is wasted.

**interrupt** A software/hardware synchronization method whereby the hardware causes a special software program (interrupt handler) to execute when its operation to complete. The software usually can perform other work while waiting for the hardware.

**interrupt flag** A status bit that is set by the timer hardware to signify that an external event has occurred.

**interrupt mask** A control bit that, if programmed to 1, will cause an interrupt request when the associated flag is set. Same as **arm**.

**interrupt service routine (ISR)** Program that runs as a result of an interrupt.

**interrupt vector** Sixteen-bit values at the end of memory specifying where the software should execute after an interrupt request. There is a unique interrupt vector for each type of interrupt including reset.

**intrusive** The debugger itself affects the program being tested. (*see* nonintrusive).

**Inverse Discrete Fourier Transform (IDFT)** A technique to convert data in the frequency domain to data in the time domain. There are  $N$  data points and the sampling rate is  $f_s$ . The resulting frequency resolution is  $f_s/N$ .

**invocation coupling** Module A is connected to Module B, because A calls B.

**I<sub>OH</sub>** Output current when the signal is high.

**I<sub>OL</sub>** Output current when the signal is low.

**IRQ** A interrupt mechanism on the 9S12.

**isolated I/O** A configuration in which the I/O devices are interfaced to the computer in a manner different from the way memories are connected. From an interfacing perspective, I/O devices and memory modules have separate bus signals; from a programmer's point of view, the I/O devices have their own I/O address map separate from the memory map and I/O device access requires the use of special I/O instructions.

**jerk** The change in acceleration; the derivative of the acceleration.

**Johnson noise** A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as thermal noise and white noise.

**Karnaugh map** tabular representation of the input/output relationship for a combinational digital function. The inputs possibilities are placed in the row and column labels, and the output values are placed inside the table.

**kibibyte (KiB)**  $2^{10}$  or 1024 bytes.

**latch** As a noun, it means a register. As a verb, it means to store data into the register.

**latched input port** An input port at which the signals are latched (saved) on an edge of an associated strobe signal.

**latency** In this book latency usually refers to the response time of the computer to external events. For example, the time between new input becoming available and the time the input is read by the computer, or the time between an output device becoming idle and the time the input is the computer writes new data to it. There can also be a latency for an I/O device, which is the response time of the external I/O device hardware to a software command.

**LCD** Liquid crystal display, whereby the computer controls the reflectance or transmittance of the liquid crystal, characterized by its flexible display patterns, low power, and slow speed.

**LED** Light emitting diode, whereby the computer controls the electrical power to the diode, characterized by its simple display patterns, medium power, and high speed.

**light-weight process** Same as a thread.

**linear filter** A filter whose output is a linear combination of its inputs.

**linear variable differential transformer (LVDT)** A transducer that converts position into electric voltage.

**little endian** Mechanism for storing multiple byte numbers such that the least significant byte exists first (in the smallest memory address). Contrast with big endian.

**loader** System software that places the object code into the microcomputer's memory. If the object code is stored in EPROM, the loader is also called an EPROM programmer.

**Local Area Network (LAN)** A connection between computers confined to a small space, such as a room or a building.

**logic analyzer** A hardware debugging tool that allows you to visualize many digital logic signals versus time. Real logic analyzers have at least 32 channels and can have up to 200 channels, with sophisticated techniques for triggering, saving, and analyzing the real-time data. In TExaS, logic analyzers have only 8 channels and simply plot digital signals versus time.

**LSB** The least significant bit in a number system is the bit with the smallest significance, usually the rightmost bit. With signed or unsigned integers the significance of the LSB is 1.

**maintenance** Process of verifying, changing, correcting, enhancing, and extending a system.

**mailbox** A formal communication structure, similar to a FIFO queue, where the source task puts data into the mailbox and the sink task gets data from the mailbox. The mailbox can hold at most one piece of data at a time, and has two states: mailbox has valid data or mailbox is empty.

**make before break** In a double-throw relay or double-throw switch, there is one common contact and two separate contacts. Make before break means as the common contact moves from one of separate contacts to another, it will make (finishing bouncing) the second contact before it breaks off (start bouncing) the first contact. A *form D* relay has a *make before break* operation.

**mark** A digital value of true or logic 1. Contrast with space.

**mask** As a verb, mask is the operation that selects certain bits out of many bits, using the logical and operation. The bits that are not being selected will be cleared to zero. When used as a noun, mask refers to the specific bits that are being selected.

**Mealy FSM** A FSM where both the output and next state are a function of the input and state.

**measurand** A signal measured by a data acquisition system.

**mebibyte (MiB)**  $2^{20}$  or 1,048,576 bytes.

**membership sets** Fuzzy logic variables that can take on a range of values from true (255) to false (0).

**memory** A computer component capable of storing and recalling information.

**memory-mapped I/O** A configuration in which the I/O devices are interfaced to the computer in a manner identical to the way memories are connected. From an interfacing perspective, I/O devices and memory modules share the same bus signals; from a programmer's point of view, the I/O devices exist as locations in the memory map, and I/O device access can be performed using any of the memory access instructions.

**microcomputer** A small electronic device capable of performing input/output functions, containing a microprocessor, memory, and I/O devices, where small means you can carry it.

**microcontroller** A single-chip microcomputer such as the Freescale 6811, Freescale 9S12, Intel 8051, PIC16, or the Texas Instruments TMS370.

**mnemonic** The symbolic name of an operation code (e.g., like `ldaa psha stx`).

**modem** An electronic device that MOdulates and DEModulates a communication signal. Used in serial communication across telephone lines.

**monitor or debugger window** A monitor is a debugger feature that allows users to passively view strategic software parameters during the real-time execution of a program. An effective monitor is one that has minimal effect on the performance of the system. When debugging software on a windows-based machine, we can often set up a debugger window that displays the current value of certain software variables.

**Moore FSM** A FSM where the both the output is only a function of the state and the next state is a function of the input and state.

**MOSFET** Metal oxide semiconductor field effect transistor.

**MSB** The most significant bit in a number system is the bit with the greatest significance, usually the leftmost bit. If the number system is signed, then the MSB signifies positive (0) or negative (1).

**multiple access circular queue (MACQ)** A data structure used in data acquisition systems to hold the current sample and a finite number of previous samples.

**multi-threaded** A system with multiple threads (e.g., main program and interrupt service routines) that cooperate towards a common overall goal.

**mutual exclusion or mutex** Thread synchronization whereby at most one thread at a time is allowed to enter.

**negative feedback** An analog system with negative gain feedback paths. These systems are often stable.

**negative logic** A signal where the true value has a lower voltage than the false value. In digital logic, true is 0 and false is 1; in TTL logic, true is less than 0.7 volts and false is greater than 2 volts; in RS232 protocol, true is -12 volts and false is +12 volts. Contrast with positive logic.

**negative predictive value (NPV)** is the probability the event is false if our device says it is false ( $TN/(TN + FN)$ ).

**nibble** Four binary bits or one hexadecimal digit.

**nonatomic** Software execution that can be divided or interrupted. Most lines of C code require multiple assembly language instructions to execute; therefore, an interrupt may occur in the middle of a line of C code.

**nonintrusive** A characteristic whereby the presence of the collection of information itself does not affect the parameters being measured. Nonintrusiveness is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as though the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in program performance by an instrument. For example, a print statement added to your source code and single-stepping are very intrusive because they significantly affect the real-time interaction of the hardware and software. When a program interacts with real-time events, the performance is significantly altered. On the other hand, an instrument that toggles an LED on and off (requiring just 10  $\mu$ s to execute) is much less intrusive. A logic analyzer that passively monitors the address and data is completely nonintrusive. An in-circuit emulator is also nonintrusive, because the software input/output relationships will be the same with and without the debugging tool.

**nonlinear filter** A filter in which the output is not a linear combination of its inputs. For example, median, minimum, maximum are examples of nonlinear filters. Contrast to linear filter.

**nonpreemptive scheduler** A scheduler that cannot suspend execution of a thread without the thread's permission. The thread must cooperate and suspend itself. Same as cooperative multitasking.

**nonreentrant** A software module that once started by one thread should not be interrupted and executed by a second thread. Nonreentrant modules usually involve nonatomic accesses to global variables or I/O ports: read modify write, write followed by read or a multistep write.

**nonvolatile** A condition whereby information is not lost when power is removed. When power is restored, then the information is in the state that occurred when the power was removed.

**null cycle** A computer bus cycle that fetches data at address \$FFFF, but the data is not used.

**Nyquist Theorem** If an input signal is captured by an ADC at the regular rate of  $f_s$  samples/sec, then the digital sequence can accurately represent the 0 to  $\frac{1}{2} f_s$  frequency components of the original signal.

**object code** Programs in machine-readable format created by the compiler or assembler. The S19 records contain object code.

**odd parity** A communication protocol in which the number of ones in the data plus a parity bit is an odd number. Contrast with even parity.

**op amp** An integrated analog component with two inputs, ( $V_2, V_1$ ) and an output ( $V_{out}$ ), where  $V_{out} = K \cdot (V_2 - V_1)$ . The amp has a very large gain,  $K$ . Same as operational amplifier.

**op code, opcode, or operation code** A specific instruction executed by the computer. The op code along with the operand completely specifies the function to be performed. In assembly language programming, the op code is represented by its mnemonic (e.g., ldaa). During execution, the op code is stored as a machine code loaded in memory. The ldaa instruction with immediate addressing has a machine code of \$86.

**open collector** A digital logic output that has two states, low and HiZ. Same as wire-or-mode.

**open loop control system** A control system that does not include sensors to measure the current state variables. An analog system with no feedback paths.

**operand** The second part of an instruction that specifies either the data or the address for that instruction. An assembly instruction typically has an op code (e.g., ldaa) and an operand (e.g., #55). Instructions that use inherent addressing mode have no operand field.

**operating system** System software for managing computer resources and facilitating common functions such input/output, memory management, and file system.

**originate modem** The device that places the telephone call.

**oscilloscope** A hardware debugging tool that allows you to visualize one or two analog signals versus time. In **TExaS**, oscilloscopes can plot up to eight channels.

**output compare** A mechanism to cause a flag to be set and an output pin to change when the TCNT matches a preset value. The output compare event can also request an interrupt.

**output impedance** Open circuit output voltage divided by short circuit output current.

**overflow** An error that occurs when the result of a calculation exceeds the range of the number system. For example, with 8-bit unsigned integers,  $200 + 57$  will yield the incorrect result of 1. Also, when TCNT increments from \$FFFF back to \$0000, setting the TOF flag. This overflow event can also request an interrupt.

**overrun error** An error that occurs when the receiver gets a new frame but the data register and shift register already have information.

**paged memory** A memory organization whereby logical addresses (used by software) have multiple and distinct components or fields. The number of bits in the least significant field defines the page size. The physical memory is usually continuous, having sequential addresses. There is a dynamic address translation (logical to physical).

**parallel port** A port at which all signals are available simultaneously. In this book the parallel ports are 8 bits wide.

**parallel programming** A software system that supports two or more programs being executed at the same time. A computer with multiple cores implements parallel programming.

**partially asynchronous bus** A communication protocol that has a central clock, but the memory module can dynamically extend the length of a bus cycle (cycle stretch) if it needs more time.

**path expression** A software technique to guarantee subfunctions within a module are executed in a proper sequence. For example, it forces the user to initialize I/O device before attempting to perform I/O.

**PC relative** An addressing mode in which the effective address is calculated by its position relative to the current value of the program counter.

**performance debugging or profiling** The process of acquiring or modifying timing characteristics and execution patterns of a program. The process of instrumenting a program for such purposes is called performance debugging or profiling.

**periodic polling** A software/hardware synchronization method that is a combination of interrupts and busy-waiting. An interrupt occurs at a regular rate (periodic) independent of the hardware status. The interrupt handler checks the hardware device (polls) to determine whether its operation is complete. The software usually can perform other work while waiting for the hardware.

**Personal Area Network (PAN)** A connection between computers controlled by a single person or all working toward a well-defined single task.

**phase shift key (PSK)** A protocol that encodes the information as phase changes between the sounds.

**photosensor** A transducer that converts reflected or transmitted light into electric current.

**physical plant** The physical device being controlled.

**PID controller** A control system where the actuator output depends on a linear combination of the current error (P), the integral of the error (I) and the derivative of the error (D).

**pink noise** A fundamental noise in resistive devices arising from fluctuating conductivity. Same as 1/f noise.

**pole** A place in the frequency domain where the filter gain is infinite.

**poll for zeros and ones** An interrupt handler that checks both for the interrupt flag and for the presence of other ones and zeros in the status register.

**polling** A software function to look and see which of the potential sources requested the interrupt.

**port** External pins through which the microcomputer can perform input/output. Same as I/O port.

**positive feedback** An analog system with positive gain feedback paths. These systems will saturate.

**positive logic** A signal in which the true value has a higher voltage than the false value. In digital logic, true is 1 and false is 0; in TTL logic, true is greater than 2 volts and false is less than 0.7 volts; in RS232 protocol, true is +12 volts and false is -12 volts. Contrast with negative logic.

**positive predictive value (PPV)** is the probability that an event is true when our device says it is ( $TP/(TP + FP)$ ).

**potentiometer** A transducer that converts position into electric resistance.

**precision** A term specifying the degree of freedom from random errors. For an input signal, it is the number of distinguishable input signals that can be reliably detected by the

measurement. For an output signal, it is the number of different output parameters that can be produced by the system. For a number system, precision is the number of distinct or different values of a number system in units of “alternatives.” The precision of a number system is also the number of binary digits required to represent all its numbers in units of “bits.”

**preemptive scheduler** A scheduler that has the power to suspend execution of a thread without the thread’s permission.

**priority** When two requests for service are made simultaneously, priority determines the order in which to process them.

**private** Can be accessed only by software modules in that local group.

**private variable** A global variable that is used by a single thread, and not shared with other threads.

**process** The execution of software that does not necessarily cooperate with other processes.

**producer-consumer** A multithreaded system in which the producers generate new data, and the consumers process or output the data.

**profiling** (*see* performance debugging).

**program counter (PC)** A register in the processor that points to the memory containing the instruction to execute next.

**PROM** Same as EPROM. Programmable read only memory that is nonvolatile and requires external devices to erase and reprogram. It is usually erased using UV light.

**promotion** Increasing the precision of a number for convenience or to avoid overflow errors during calculations.

**pseudo op** Operations included in the program that are not executed by the computer at run time, but rather are interpreted by the assembler during the assembly process. Same as assembly directive.

**pseudo-code** A shorthand for describing a software algorithm. The exact format is not defined, but many programmers use their favorite high-level language syntax (e.g., C) without paying rigorous attention to the punctuation.

**pseudo interrupt vector** A secondary place for the interrupt vectors for the convenience of the debugger, because the debugger can not or does not want the user to modify the real interrupt vectors. They provide flexibility for debugging but incur a run time delay during execution.

**public** Can be accessed by any software module.

**public variable** A global variable that is shared by multiple programs or threads.

**pulse-width modulation** A technique to deliver a variable signal (voltage, power, energy) using an on/off signal with a variable percentage of time the signal is on (duty cycle). Same as **variable duty cycle**.

**Q** The Q of a bandpass filter (passes  $f_{\min}$  to  $f_{\max}$ ) is the center pass frequency ( $f_o = (f_{\max} + f_{\min})/2$ ) divided by the width of the pass region,  $Q = f_o/(f_{\max} - f_{\min})$ . The Q of a bandreject filter (rejects  $f_{\min}$  to  $f_{\max}$ ) is the center reject frequency ( $f_o = (f_{\max} + f_{\min})/2$ ) divided by the width of the reject region,  $Q = f_o/(f_{\max} - f_{\min})$ .

**quadrature amplitude modem (QAM)** A protocol that uses both the phase and amplitude to encode up to 6 bits onto each baud.

**qualitative DAS** A DAS that collects information not in the form of numerical values, but rather in the form of the qualitative senses (e.g., sight, hearing, smell, taste, and touch). A qualitative DAS may also detect the presence or absence of conditions.

**quantitative DAS** A DAS that collects information in the form of numerical values.

**R/W signal** A bus signal specifying whether it is a read (R/W = 1) or a write (R/W = 0) cycle.

**RAM** Random Access Memory, a type of memory in which the information can be stored and retrieved easily and quickly. Since it is volatile, the information is lost when power is removed.

**range** Includes both the smallest possible and the largest possible signal (input or output).

The difference between the largest and smallest input that can be measured by the instrument. The units are in the units of the measurand. When precision is in alternatives, range = precision • resolution. Same as span.

**read cycle** Data flows from the memory or input device to the processor, the address bus specifies the memory or input device location, and the data bus contains the information at that address.

**read data available** The time interval (start, end) during which the data will be valid during a read cycle, determined by the memory module.

**real time** A system that can guarantee an upper bound (worst case) on latency.

**real-time computer system** A system in which time-critical operations occur when needed.

**recursion** A programming technique whereby a function calls itself.

**reentrant** A software module that can be started by one thread and interrupted and executed by a second thread. A reentrant module allow both threads to properly execute the desired function. Contrast with nonreentrant.

**registers** High-speed memory located in the processor. The registers on the 9S12 are CCR, A, B, X, Y, SP, and PC.

**relay** A mechanical switch that can be turned on and off by the computer.

**reliability** The ability of a system to operate within its specified parameters for a stated period of time. Given in terms of mean time between failures (MTBF).

**reproducibility (or repeatability)** A parameter specifying how consistent over time the measurement is when the input remains fixed.

**requirement** Detailed performance parameter that the system must satisfy, generally derived from the overall objective of the system.

**requirements document** A formal description of what the system will do in a very complete way, but not including how it will be done. It should be unambiguous, complete, verifiable, and modifiable.

**reset vector** The 16-bit value at memory locations \$FFFE and \$FFFF specifying where the software should start after power is turned on or after a hardware reset.

**resistance temperature device (RTD)** A linear transducer that converts temperature into electric resistance.

**resolution** For an input signal, the smallest change in the input parameter that can be reliably detected by the measurement. For an output signal, the smallest change in the output parameter that can be produced by the system; range equals precision times resolution. The units are in the units of the measurand. When precision is in alternatives, range = precision • resolution.

**response time** Similar to latency, it is the delay between the time an event occurs and the time the software responds to the event.

**ritual** Software, usually executed once at the beginning of the program, that defines the operational modes of the I/O ports.

**ROM** Read-Only Memory, a type of memory in which the information is programmed into the device once but can be accessed quickly. It is low cost, must be purchased in high volume, and can be programmed only once. See also EPROM, EEPROM, and flash EEPROM.

**rotor** The part of a motor that rotates.

**round-robin scheduler** A scheduler that runs each active thread equally.

**roundoff** The error that occurs in a fixed-point or floating-point calculation when the least significant bits of an intermediate calculation are discarded so that the result can fit into the finite precision.

**RTD** Resistance temperature device, a sensor used to measure temperature, usually made from platinum.

**sample and hold** A circuit used to latch a rapidly changing analog signal, capturing its input value and holding its output constant.

**sampling rate** The rate at which data is collected in a data acquisition system.

**saturation** A device that is no longer sensitive to its inputs when its input goes above a maximum value or below a minimum value.

**scan or ScanPoint** Any instrument used to produce a side effect without causing a break (halt) is a scan. Therefore, a scan may be used to gather data passively or to modify functions of a program. Examples include software added to your source code that simply outputs or modifies a global variable without halting. A ScanPoint is triggered in a manner similar to a breakpoint, but a ScanPoint simply records data at that time without halting execution.

**scheduler** System software that suspends and launches threads.

**Schmitt Trigger** A digital interface with hysteresis making it less susceptible to noise.

**scope** A logic analyzer or an oscilloscope, hardware debugging tool that allows you to visualize multiple digital or analog signals versus time.

**SCSI** Small Computer Systems Interface, a high-speed handshaking parallel I/O standard.

**select signal** The output of the address decoder (each module on the bus has a separate address decoder); a Boolean (true/false) signal specifying whether or not the current address of the bus matches the device address.

**semaphore** A system function with two operations (wait and signal) that provide for thread synchronization and resource sharing.

**sensitivity** The sensitivity of a transducer is the slope of the output versus input response. The sensitivity of a qualitative DAS that detects events is the percentage of actual events that are properly recognized by the system ( $TP/(TP + FN)$ ).

**serial communication** A process whereby information is transmitted one bit at a time.

**serial communications interface (SCI)** A device to transmit data with asynchronous serial communication protocol (same as UART and ACIA).

**serial peripheral interface (SPI)** A device to transmit data using the synchronous serial communication protocol.

**serial port** An I/O port with which the bits are input or output one at a time.

**servo** A DC motor with built in controller. The microcontroller specifies desired position and the servo adds/subtracts power to move the shaft to that position.

**setup time** When latching data into a register with a clock, the time before an edge at which the input must be valid. Contrast with hold time.

**shot noise** A fundamental noise that occurs in devices that count discrete events.

**signed two's complement binary** A mechanism to represent signed integers where 1 followed by all 0s is the most negative number, all 1s represents the value  $-1$ , all 0s represents the value 0, and 0 followed by all 1s is the largest positive number.

**sign-magnitude binary** A mechanism to represent signed integers whereby the most significant bit is set if the number is negative, and the remaining bits represent the magnitude as an unsigned binary.

**simple poll** An interrupt handler that simply checks the interrupt flag.

**simplex channel** Hardware that allows bits (information, error checking, synchronization, or overhead) to transfer only in one direction. Contrast with half-duplex and full-duplex channels.

**simplex communication** A system that allows information to transfer only in one direction.

**simulator** A simulator is a software application, such as **TEXaS**, that simulates or mimics the operation of a processor or computer system. Most simulators recreate only simple I/O ports and often do not effectively duplicate the real-time interactions of the software/hardware interface. On the other hand, they do provide a simple and interactive mechanism to test software. Simulators are especially useful when learning a new language, because they provide more control and access to the simulated machine, than one normally has with real hardware.

**single-address DMA** DMA that requires only one bus cycle to transfer data from an input device into memory, or from memory to an output device.

**single-pole relay** A simple relay with only one copy of the switch mechanism. Contrast with double pole.

**single-pole switch** A simple switch with only one copy of the switch mechanism. Contrast with double pole.

**single-throw switch** A switch with two contact connections. The two contacts may be connected or disconnected. Contrast with double-throw.

**slew rate** The maximum slope of a signal. If the time-varying signal  $x(t)$  is in volts, the slew rate is the maximum  $dx/dt$  in volts/s.

**Small Computer Systems Interface (SCSI)** a handshaking parallel data communication channel used to connect high-speed high-volume I/O devices to the computer.

**software interrupt vector** The 16-bit value at memory locations \$FFF6 and \$FFF7 specifying where the software should go after executing a software interrupt instruction, swi.

**software maintenance** Process of verifying, changing, correcting, enhancing, and extending software.

**solennoid** A discrete motion device (on/off) that can be controlled by the computer, usually by activating an electromagnet. For example, electronic door locks on automobiles.

**source code** Programs in human-readable format created with an editor.

**space** A digital value of false or logic 0. Contrast with mark.

**span** Same as range.

**spatial resolution** The volume over which the DAS collects information about the measurand.

**specificity** The specificity of a transducer is the relative sensitivity of the device to the signal of interest versus the sensitivity of the device to other unwanted signals. The sensitivity of a qualitative DAS is the fraction of properly handled non-events ( $TN/(TN + FP)$ ).

**spinlock semaphore** A semaphore in which the threads will spin (run but perform no useful function) when they execute wait on a busy semaphore. Contrast to blocking semaphore.

**stabilize** The debugging process of stabilizing a software system involves specifying all its inputs. When a system is stabilized, the output results are consistently repeatable. Stabilizing a system with multiple real-time events, such as input devices and time-dependent conditions, can be difficult to accomplish. It often involves replacing input hardware with sequential reads from an array or disk file.

**stack** Last-in-first-out data structure located in RAM and used to temporarily save information.

**stack pointer (SP)** A register in the processor that points to the RAM location of the stack.

**start bit** An overhead bit(s) specifying the beginning of the frame, used in serial communication to synchronize the receiver shift register with the transmitter clock. *See also stop bit, even parity, and odd parity.*

**starvation** A condition that occurs with a priority scheduler whereby low-priority threads are never run (*see aging*).

**static allocation** Data structures such as an FSM or TCB that are defined at assembly or compile time and exist throughout the life of the software. Contrast to dynamic allocation.

**static RAM** Volatile read/write storage built from three transistors having fast speed and not requiring refresh. Contrast with dynamic RAM.

**stator** The part of a motor that remains stationary. Same as frame.

**stepper motor** A motor that moves in discrete steps.

**stop bit** An overhead bit(s) specifying the end of the frame, used in serial communication to separate one frame from the next. *See also start bit, even parity, and odd parity.*

**strain gage** A transducer that converts displacement into electric resistance. It can also be used to measure force or pressure.

**string** A sequence of ASCII characters, usually terminated with a zero.

**symbol table** A mapping from a symbolic name to its corresponding 16-bit address, generated by the assembler in pass one and displayed in the listing file.

**synchronous bus** A communication protocol that has a central clock; there is no feedback from the memory to the processor, so every memory cycle takes exactly the same time; data transfers (put data on bus, take data off bus) are synchronized to the central clock.

**synchronous protocol** A system whereby the two devices share the same clock.

**tachometer** A sensor that measures the revolutions per second of a rotating shaft.

**thermal noise** A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as Johnson noise and white noise.

**thermistor** A nonlinear transducer that converts temperature into electric resistance.

**thermocouple** A transducer that converts temperature into electric voltage.

**thread** The execution of software that cooperates with other threads. A thread embodies the action of the software. One concept describes a thread as the sequence of operations including the input and output data.

**thread control block (TCB)** Information about each thread.

**three-pole relay** Three separate and complete relays, which are activated together. *See also single pole.*

**three-pole switch** Three separate and complete switches, which are activated together. *See also single pole.*

**throughput** The information transfer rate; the amount of data transferred per second. Same as bandwidth.

**time constant** The time to reach 63.2% of the final output after the input is instantaneously increased.

**time profile and execution profile** Time profile refers to the timing characteristic of a program, and execution profile refers to the execution pattern of a program.

**timing signals** The lines used to clock data onto or off of the bus; signals that specify when to activate during this cycle; the specific times for the rise and fall edges are synchronized to the E clock. Contrast to command signals.

**tolerance** The maximum deviation of a parameter from a specified value.

**total harmonic distortion (THD)** A measure of the harmonic distortion present and is defined as the ratio of the sum of the powers of all harmonic components to the power of the fundamental frequency.

**transducer** A device that converts one type of signal into another type.

**tristate** The state of a tristate logic output when off or not driven.

**tristate logic** A digital logic device that has three output states: low, high, and off (HiZ).

**truncation** The act of discarding bits as a number is converted from one format to another.

**two's complement** A number system used to define signed integers. The MSB defines whether the number is negative (1) or positive (0). To negate a two's complement number, one first complements (flips from 0 to 1 or from 1 to 0) each bit, then add 1 to the number.

**two-pole relay** Two separate and complete relays, which are activated together; same as double-pole.

**two-pole switch** Two separate and complete switches, which are activated together; same as double-pole.

**ultrasound** A sound with a frequency too high to be heard by humans, typically 40 kHz to 100 MHz.

**unbuffered I/O** The hardware and software are tightly coupled so that both wait for each other during the transmission of data.

**unipolar stepper motor** A stepper motor in which the current flows in only one direction (on/off) along the interface wires; a stepper with five or six interface wires.

**universal asynchronous receiver/transmitter (UART)** A device to transmit data with asynchronous serial communication protocol, same as SCI and ACIA.

**unsigned binary** A mechanism to represent unsigned integers where all 0s represents the value 0, and all 1s represents the largest positive number.

**vector** An address at the end of memory containing the location of the interrupt service routines. *See also* reset vector and interrupt vector.

**velocity factor (VF)** The ratio of the speed at which information travels relative to the speed of light.

**vertical parity** The normal parity bit calculated on each individual frame; can be even or odd parity.

**V<sub>OH</sub>** The smallest possible output voltage when the signal is high.

**V<sub>OL</sub>** The largest possible output voltage when the signal is low.

**volatile** A condition whereby information is lost when power is removed.

**voltage follower** An analog circuit with gain equal to 1, large input impedance, and small output impedance. Same as follower.

**volatile** A property of a variable in C, such that the value of the variable can change outside the immediate scope of the software accessing the variable.

**vulnerable window** Locations within a software module. If an interrupt were to occur at one of these locations, then an error could occur (e.g., data lost, corrupted data, program crash). Same as critical section.

**white noise** A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as Johnson noise and thermal noise.

**wire-or-mode** A digital logic output that has two states: low and HiZ. Same as open collector.

**word** Two bytes containing 16 bits. Same as double byte.

**workstation** A powerful general-purpose computer system having a price in the \$10K to 50K range and used for handling large amounts of data and performing many calculations.

**write cycle** Data is sent from the processor to the memory or output device, the address bus specifies the memory or output device location, and the data bus contains the information.

**write data available** Time interval (start, end) during which the data will be valid during a write cycle, determined by the processor.

**write data required** Time interval (start, end) during which the data should be valid during a write cycle, determined by the memory module.

**XIRQ** A high-priority interrupt mechanism available on the 9S12.

**XON/XOFF** A protocol used by printers to feedback the printer status to the computer. XOFF is sent from the printer to the computer in order to stop data transfer, and XON is sent from the printer to the computer in order to resume data transfer.

**Z Transform** A transform equation converting a digital time-domain sequence into the frequency domain. In both the time and frequency domain it is assumed the signal is band limited to 0 to  $\frac{1}{2}$  fs.

**zero** A place in the frequency domain where the filter gain is zero.

# Appendix 2

## Solutions to Checkpoints

**Checkpoint 1.1:** A microcomputer is a small computer that includes a processor memory and input/output. A microprocessor is a small processor that includes registers, ALU, a control unit, and a bus interface unit. A microcontroller is a single-chip microcomputer.

**Checkpoint 1.2:** An input port is hardware that is part of the computer and the channel through which information enters into the computer. An input interface includes hardware components external to the computer, the input port, and software, which together perform the input function.

**Checkpoint 1.3:** Typical input devices include the keys on the keyboard, the mouse and its buttons, joystick, CD reader, and microphone. The floppy disk can be used for input and output.

**Checkpoint 1.4:** Typical output devices include the LEDs on the keyboard, monitor, speaker, printer, CD burner, and speaker. The floppy disk can be used for input and output.

**Checkpoint 1.5:** CU (control unit), BIU (bus interface unit), and ALU (arithmetic logic unit) are all part of the processor. DMA stands for direct memory access, which is a high-speed mechanism to move data directly from input to memory, or move data directly from memory to output.

**Checkpoint 1.6:** An embedded system is a microcomputer with mechanical, chemical, and electrical devices attached to it, programmed for a specific dedicated purpose, and packaged as a complete system.

**Checkpoint 1.7:** The software in the alarm clock must maintain time using a real-time clock, output the current time on the display, respond to button pushes updating parameters as required, and check to see whether the current time matches the alarm time.

**Checkpoint 1.8:** A requirement is a detailed performance parameter that the system must satisfy, generally derived from the overall objective of the system. A constraint is a condition defining how the system will be developed, generally restricting the range of solutions from which the system will be built.

**Checkpoint 1.9:** If two modules output to the same port, then the second module will undo the function of the first one. For example, if one module says “go” and the other one says “stop,” then the order of execution determines the resulting function. A similar error can occur for input ports.

**Checkpoint 1.10:**  $V_{IL}$  is 1.5 V, and  $V_{IH}$  is 3.5 V. Therefore, 0 and 1 V are considered low, and 4 and 5 V are considered high. 2 and 3 V are indeterminate.

**Checkpoint 1.11:** Yes,  $V_{OL}(0.5\text{ V})$  is less than  $V_{IL}(0.8\text{ V})$ ,  $V_{OH}(4.4\text{ V})$  is greater than  $V_{IH}(2\text{ V})$ ,  $I_{OH}(4\text{ mA})$  is greater than  $I_{IH}(20\text{ }\mu\text{A})$ , and  $I_{OL}(4\text{ mA})$  is greater than  $I_{IL}(0.4\text{ mA})$ .

**Checkpoint 1.12:** No, because  $V_{OH}(2.4\text{ V})$  is less than  $V_{IH}(3.5\text{ V})$ . To fix this situation we can add a pull-up resistor on the output of the 74LS04 gate. In fact, the 9S12 has a feature allowing you to enable internal pull-up resistors on its inputs (see the PERT register).

**Checkpoint 1.13:**  $2^{1/2}$  decimal digits is 200 alternatives, which is about 8 bits.

**Checkpoint 1.14:** The rule of thumb says  $2^{60}$  is about  $10^{18}$ , which is 18 decimal digits.  $2^4$  is 16, which is about 1½ decimal digits. Together, we have 19½ decimal digits.

**Checkpoint 1.15:** First, break into nibbles %1110, %1110, %1011, then convert each, \$EEB.

**Checkpoint 1.16:** First, convert each hex digit one at a time 0011 1000 0000 0000, then combine to get %0011000000000000.

**Checkpoint 1.17:** Each hex digit needs 4 bits, so a total of 20 bits will be required.

**Checkpoint 1.18:** %01101010 equals  $64 + 32 + 8 + 2 = 106$ .

**Checkpoint 1.19:** \$32 equals  $3 \cdot 16 + 2 = 50$ .

**Checkpoint 1.20:** 35 equals  $32 + 2 + 1 = \%00100011=\$23$ .

**Checkpoint 1.21:** 200 equals  $128 + 64 + 8 = \%11001000=\$C8$ .

**Checkpoint 1.22:** They are the same, both equally 53.

**Checkpoint 1.23:** -35 equals  $-128 + 64 + 16 + 8 + 4 + 1 = \%11011101=\$DD$ .

**Checkpoint 1.24:** The range of 8-bit signed numbers is -128 to +127.

**Checkpoint 1.25:** The character '0' is represented in ASCII as \$30.

**Checkpoint 1.26:** Converting each character to ASCII yields

"48656C6C6F20576F726C6400"

**Checkpoint 1.27:**  $\pi$  is about 3142, with a resolution of 0.001.

**Checkpoint 1.28:**  $\pi$  is about 804, with a resolution of 1/256.

**Checkpoint 1.29:**  $y = (1000 \cdot x - 53 \cdot x_1 + 1000 \cdot x_2 + 51 \cdot y_1 - 903 \cdot y_2)/1000$ .

**Checkpoint 1.30:** We set the I bit to 1 to disable interrupts.

**Checkpoint 1.31:** Registers D, X, Y, SP, and PC can hold 16-bit addresses.

**Checkpoint 1.32:** The addressing mode specifies where the instruction will read or write data.

**Checkpoint 1.33:** They have the same machine code and thus perform the same function when executed. The only difference is programming style, or how it looks to the programmer.

**Checkpoint 1.34:** The instruction ldaa #36 puts the number 36 into Register A, whereas the instruction ldaa 36 fetches the 8-bit data from memory location 36 and puts those data into Register A.

**Checkpoint 1.35:** The instruction ldx #\$0801 puts the number \$0801 into Register X, whereas the instruction ldx \$0801 fetches the 16-bit data from memory locations \$0801, \$0802 and puts those data into Register X.

**Checkpoint 1.36:** The instruction ldaa \$12 fetches the 8-bit data from memory location \$12 and puts that data into Register A, whereas the instruction ldx \$12 fetches the 16-bit data from memory locations \$0012, \$0013 and puts those data into Register X.

**Checkpoint 1.37:** The 9 means flash EEPROM, and the 64 means 64 K bytes.

**Checkpoint 1.38:** Nothing happens if the software writes to an input port.

**Checkpoint 1.39:** This is the way the 6811 output pins operate. If the software reads an output port, it will retrieve the value on the external pin. On the 9S12, when the software reads from an output port, it will get the value that was previously written to it.

**Checkpoint 1.40:** Since there are 8 bits in a port and 8 bits in the direction register, each bit can be individually programmed as input or output.

**Checkpoint 1.41:** It will still operate according to specifications, but it may be more expensive to build or it may be harder to order components to build it.

**Checkpoint 1.42:** It will no longer operate according to specifications.

**Checkpoint 1.43:** Our eyes can process visual information only at frequencies less than about 10–15 Hz. Information occurring at rates faster than that cutoff is seen as an average value. If you want to see the LEDs with your eyes, you will have to slow it down, placing four delays, one after each output. The value of the delay should be greater than 0.1 sec.

**Checkpoint 2.1:** The assembler builds the symbol table.

**Checkpoint 2.2:** The assembler creates the object code and listing file.

**Checkpoint 2.3:** leay 10,x

**Checkpoint 2.4:** Either

```
xgdx
addd #2000
xgdx
```

or

```
leay 2000,y
```

**Checkpoint 2.5:** Since the values in Registers A and B range from 0 to 255, their product must range from 0 to 65025. Therefore, all potential results will fit in Register D.

**Checkpoint 2.6:** dividend = quotient\*divisor + remainder.

**Checkpoint 2.7:**

```
ldaa N
ldab #10
mul      ;D=10*N
ldx #51
idiv      ;X=(10*N)/51
xgdx
stab M
```

**Checkpoint 2.8:**

```
ldd N      ;2.5 is about 65536/26214
ldx #26214
fdiv      ;RegX=(65536*N)/26214
stx M
```

**Checkpoint 2.9:**

```
ldaa PORTB
anda #$EF  ;clear bit 4
staa PORTB
```

**Checkpoint 2.10:**

```
ldaa PORTB
ora a #$08  ;set bit 3
staa PORTB
```

**Checkpoint 2.11:**

```
ldaa M
psha
ldaa N
staa M
pula
staa N
```

**Checkpoint 2.12:** It determines whether to use the **bls** or **ble** instruction. That is,

<pre>;unsigned version ldaa N cmpa #25 <b>bls</b> next ; skip if N&lt;=25 jsr isGreater ; N&gt;25 next</pre>	<pre>;signed version ldaa N cmpa #25 <b>ble</b> next ; skip if N&lt;=25 jsr isGreater ; N&gt;25 next</pre>
--	--

**Checkpoint 2.13:** The assembler must create the symbol table during pass 1, so it must know the size of each assembly line during pass 1. A forward reference would prevent the assembler from knowing how many bytes to allocate during pass 1. It would probably create a phasing error.

**Checkpoint 2.14:** Public functions have an underline (e.g., SCI\_OutString). Private functions have no underline in the name.

**Checkpoint 2.15:** Local variables begin with a lower-case letter (e.g., myKey). Global variables begin with an upper case letter (e.g., TheKey).

**Checkpoint 2.16:** In a Moore FSM, the output depends only on the state. In a Mealy FSM, the output depends on the state and the input.

**Checkpoint 2.17:** The period of the TCNT timer will be the resolution of the delay function. The maximum delay for the assembly version is 32767 times the resolution, and the maximum delay for the C implementation will be 65,000 times the resolution.

**Checkpoint 2.18:** It will first sit up, then it will stand up.

**Checkpoint 2.19:** Define the variable within the scope of the function. For example,

```
void MyFunction(void){ short myLocalVariable;
}
```

**Checkpoint 2.20:** Define the variable outside the scope of the function. For example,

```
short MyGlobalVariable; // accessible by all programs
void MyFunction(void){
}
```

**Checkpoint 2.21:** A **static** local has permanent allocation, which means it maintains its value from one call to the next. It is still local in scope, meaning it is accessible only from within the function. In the following example, count contains the number of times MyFunction is called:

```
void MyFunction(void){ static short count;
count++;
}
```

A **static** global reduces scope. Regular globals can be accessed from any function in the system, whereas a **static** global can be accessed only by functions within the same file. Static globals are private. Functions can be static also, meaning they can be called only from other functions in the file. For example,

```
static short myPrivateGlobalVariable; // accessible by this
void static MyPrivateFunction(void){ // file only
}
```

**Checkpoint 2.22:** A `const` global is read-only. It is allocated in the ROM portion of memory. When `const` modifies a function parameter, it means that parameter can't be changed by the function. Function parameters are always allocated in registers or on the stack.

**Checkpoint 2.23:** You don't explicitly define the size of the stack. It can be implicitly calculated if the global variables are contiguously allocated starting at the beginning of RAM and the stack pointer is initialized to the end of RAM. Then, the stack size is defined as the total RAM bytes minus the size of the global variables. Some systems have a heap, which is used by `malloc` and `free` and also is defined in RAM.

**Checkpoint 2.24:** The local variable name can be reused in other subroutines, just as in C.

**Checkpoint 2.25:**

<pre>; One possibility pshx psha    allocate 3 locals ; access locals ins ins ins    deallocate</pre>	<pre>; Another possibility leas -3,sp allocate 3 locals ; access locals leas 3,sp deallocate locals</pre>
---	---

**Checkpoint 2.26:**

```
psha      ; save registers
ldaa PORTA
eora #\$08 ; bit 3
staa PORTA
pula      ; restore
```

**Checkpoint 3.1:** Negative logic means pushing the switch makes the signal low. Positive logic means pushing the switch makes the signal high.

**Checkpoint 3.2:** If we are not worrying about being friendly, we can simply make

```
DDRJ=0 because PJ6 and PJ7 are inputs
PPSJ=0x80 because PJ7 needs a pull-down, and PJ6 needs a pull-up
PERJ=0xC0 in order to activate pull up/down on PJ6 and PJ7
```

The following is friendly assembly code

```
bclr DDRJ,#$C0 ; because PJ6 and PJ7 are inputs
bset PPSJ,#$80 ; because PJ7 needs a pull-down
bclr PPSJ,#$40 ; because PJ6 needs a pull-up
bclr PERJ,#$C0 ; activates pulls on PJ6 and PJ7
```

**Checkpoint 3.3:** Any of the ten key wakeup pins on the 9S12C32 or twenty key wakeup pins on the 9S12DP512.

**Checkpoint 3.4:** Port P outputs are working properly if PTP bits equal the PTIP bits.

**Checkpoint 3.5:** In C, we execute `PIFJ=0x80;` and in assembly, `movb #$80,PIFJ.`

**Checkpoint 3.6:** It clears all the bits in the flag register, because of the read-modify-write sequence.

**Checkpoint 3.7:** The `Clk` pulse is 4 cycles wide. At 4 MHz, each cycle is 250 ns, so the pulse will be 1  $\mu$ s wide. It still works because the setup and hold times are still satisfied.

**Checkpoint 3.8:** Out before In

<pre> // Transmitter 1) PTT = data; // 1)Data out 2) PTJ  = 0x40; // 2)Ready=1 3) while((PTJ&amp;0x80)); // wait 3)  6) while((PTJ&amp;0x80)==0); // wait 5)  9) PTJ &amp;= ~0x40; // 6)Ready=0 </pre>	<pre> // Receiver  4) while((PTJ&amp;0x80)==0); // wait 2) 5) PTJ &amp;= ~0x40; // 3)Ack=0  7) data = PTT; // 4)read data 8) PTJ  =0x40; // 5)Ack=1  10) while((PTJ&amp;0x80)); // wait 6) 11) return(data);} </pre>
--	--

In before Out

<pre> // Transmitter  2) PTT = data; // 1)Data out 3) PTJ  = 0x40; // 2)Ready=1 4) while((PTJ&amp;0x80)); // wait 3)  6) while((PTJ&amp;0x80)==0); // wait 5)  9) PTJ &amp;= ~0x40; // 6)Ready=0 </pre>	<pre> // Receiver 1) while((PTJ&amp;0x80)==0); // wait 2)  5) PTJ &amp;= ~0x40; // 3)Ack=0  7) data = PTT; // 4)read data 8) PTJ  =0x40; // 5)Ack=1  10) while((PTJ&amp;0x80)); // wait 6) 11) return(data);} </pre>
---	--

**Checkpoint 3.9:** The bandwidth equals  $9600/10 = 960$  bytes/sec.**Checkpoint 3.10:**  $4,000,000/(16*13) = 19231$ , which is about 19200 bits/sec.**Checkpoint 3.11:** SCIBD = 41.  $25,000,000/(16*41) = 38110$  bits/sec, which is about 38400 bits/sec.**Checkpoint 3.12:** The ldab instruction will set the N bit with TDRE:

<pre> SCI_OutChar ldab SCSR    output ready? bpl SCI_OutChar wait for TDRE staa SCDR    write ASCII rts </pre>	<pre> SCI_OutChar ldab SCISR1 ; output ready? bpl SCI_OutChar ; wait staa SCIDRL ; write ASCII rts </pre>
--	---

**Checkpoint 3.13:** Read status register with RDRF set, followed by read SCI data register.**Checkpoint 3.14:** Read status register with TDRE set, followed by write SCI data register.**Checkpoint 3.15:** Each transmitted frame will be received as two frames, both wrong and the first one might have a framing error.**Checkpoint 3.16:** Each transmitted frame will be received as one frame, which will be wrong and might set the noise flag. In fact, two transmitted frames might be received as one frame, which will be wrong.**Checkpoint 4.1:** I = 0 enables interrupts.**Checkpoint 4.2:** If the average rate at which data input into the FIFO is less than the average rate at which data are removed, then increasing the FIFO size will solve a full error.

If the average rate at which data input into the FIFO is more than the average rate at which data are removed, the FIFO will become full regardless of its size.

**Checkpoint 4.3:** The bandwidth of the input is slow compared to the speed of the computer, so it is I/O bound.

**Checkpoint 4.4:** The bandwidth of the output is fast compared to the speed of the computer, so it is CPU bound.

**Checkpoint 4.5:** First CCR is pushed (with I = 0), then I = 1.

**Checkpoint 4.6:** What makes XIRQ interrupts such a high priority is that once enabled (X = 0), they cannot be disabled. This means an XIRQ will interrupt anywhere, even inside an IRQ interrupt service routine. On the other hand, an IRQ interrupt cannot interrupt an XIRQ interrupt service routine.

**Checkpoint 4.7:** The rti instruction takes 8 cycles if no other interrupt is pending and 11 cycles if another interrupt is armed and triggered.

**Checkpoint 4.8:** The ISR calls `Fifo_Put`, and there are 10 or 11 instructions required to execute this subroutine.

**Checkpoint 4.9:** We simply take the 32 cycles and remove the 19-cycle overhead. This results in an estimate of 13 cycles or 1.625 µs, plus the time to finish the current instruction.

**Checkpoint 4.10:** In a typical system, the largest component to latency is the time running with I = 1, such as while inside other ISRs.

**Checkpoint 4.11:** RTICTL=0x71; or RTICTL=0x63; or RTICTL=0x47; or RTICTL=0x4F;

**Checkpoint 4.12:** Change `TSCR2=0x07`; to make a 16 µsec TCNT and `#define PERIOD 62500`

**Checkpoint 5.1:** A program is a list of commands, whereas a thread is the action caused by the execution of software. For example, there might be one copy of a program that searches the card catalog of a library, while separate threads are created for each user who logs into a terminal to perform a search. Similarly, there might be one set of programs that implement the features of a window (open, minimize, maximize, etc.), while there will be a separate thread for each window created.

**Checkpoint 5.2:** Threads can't communicate with each other using the stack, because they have logically separate stacks. Global variables will be used, because one thread may write to the global, and another can read from it.

**Checkpoint 5.3:** It is efficient not to schedule a thread when it cannot continue executing at this point (it needs something not currently available). In this way, we schedule only threads that can produce work.

**Checkpoint 5.4:** A blocked thread is placed back in the ready queue when the output display becomes available.

**Checkpoint 5.5:** Yes, PT0 is high during the context switch. This means Thread 3 was running `sub` at that time. Thread 1 then calls `sub`, so `sub` has been reentered.

**Checkpoint 5.6:** The `RunPt` points to the TCB of the thread currently being executed.

**Checkpoint 5.7:** The `sts` instruction suspends the current thread, and the `lds` instruction will activate the next thread.

**Checkpoint 5.8:** It will be suspended (its time slice is over), and the next thread will run. It will essentially sleep until its next turn in the queue. It is a simple way to implement cooperative multitasking.

**Checkpoint 5.9:** If the semaphore is zero, it will crash because the semaphore will never be incremented (signal cannot be called).

**Checkpoint 5.10:** There is a read-modify-write critical section. One thread could call `wait` and read the semaphore. At this point there are two copies of the semaphore, one in

memory and the other copy in a register of the first thread. If a thread switch interrupt occurs, then a second thread calls wait, decrements the semaphore, and continues. When the first thread is run again, it does not refetch the semaphore, from memory; rather, it will use the copy in the register. Both threads called wait, but the memory copy of the semaphore got decremented only once.

**Checkpoint 5.11:** On average, over a long period of time, the number of calls to wait equals the number of calls to signal.

**Checkpoint 5.12:** In Program 5.17, the stack pointer is the first entry of the TCB, whereas in the previous examples it was the second entry.

**Checkpoint 5.13:** Since each instruction runs six times faster, the time-jitter will be reduced by 1/6.

**Checkpoint 6.1:** An input capture event occurs on the selected edge on an input pin.

**Checkpoint 6.2:** TCNT is copied into the input capture latch, the flag is set, and, if armed, an interrupt is requested.

**Checkpoint 6.3:** Clear bit 0 of TIOS to specify input capture on channel 0. Clear bit 0 of DDRT to make it an input. Set TEN to enable the timer. For the rising edge, make EDG0B, EDG0A equal to 0, 1. To arm the channel, set COI = 1.

**Checkpoint 6.4:** Clear bit 3 of TIOS to specify input capture on channel 3. Clear bit 3 of DDRT to make it an input. Set TEN to enable the timer. For the falling edge, make EDG3B, EDG3A equal to 1, 0. To arm the channel, set C3I = 1.

**Checkpoint 6.5:**

movb #\$40 ,TFLG1	TFLG1=0x40 ;
-------------------	--------------

**Checkpoint 6.6:** Change the TCNT period to 2  $\mu$ s.

TSCR2=0x03; // MC9S12C32 with 4MHz E clock

**Checkpoint 6.7:** Resolution\*65535 equals the maximum pulse-width.

**Checkpoint 6.8:** An output compare event occurs when the TCNT value equals the value in the output compare register.

**Checkpoint 6.9:** The output can change (set high, clear low, or toggle), the flag is set, and, if armed, it will request an interrupt.

**Checkpoint 6.10:**

movb #\$10 ,TFLG1	TFLG1=0x10 ;
-------------------	--------------

**Checkpoint 6.11:** This is a read-modify-write operation. It first reads the register, and any flag that is set will return a 1 in its bit location, the logical or will set bit 5, then when the result is written back, all flags will be cleared. Writing ones to this register clears the flags.

**Checkpoint 6.12:** Set bit 0 of TIOS to specify output compare on channel 0. Set bit 0 of DDRT to make it an output. Set TEN to enable the timer. For the toggle output, make OM0 = 0 and OL0 = 1. To arm the channel, set COI = 1.

**Checkpoint 6.13:** In the existing program, High + Low equals 10000. You can change it so High + Low equals 40000.

**Checkpoint 6.14:** 1 Hz. If the frequency changes from 100 to 101 Hz, the system can detect the change.

**Checkpoint 6.15:** The existing system counts for 1 second. In general, the frequency resolution is 1 divided by the wait time. To make a 1 kHz resolution, count pulses for 1 ms.

**Checkpoint 6.16:** The resolution is 16  $\mu$ s. If the period of the input wave changes by more than a resolution, system can detect the change.

**Checkpoint 6.17:** The result will be  $1234.5/16 = 77$ .

**Checkpoint 6.18:** ( $A = E/8, SA = A/20$ ) or ( $A = E/4, SA = A/40$ ) or ( $A = E/2, SA = A/80$ ) or ( $A = E, SA = A/160$ ).

**Checkpoint 6.19:** Change  $PWMSCLA=5$ ; to  $PWMSCLA=50$ ;

**Checkpoint 6.20:** The precision would be reduced from 16 bits (62501 alternatives) to 10 bits (1001 alternatives).

**Checkpoint 6.21:** Yes, one uses Clock A and the other uses Clock B.

**Checkpoint 7.1:** Because every RS232 protocol has one start bit, there are 10 bits in this frame.

**Checkpoint 7.2:** Full-duplex allows communication in both directions simultaneously, whereas half-duplex allows communication in both directions but only one direction at a time.

**Checkpoint 7.3:** With synchronous serial (SPI), the transmitter and receiver operate on the same clock, which is included in the cable. With asynchronous serial (SCI), the transmitter and receiver operate with clocks of similar frequencies, and the receiver uses transitions in the data to synchronize with the transmitter.

**Checkpoint 7.4:** If the baud rates are more than 5% different, there is a good chance the receiver will get framing and noise errors.

**Checkpoint 7.5:** Read status register with RDRF set, followed by read SCI data register.

**Checkpoint 7.6:** Read status register with TDRE set, followed by write SCI data register.

**Checkpoint 7.7:** Because the TxFifo is empty, meaning there are no data to output.

**Checkpoint 7.8:** Because it is in the ISR, running with interrupts disabled, meaning no other software could run that might empty the RxFifo.

**Checkpoint 7.9:**

```
SCIBD = 208; // 6812 1200 bits/sec
```

**Checkpoint 7.10:** Read status register with SPIF set, followed by read SPI data register.

**Checkpoint 7.11:** Connect SS-SS, SCLK-SCLK, MOSI-MOSI, and MISO-MISO. Configure one as a master and the other as a slave. It doesn't matter which mode you use for CPOL CPHA as long as they are the same. Similarly, it doesn't matter which baud rate is used, but there is no need not to use the fastest available rate. First, the slave writes; then, when the master writes, the two SPDR registers are exchanged.

**Checkpoint 7.12:** The rise time of the open collector signal will follow an R-C relationship. As the number of devices increases so does the capacitance. It will rise faster with a smaller R.

**Checkpoint 7.13:** If no device sends an acknowledgement, the SDA signal will float high, generating a negative acknowledgment.

**Checkpoint 7.14:** If they both send the same address and the same sequence of data bits, both will finish without getting a lost-arbitration error. If they both send the same address but different data values, an arbitration will occur during the data transfer, and the master with the smaller data value will win arbitration.

**Checkpoint 7.15:**  $t_E/t_{bit}$  is  $24000/100 = 240$ . The only solution is to make IBFD equal to \$1F, making MUL = 1, scl2start = 6, scl2stop = 9, scl2tap = 6, tap2tap = 8, and SCLTap = 15

$$\begin{aligned} MUL \cdot \{2 \cdot (scl2tap + [(SCLTap - 1) \cdot tap2tap] + 2)\} \\ = 1 \cdot \{2 \cdot (6 + [(15 - 1) \cdot 8] + 2)\} = 240 \end{aligned}$$

**Checkpoint 8.1:**

```
DDRM = 0x00; // PM5-0 inputs
PPSM = 0x00; // select pull-up
PERM = 0x3F; // enable pull-up
```

**Checkpoint 8.2:** The solution will be a sinusoid with frequency  $\sqrt{K/m}/(2\pi)$ .

**Checkpoint 8.3:** No, the bounce time is a function of the mass, friction, and spring constant of the switch.

**Checkpoint 8.4:** Since each key needs to be interfaced separately, it will take 88 pins.

**Checkpoint 8.5:** We will have 10 rows and 10 columns, so it will take 20 pins.

**Checkpoint 8.6:** Because the Ctrl, Alt, and Shift keys are simultaneously pressed, they need to be interfaced directly. The remaining 100 keys can easily be interfaced with a 10 by 10 scanned matrix.

**Checkpoint 8.7:** Because you touch just one point at a time, we can use a multiplexed/demultiplexed scanned matrix. A 10 to 1024 multiplexer will drive the rows, and a 1024 to 10 demultiplexer will sense the columns.

**Checkpoint 8.8:**  $R = (5-V_d-V_{OL})/I_d = (5-2.5-0.5)/0.02 = 150 \Omega$ .

**Checkpoint 8.9:**  $R = (4.5 - 2)/0.001 = 2500 \Omega$ .

**Checkpoint 8.10:** Since it is a DC analysis we ignore the inductor (which is modeled as a short). The total voltage across the resistance will be 25 V, so  $25 \text{ V}/50 \Omega = 0.5 \text{ A}$  will flow.

**Checkpoint 8.11:**  $I_b$  needs to be  $150 \text{ mA}/100 = 1.5 \text{ mA}$ . Assuming  $V_{OH}$  is 5 V and  $V_{BE} = 0.6 \text{ V}$ , we calculate  $R_b = (5-0.6 \text{ V})/1.5 \text{ mA} = 2.9 \text{ k}\Omega$ . Picking a value 2 to 5 times smaller, we could use  $1 \text{ k}\Omega$ .

**Checkpoint 8.12:** Yes,  $I_{OH}$  of the 9S12 is 10 mA.

**Checkpoint 8.13:** No changes are necessary; just connect +12 V to the Vcc input of the TPIC0107B.

**Checkpoint 8.14:** There are three approaches to increasing torque. You can buy a larger stepper with more torque. You can increase the torque by using a reducing gear box. Your software may be exceeding the maximum jerk, so you can adjust the way you change speeds, as shown in Table 8.15.

**Checkpoint 8.15:** Feedback is important to know the exact position when the system starts, or whether there is a chance that a step command does not actually move the motor.

**Checkpoint 9.1:** 001X,XXXX,XXXX,XXXX. Setting all Xs to 0 gives \$2000; setting all Xs to 1 gives \$3FFF.

**Checkpoint 9.2:** \$D800 to \$FFFF.

**Checkpoint 9.3:** Change to

```
SYNR = 2; REFDV = 0; // PLLCLK=2*OSCCLK*(SYNR+1)/(REFDV+1)
```

**Checkpoint 9.4:**

$$t_{cyc} = t_1 = 100 \text{ ns}$$

$$\text{RDR} = \text{Read Data Required} = (t_1 - 15, t_1) = (85, 100)$$

$$\text{WDA} = \text{Write Data Available} = (\frac{1}{2}t_{cyc} + 7, t_1 + 2) = (57, 102)$$

$$\text{AA (narrow)} = (\frac{1}{2}t_{cyc} + [2, 6.5], t_1 + 2) = ([52, 56.5], 102)$$

$$\text{AA (wide)} = (\frac{1}{2}t_{cyc} + [2, 6.5], t_1 + \frac{1}{2}t_{cyc}) = ([52, 56.5], 150)$$

**Checkpoint 9.5:**

$$t_{cyc} = 40 \text{ ns} \text{ and } t_1 = 120 \text{ ns}$$

$$\text{RDR} = \text{Read Data Required} = (t_1 - 15, t_1) = (105, 120)$$

$$\text{WDA} = \text{Write Data Available} = (\frac{1}{2}t_{\text{cyc}} + 7, t_1 + 2) = (27, 122)$$

$$\text{AA (narrow)} = (\frac{1}{2}t_{\text{cyc}} + [2,6.5], t_1 + 2) = ([22,26.5], 122)$$

$$\text{AA (wide)} = (\frac{1}{2}t_{\text{cyc}} + [2,6.5], t_1 + \frac{1}{2}t_{\text{cyc}}) = ([22,26.5], 140)$$

**Checkpoint 9.6:**

$$\downarrow E = 250 \text{ ns}, 140.5 + t_{\text{acc}} \leq 250 - 15, \text{ so } t_{\text{acc}} \leq 94.5 \text{ ns}$$

**Checkpoint 9.7:** The end of WDA occurs at  $\downarrow E + 2$ . Let  $t_p$  be the propagation delay through external digital logic. To synchronize E1 or W, it would require external logic, so the end of WDR occurs at  $\downarrow E + t_p$ . If  $t_p > 2$  ns, then WDA would not overlap WDR.

**Checkpoint 9.8:**

$$\frac{1}{2}t_{\text{cyc}} = 62.5 \text{ ns}, \downarrow E = t_1 = (n + 1)*125, \text{ timing is limited by read cycle } \overline{EI} \text{ access}$$

$$\text{AA (narrow)} = (\frac{1}{2}t_{\text{cyc}} + [2,6.5], t_1 + 2) = ([64.5,69], t_1 + 2)$$

$$\downarrow \overline{EI} + [64.5,69] + [1.5,9] = [66,78]$$

$$\text{RDR} = \text{Read Data Required} = (\downarrow E - 15, \downarrow E)$$

$$78 + 150 \leq \downarrow E - 15, \text{ so } 243 \leq \downarrow E, \text{ so 1 stretch will be OK}$$

**Checkpoint 10.1:** Latency is the response time (time delay from device ready to device gets service), while bandwidth is the amount of information transferred per time.

**Checkpoint 10.2:** If we do not meet the latency requirement, those data are lost. If it happens every time, the system doesn't work. If it happens occasionally, it will run slow because we will have to wait for the disk to spin around one revolution and try it again.

**Checkpoint 10.3:** A portion of the sound is lost, and it will sound like a skip. We may also hear a click because the waveform is discontinuous.

**Checkpoint 10.4:** The system runs slow, because the transmitter will time out and try to resend the packets.

**Checkpoint 10.5:** The bidirectional driver has three possibilities, determined by two control pins. An example of this type of logic is the 74HC245. It can drive data left to right, making the left input and right output. It can drive data right to left, making the right input and left output. The third possibility is that the device can be off, driving neither the left nor the right. This is a noninverting driver, so the output equals the input.

**Checkpoint 10.6:** Substitute the four bidirectional data bus drivers with four unidirectional tristate drivers. All four data bus drivers operate in the direction of the simplex transfer (left to right). The bank-switched memory looks like a write-only memory to the computer and a read-only memory to the I/O hardware. An example of this simplex channel can be seen in Figure 10.18.

**Checkpoint 10.7:** The maximum latency for cycle steal DMA is one bus cycle; assume there is only one DMA channel active. If there are more than one DMA channel operating, one DMA request may have to wait for another.

**Checkpoint 10.8:** 842 ns is not enough time to output one 4-bit packet to the display. If you tried to control it from software, it will be much slower and the LCD screen will flicker.

**Checkpoint 10.9:** 10 Hz is a particularly disturbing frequency for humans. It is too fast to see the individual on/off cycles, but too slow to be seen as continuous.

**Checkpoint 10.10.** There can be up to 4 million blocks, so a 16-number would be too small. It could have been a 24-bit number if the software were written in assembly.

**Checkpoint 10.11.** The  $V_{IL}$  of the 7407 is 0.8 V. The  $V_{IH}$  of the 7407 is 2 V. When the input is low, the output is low. This works both directions. Going from +5 to +3.3, if the input is + 5 V the 7407 output floats and the resistor pull-up makes the output 3.3 V.

Going from +3.3 to +5, if the input is +3.3 V ( $3.3 > V_{IH}$ ) the 7407 output floats and the resistor pull-up makes the output 5 V.

**Checkpoint 10.12.** The largest contiguous part of the disk is 8 blocks. So the largest new file can have  $8 \times 512$  bytes of data (4096 bytes). This is less than the available 16 free blocks.

**Checkpoint 10.13.** First fit would put the file in block 1 (block 0 has the directory). Best fit would put the file in block 10, because it is the smallest free space that is big enough. Worst fit would put it in block 14.

**Checkpoint 10.14.** 2 Gibabytes is  $2^{31}$  bytes. 512 bytes is  $2^9$  bytes.  $31 - 9 = 22$ , so it would take 22 bits to store the block number.

**Checkpoint 10.15.** 2 Gibabytes is  $2^{31}$  bytes. 32k bytes is  $2^{15}$  bytes.  $31 - 15 = 16$ , so it would take 16 bits to store the block number.

**Checkpoint 10.16.** There are 16 free blocks; they can all be linked together to create one new file.

**Checkpoint 10.17.** You could store a byte count in the directory or in each block.

**Checkpoint 10.18.**  $16 + 9 = 25$ .  $2^{25}$  is 32 Mebibytes, which is the largest possible disk.

**Checkpoint 10.19.** There are  $2^{31}/2^{10} = 2^{21}$  blocks, so the 22-bit block address will be stored as a 32-bit number. One can store  $1024/4 = 256$  index entries in one 512-byte block. So the maximum file size is  $256 \times 512 = 2^8 \times 2^9 = 2^{17} = 128$  kibibytes. You can increase the block size or store the index in multiple blocks.

**Checkpoint 10.20.** There are 15 free blocks; they can create an index table using all the free blocks to create one new file.

**Checkpoint 10.21.** Each directory entry now requires 10 bytes. You could have 50 files, leaving some space for the free space management.

**Checkpoint 10.22.** There are 15 free blocks; they can create FAT using all the free blocks to create one new file.

**Checkpoint 11.1:** The open loop gain is 108 dB. In V/V, the gain is  $10^{108/20} = 10^{5.4} = 251,000$ .

**Checkpoint 11.2:** The output impedance is open-circuit voltage divided by short-circuit current, 5 V/30 mA, which is  $167 \Omega$ .

**Checkpoint 11.3:** The offset voltage is 0.5 mV. The error will be 0.5 mV times 100, which will be 50 mV.

**Checkpoint 11.4:** The offset voltage is 3 mV. The error is 3 mV times 1000, which would be 3 V.

**Checkpoint 11.5:** The gain-bandwidth product of the MAX494 is 500 kHz. The bandwidth of the circuit will be 500 kHz divided by the gain, which will be 5 kHz. The noise density of the MAX494 is 25 nV/ $\sqrt{\text{Hz}}$ . The noise will be  $25 \text{ nV} * \sqrt{5000}$ , which is about  $1.8 \mu\text{V}$ .

**Checkpoint 11.6:**  $10 \text{ k}\Omega$  in parallel with  $100 \text{ k}\Omega$  is about  $9.1 \text{ k}\Omega$ .

**Checkpoint 11.7:** The input impedance of the op amp determines the input impedance of a noninverting amplifier, which for the OPA227 is  $10 \text{ M}\Omega$ .

**Checkpoint 11.8:** Use rail-to-rail op amps, powered by +5 V and ground. Replace all connections to analog ground to a 2.50 V reference.

**Checkpoint 11.9:** Use an AD620 with  $R_G$  of  $4.94 \text{ k}\Omega$ .

**Checkpoint 11.10:**  $j$  is the square root of  $-1$ , a complex number.

**Checkpoint 11.11:** The average current must be less than  $(500 \text{ mA}\cdot\text{hr})/5y*(1y/356d)* (1d/24 \text{ hr}) = 11 \mu\text{A}$ .

**Checkpoint 11.12:**  $1023 * 1 \text{ V} / 5 \text{ V}$  is 205.

**Checkpoint 12.1:** If precision is given in alternatives, then range = precision\*resolution.

**Checkpoint 12.2:** Noise in the transducer, noise in the electronics, and the ADC resolution.

**Checkpoint 12.3:** A higher sensitivity means a lower amplifier gain, resulting in less noise at the output.

**Checkpoint 12.4:** The stability (low drift) and reproducibility of a transducer are major factors affecting accuracy. Accuracy combines resolution and calibration. If you calibrate a device, then the calibration parameters change and the instrument will measure incorrectly.

**Checkpoint 12.5:** There are 8 slots in the disk, and 60 seconds in a minute.  $7.5 = 60/8$ . The velocity resolution in rpm will be 7.5 times the frequency resolution of the input capture system in Hz.

**Checkpoint 12.6:** Pressure is force per area. So a force transducer can be directly used to measure pressure if we know the area over which the force is measured.

**Checkpoint 12.7:** Detection system will have thresholds. Adjusting these thresholds will trade false positives for false negatives.

**Checkpoint 12.8:**  $10/0.01$  is 1000, so 10 bits are needed.

**Checkpoint 12.9:** The resolution would decrease by a factor of 4 from about  $0.1^\circ\text{C}$  to  $0.025^\circ\text{C}$ .

**Checkpoint 12.10:** At 60 BPM, the `Rcount` will be 4 and the difference `Rlast-Rfirst` will be 480. If the heart rate increases a little, then the difference will reduce to 479, giving a measurement of 60.13. Thus, the inherent measurement resolution is about 0.12 BPM. Because integer math is used, the system will have a resolution of 1 BPM.

**Checkpoint 12.11:** The resolution would decrease by a factor of 4 from about  $0.1^\circ\text{C}$  to  $0.025^\circ\text{C}$ .

**Checkpoint 13.1:** Assume the yaw motor position is fixed. The location of the pitch motor is fixed relative to the yaw motor and has no translational motion. The location of the roll motor is affected by length of the pitch shaft and rotating the yaw motor. The translational motion on the tip caused by rotating the pitch motor depends on the length of the roll shaft, the pitch angle, and the yaw angle.

**Checkpoint 13.2:** The greatest common factor of 75 ms and 500 ms is 25 ms. A single ISR interrupting every 25 ms could have been used to solve this system. The `YawTime` `PitchTime` `RollTime` parameters would be multiplied by 3, and the `Duration` values multiplied by 20.

**Checkpoint 13.3:** There are no critical sections, because the global is accessed within an ISR. The ISR is atomic, because interrupts are disabled.

**Checkpoint 13.4:** If they are too close, then the system can turn on-off-on-off- . . . very quickly, causing the electromagnetic relays to prematurely fail.

**Checkpoint 13.5:** If they are too far apart, then the system will oscillate with large positive and negative errors.

**Checkpoint 13.6:** At every interrupt, the actuator would be increased or decreased, causing a lot of output changes.

**Checkpoint 13.7:** If it were too fast, the actuator would be increased to maximum or decreased to minimum, causing it to behave like a bang-bang controller.

**Checkpoint 13.8:** The output will saturate. The error increases to a very large positive value or decreases down to a very large negative value.

**Checkpoint 13.9:** The limit of the discrete integral as  $\Delta t$  goes to zero is the continuous integral.

**Checkpoint 13.10:** The limit of the discrete derivative as  $\Delta t$  goes to zero is the continuous derivative.

**Checkpoint 13.11:** Yes. Let **watts** be the units of the actuator output and **cm** be the units of the sensor input. The units of the lag **L** is **sec**. The units of the rate **R** is **cm/sec**. The units of  $\Delta U$  is **watts**.

$$\text{Proportional} \quad K_P = 1.2 \frac{\Delta U}{(L * R)} \quad \text{watts/(sec*(cm/sec))} = \text{watts/cm}$$

$$\text{Integral} \quad K_I = 0.5 K_P / L \quad \text{watts/(cm-sec)}$$

$$\text{Derivative} \quad K_D = 0.5 K_P L \quad (\text{watts-sec})/\text{cm}$$

**Checkpoint 13.12:**  $E = X^* - X$ , so the error is very negative, causing the P term to be very negative, making  $U = 100$ . This removes power, and gravity will force it down.

**Checkpoint 13.13:**

$$\text{SlowDown} = \text{WayTooFast} + \text{SpeedingUp} * \text{LittleBitFast} = 50 + (40 * 60) = 50$$

**Checkpoint 14.1:** Since information is encoded as energy, and data are transferred at a fixed rate, each energy packet will exist for a finite time. Energy per time is power.

**Checkpoint 14.2:** The performance measure for a storage system is information density in bits/cm<sup>3</sup>.

**Checkpoint 14.3:** With open collector outputs, the low will dominate over HiZ. The signal will be low.

**Checkpoint 14.4:** On average, it will take  $N/2$  transmissions for the message to go from one computer to another. There are 10 bits/frame, so there are 10,000 bytes/sec. Because there are 10 bytes/message, it takes 1 ms to transmit a message. Because it has to be sent 5 times, it takes 5 ms on average.

**Checkpoint 14.5:** The frame sent by a transmitter is echoed to its own receiver. If the data do not match, or if there are any framing or noise errors, then a collision occurred.

**Checkpoint 14.6:** Parity could be used to detect collisions. Also the message could have checksum added. Framing or noise errors can also indicate a collision.

**Checkpoint 14.7:** Open collector has two output states: low and HiZ. The low state will dominate over HiZ. HiZ is the recessive state.

**Checkpoint 14.8:**  $z + 2$  is equal to  $y + 1$ . So, make  $y$  equal to  $z + 1$ .

**Checkpoint 15.1:**  $4,000,000/1000$  is 4000.

**Checkpoint 15.2:**

For DC,  $z = 1$

$$H(z) = \frac{1 + z^{-1}}{2} = 1$$

For 250,  $z = j$

$$H(z) = \frac{1-j}{2} = 0.707$$

For 500,  $z = -1$

$$H(z) = \frac{1-1}{2} = 0$$

**Checkpoint 15.3:** Using Equation 22, at 60 Hz,  $f/f_s$  is 1/6.

$$\begin{aligned} \text{Gain} &= 1/6[\{1 + \cos(2\pi/6) + \cos(4\pi/6) + \cos(6\pi/6) + \cos(8\pi/6) + \cos(10\pi/6)\}^2 \\ &\quad + \{\sin(2\pi/6) + \sin(4\pi/6) + \sin(6\pi/6) + \sin(8\pi/6) + \sin(10\pi/6)\}^2] \end{aligned}$$

The gain is zero using the fact that  $\cos(x + \pi) = -\cos(x)$  and  $\sin(x + \pi) = -\sin(x)$ .

**Checkpoint 15.4:** Using Equation 27, at 60 Hz,  $f/f_s$  is 1/6.

$$\text{Gain} = 0.5 \sqrt{\{1 + \cos(6\pi/6)\}^2 + \{\sin(6\pi/6)\}^2} = \sqrt{\{1 - 1\}^2 + 0^2} = 0$$

**Checkpoint 15.5:** If the gain is larger than one, amplification occurs. For example, if the gain is 1.2, if you put in a sinusoidal wave with amplitude 100, then the output of the filter will be a sinusoidal wave with amplitude 120. This is important because a filtered signal from an 8-bit ADC will not fit into an 8-bit variable.

**Checkpoint 15.6:** In this case it is unsigned short (16-bit), so the numerator must be less than 65535.

**Checkpoint 15.7:** In this case the math is signed 32-bit, so the numerator must be less than 2147483647.

**Checkpoint 15.8:** Some of the minor causes of time-jitter are the variability in which instruction is being executed when the interrupt request is issued, and the variability in software execution paths caused by conditional branching. The largest source of time-jitter occurs when the software runs with interrupts disabled, which can occur either while executing other ISRs or during the foreground when the software temporarily disables interrupts to perform tasks in a critical section.

**Checkpoint 15.9.** The input is bounded from 0 to 1023 because the data comes from the 10-bit ADC. The largest gain in this filter is 2, the fixed-point coefficient is 1024.  $1023*2*1024$  will fit in the 32-bit signed intermediate result, sum. The full-scale input is  $\pm 512$ . So, at maximum gain of 2, the output result in Reg D will be between  $-1024$  and  $+ 1024$ .

**Checkpoint 15.10.** Because of the linear phase, the  $h(n)$  filter coefficients are symmetric. Notice that  $h(k)$  equals  $h(50-k)$ . For example,  $4 \cdot x(n) + 4 \cdot x(n-50)$  can be replaced with  $4 \cdot (x(n) + x(n-50))$ . In general,  $h(k) \cdot x(nk) + h(50-k) \cdot x(n-50-k)$  can be replaced with  $h(k) \cdot (x(n-k) + x(n-50-k))$ , saving 25 multiplies.

# Index

- #define instruction, 128
- 16-bit numbers, 27–28
- 16-bit timer (TCNT), 97–98
- 8-bit numbers, 25–26
- HD44780 controller, LCD interfacing, 178–180
- MC145000 driver, LCD interfacing, 424–421
- MC14543 driver, LCD interfacing, 423–424
- MC9S12C32 microcontroller, 16, 37–39, 222, 318–322, 360–363, 461, 472–479
- address decoding, 461
- architecture, 37–39
- digital logic of, 16
- external bus timing, 472–479
- key wakeup interrupts, 222
- memory interfacing, 461, 472–479
- pseudo-operation code (pseudo-op), 37–39
- pulse-width modulation (PWM), 318–322
- serial peripheral interface (SPI) details, 360–363
- 9S12 microprocessor, 3–7, 26, 29–46, 65–71, 76–77, 97–98, 107–110, 136, 160–162, 175–176, 212–216, 235–240, 288–292, 316–318, 350–357, 363–368, 373–377, 460–461, 480–496, 582–586, 695–700
- 16-bit timer (TCNT), 97–98
- addresses, 29–32
- addressing modes, 33–36, 68–71
- analog-to-digital converter (ADC) details, 582–586
- architecture, 30–36
- arithmetic extended instructions, 76–77
- assembly language instructions, 32–33
- assembly language programming, 65–71
- background debug modules (BDM), 136
- bus cycles, 3–4
- controller area network (CAN), 695–700
- device driver, CAN, 698–700
- expanded (wide/narrow) interfacing modes, 460, 480–489
- flash EEPROM programming, 492–496
- full-duplex serial port, 352–355
- history of, 3–31
- I/O mapped I/O, 5–6
- indexed addressing modes, 68–71
- input capture, 288–292
- interfacing, 36, 175–176
- inter-integrated circuit ( $I^2C$ ) interface, 373–377
- interrupt interfacing features, 212–216, 235–240
- key wake-up, 160–162
- low power design, 239–240
- memory interfacing, 460–461, 480–496
- memory operations, 66–68
- microcontrollers, 36–45
- module routing register, 363–368
- numbering scheme, 36
- numerical operation instructions, 26
- operands, 65–66
- operating modes, 45–46
- paged memory, 489–492
- processor, 5–7
- pulse accumulator, 316–318
- real-time periodic interrupts (RTI), 235–239
- receiver, CAN, 695–697
- RS232 SCI interfaces, 350–357
- serial communication interface (SCI) details, 36, 175–176, 350–357
- serial peripheral interface (SPI) details, 363–368
- signals, command and timing, 460–461
- simplex printer interface, 355–357
- stack implementation, 107–110
- timer details, 97–98, 288–292, 316–318
- 9S12DP512 microcontroller, 39–44, 222, 224
  - architecture, 39–44
  - key wakeup interrupts, 222, 224
- 9S12E128 microcontroller architecture, 44–45
- RS422 system, 331–332, 339–340, 342–343
  - protocol, 331–332
  - balanced differential lines, 339–340, 342–343
- RS485 system balanced differential lines, 343–344
- RS423 system balanced differential lines, 340
- RS232 system, 331–332, 337–341, 350–357
  - 9S12 SCI interfacing, 350–357
  - non-return-to-zero ((NRZ) encoding, 350
  - protocol, 331–332
  - serial I/O device specifications, 337–339
- A**
  - Absolute bandwidth, 701
  - Abstraction, 96–104
    - 9S12 timer (TCNT) details, 97–98
    - finite-state machines (FSM), 96–104
    - linked structure, 96
    - proportional integral derivative (PID)
      - digital controllers, 96
    - software, 96
    - timers, 97–104
  - Access, stack implementation, 109
  - Accumulator offset addressing modes, 70
  - Accumulators, 6, 71, 316–318
    - pulse, 316–318
    - register functions, 6, 71
  - Accuracy, 551, 593–595
    - data acquisition systems (DAS), 593–595
    - digital-to-analog converters (DAC), 551
      - instrument, 593–595
  - Acquisition time, 570
  - Active state, thread, 253
  - Actuator interface, 13–14
  - Actuators, 648, 662, 669
  - ADC interface, 163
  - ADC software, 13–14
  - adda and adddb instructions, 72
  - Address bus, 3
  - Address decoders, 460–467
    - full-address, 461–463
    - functions, 460
    - Karnaugh maps for, 463–465
    - MC9S12C32 details, 461
    - memory interfacing, 460–467
    - minimal-cost, 463–465
    - special cases for, 466–467
  - Addressing modes, 33–36, 68–71
    - 9S12 instructions, 68–71
    - accumulator offset, 70
    - auto pre/post decrement/increment, 69
    - direct, 33–34
    - extended, 33, 35
    - immediate, 33–34
    - indexed, 68–71
    - indirect, 70
    - inherent, 33–34
    - offset, 70–71
    - PC-relative, 33–36
  - Aging, scheduling, 255
  - Aliasing error, 612–614
  - Allocation, 109, 519–522
  - Alternative number values, 23
  - American Standard Code for Information Interchange (ASCII) code, 26–27
  - Amplifiers, 534–548, 553–554, 611
    - analog interfacing, 534–548, 553–554
    - data acquisition systems (DAS), 611
    - instrumentation, 544–545
    - inverting, 539–541
    - noninverting, 541–544
    - operational (op amps), 534–548
    - summing, 553–554
  - Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

- Analog filters, *see* Filters  
 Analog interfacing, 3, 531–590  
   analog to digital converters (ADC), 564–570,  
 573–575, 582–586  
   BiFET multiplexer, 571–573  
   capacitors, 532–534  
   defined, 3  
   digital-to-analog converters (DAC), 551–564  
   filters, 548–551  
   low power design, 576–577  
   multiple-access circular queue (MACQ), 580–582  
   operational amplifiers (op amps), 534–548  
   power, 558–559, 575–580  
   resistors, 531–532  
   sample and hold (S/H), 570–571  
 Analog isolation, 547–548  
 Analog signal processing, 592, 615–620  
 Analog-to-digital converters (ADC), 10, 23, 564–570,  
   573–575, 582–586, 615–620  
   9S12 system details, 582–586  
   analog interfacing, 564–570, 573–575, 582–586  
   block diagrams, 573–575  
   CMOS input protection, 575  
   conversion time, 620  
   DAS design and, 615–620  
   defined, 10  
   dual slope, 567–570  
   flash, 565–566  
   internal, 582–586  
   linear, 565  
   microcomputer features, 10  
   monotonic, 565  
   number generation, 23  
   parameters, 564–565  
   power requirements, 575  
   precision, 615  
   signal processing, 615–620  
   sixteen-bit, 567–570  
   software, 585–586  
   successive approximation, 566–567  
   system, 573–580  
   two-bit, 565–566  
 and function, 84–85  
 Aperiodic thread, 253  
 Aperture time, 570–571  
 Application program interface (API), 10, 122,  
   703–704. *See also* Device drivers  
 Architecture, 2–7, 30–46  
   9S12 microprocessor, 30–46  
   9S12C32 microcontroller, 37–39  
   9S12DP512 microcontroller, 39–44  
   computer systems, 2–7  
 Arguments, 114–115  
 Arithmetic logic unit (ALU), 6  
 Arithmetic operations, 71–77  
   condition code register (CC, CCR) for, 71–75  
   9S12 extended instructions, 76–77  
 Arithmetic shift left (ASL), 78  
 Arithmetic shift right (ASR), 78  
 Arm, defined, 288  
 Assembler, 62–64  
 Assembly directive, *see* Pseudo-operation code  
   (pseudo-ops)  
 Assembly language, 32–33, 62–92, 119–121  
   9S12 instructions, 32–33, 76–77  
   arithmetic operations, 71–77  
   assembler, 62–64  
   branch operations, 83–85  
   CodeWarrior assembler, 62, 64  
   comment field, 33  
   editor, 62  
   indexed addressing modes, 68–71  
   label field, 32, 64–65  
   listing file, 63  
   loader, 62  
   logical operations, 79–80  
   memory allocation, 89–92  
   memory operations, 66–68  
   object code, 62–64  
   operand field, 33, 65–66  
 operation (opcode) field, 32–33, 65  
 programming, 62–92, 119–121  
 pseudo-ops, 65, 85–89  
 register transfer operations, 66–68  
 shift operations, 77–79  
 source code, 62–64  
 stack, 80–83  
 subroutines, 80–83  
 symbol table, 63–64  
 syntax, 64–66  
 TExaS assembler, 62–64  
 Asynchronous bus timing, 469–471  
 Asynchronous communication systems,  
   173–175, 336, 346–350  
   receiving data, 174–175, 347–350  
   SCI interfacing, 173–175, 346–350  
   serial I/O devices, 336  
   transmitting data, 173–174, 346–347  
 Asynchronous communications interface  
   adapter (ACIA), 330  
 Atomic operations, 201–202  
 Attenuation, networks, 688  
 Auto pre/post decrement/increment addressing modes, 69

**B**

- Background debug modules (BDM), 136  
 Background thread, 131, 192  
 Balanced differential line protocols, 339–344  
 Band-reject filters, 550–551  
 Bandpass filters, 550–551  
 Bandwidth, 4, 151, 255, 333–334, 536, 701  
 Bandwidth coupling, 118  
 Bang bang, nonlinear transducers, 601  
 Bang-bang controller, 655–657  
 Bank-switched memory, 505–506  
 Banked memory model, 490  
 Base pointer (BP), 90  
 Basis, 25  
 Baud, defined, 708  
 Baud rate, 333, 339–340, 708  
 Bias current, 535–536  
 Biendian storage approach, 28  
 BiFET analog multiplexer, 571–573  
 Big endian storage approach, 28  
 Binary numbers, 22–223  
 Binary recursive function, 132  
 Binary semaphore, 193  
 Binding, stack implementation, 108  
 Bit rate, 333  
 Bit stuffing, 694  
 Bit time, 332  
 Blind cycle counting synchronization, 153, 162–163  
 Block diagrams, 573–575  
 Blocked state, thread, 253  
 Blocking semaphore implementation, 267–270  
 Board-support package (BSP), 121, 123, 277–278  
 Bottom-up design, 15, 116–117  
 Bounded waiting, 267  
 Branch operations, 83–85  
 Breakdown, 601  
 Breakdown utilization (BU), 256  
 Breakpoint, 138  
 Brushed DC motor interfacing, 433  
 Brushless DC (BLDC) motor interfacing, 434, 440–441  
 bsz instruction, 81–82, 490  
 Buffered input, 198  
 Buffered output, 199  
 Built-in timers, 98–104  
 Burst mode DMA, 507–508  
 Bus, defined, 3  
 Bus arbitration, 158, 693–694  
 Bus cycle, 3–4, 7  
 Bus interface unit (BIU), 5–6  
 Bus timing, 460–461, 467–479  
   asynchronous, 469–471  
   external, 471–479  
   fully asynchronous, 469–471  
   memory interfacing, 460–461, 467–479  
   partially asynchronous, 469  
   synchronous, 460, 468

- Busy-to-done state, 151–152  
 Busy-wait synchronization, 153, 163–166, 176–178  
 Butterworth filters, 549
- C**  
 C code, 106–107, 110–111, 126–129, 294–295, 306–307, 311–312  
     assembly language functions, 106–107, 110–111, 126–129  
     const modifier, 106  
     encapsulated objects using, 126–127  
     frequency measurement, 311–312  
     object-oriented interfacing using, 126–129  
     output compare, 306–307  
     period measurement, 294–295  
     portability using, 128–129  
     static modifier, 106  
     timer implementation, 294–295, 306–307, 311–312  
     variable creation, 106–107  
 C++ code, object-oriented interfacing using, 127–129  
 Call graphs, 13–14, 115–116  
 call instruction, 490–491  
 Capacitors, 392–395, 532–534  
     analog interfacing uses, 532–534  
     hardware debouncing using, 392–395  
 Carrier frequency, 706  
 Causal filters, 717  
 Certification, RTOS, 257  
 Channels, 330–331, 343–345, 558, 688, 701  
     capacity, 688, 701  
     Channels, digital-to-analog converters (DAC), 558  
     current loop, 344  
     digital logic, 345  
     networks, 688, 701  
     optical, 345  
     RS485 half-duplex, 343–344  
     serial, 330–331  
     serial I/O devices, 330–331, 343–345  
     Shannon-Hartley channel capacity theorem, 701  
 Characters, number representation of, 36–27  
 Checksum, 691  
 Circuits, 368–377, 537–546  
     analog interfacing, 537–546  
     current-to-voltage, 545  
     derivative, 546  
     integrator, 545–546  
     inter-integrated circuit ( $I^2C$ ) interface, 368–377  
     linear, 537–544  
     operational amplifiers (op amps), 537–546  
     voltage-to-current, 545  
 Clock, DAS design, 612  
 Closed-loop control systems, 648–649, 655–659  
     bang-bang controller, 655657  
     incremental controller, 657–659  
     state variables (real and desired), 648–649  
 clra instruction, 33  
 CMOS analog input protection, 575  
 Code segment selector (CS), 89  
 CodeWarrior assembler, 62, 64  
 Cohesion, 118–119  
 Coincidental cohesion, 119  
 Commands, memory interfacing, 459–461  
 Comment field, 33  
 Comments, 92–94  
 Common-mode input impedance, 535  
 Common-mode rejection ratio (CMRR), 538–539  
 Communication, 270–272. *See also Networks; Synchronization*  
 Communicational cohesion, 119  
 Complementary metal oxide semiconductor (CMOS) devices, 16–17, 19–21  
 Compression ratio, 709  
 Computer architecture, 2–7  
     input/output, 2–5  
     processor, 2, 5–7  
     random access memory (RAM), 2–5  
     read only memory (ROM), 2  
 Computer-controlled devices, 427–443  
     current switches, 427–428  
     parallel port interfacing, 427–443  
     relays, 429–432  
 solenoids, 432–442  
 solid-state relays (SSR), 442–443  
 Computer system, defined, 2  
 Concurrent programming, 131–132  
 Condition code bits, 32  
 Condition code register (CC, CCR), 6, 71–75  
 Conditional execution, 84–85  
 Configuration, digital-to-analog converters (DAC), 558  
 const modifier, 106  
 Control bus, 3, 5  
 Control coupling, 118  
 Control field, 694  
 Control systems, 648–685  
     actuators, 648, 662, 669  
     bang-bang controller, 655657  
     closed-loop, 648–649, 655–659  
     defined, 648  
     fuzzy logic, 668–681  
     incremental, 657–659  
     microcomputer-based, 648–685  
     open-loop, 649–655  
     physical plant, 648, 669  
     proportional integral derivative (PID) controllers, 659–668  
     state variables (real and desired), 648–649  
 Control unit (CU), 6  
 Controller area network (CAN), 692–700  
     9S12 details, 695–700  
     bit stuffing, 694  
     bus arbitration, 693–694  
     control field, 694  
     cyclic redundancy check (CRC) field, 694  
     data field, 694  
     device driver, 698–700  
     frames, 694  
     intermission frame space (IFS), 694  
     messages, 694–695  
     receiver, 695–698  
     use of, 692–693  
 Controller errors, 649  
 Controller software, 13–14, 51  
 Conversion time, ADC, 620  
 Convolution, FIR filters, 737  
 Cooperative multitasking, 273  
 Cooperative scheduler, 254  
 Count instruction, 295–298  
 Counter instruction, 292, 310  
 Coupling, 115, 118  
 CPU-bound system, 197  
 Critical sections, 199–205  
 Crosstalk, 688  
 Current, op amp parameters, 535–536  
 Current-controlled devices, interfacing, 435  
 Current loop channel, 344  
 Current switches, 427–428  
 Current-to-voltage circuit, 545  
 Cutoff frequency, 548  
 Cycle steal DMA, 507–508  
 Cycle stretching, 469  
 Cycles, 458–459, 467, 506–507  
     burst mode DMA, 507–508  
     cycle steal DMA, 507–508  
     direct memory access (DMA), 507–510  
     dual-address DMA, 508–510  
     high-speed I/O interfacing, 506–507  
     memory interfacing, 458–459, 467  
     read data, 458–459, 467, 506–507  
     single-address DMA, 508–510  
     write data, 458–459, 467, 506–507  
 Cyclic redundancy check (CRC) field, 694
- D**  
 Data acquisition systems (DAS), 502, 591–647  
     accuracy, 593–595  
     components, 592–593  
     design, 611–621, 629–639  
     electrical activity measurement, 633–636  
     high-speed I/O interfacing, 502  
     mass balance, 597  
     microphones, 603, 637–639

- Data acquisition systems (*continued*)  
 noise analysis, 621–628  
 parameters, 593–596  
 position measurement, 636–637  
 potentiometers, 636–637  
 precision, 595–596  
 reproducibility, 596  
 resolution, 595  
 sound input/output, 637–639  
 strain gauges, 605–606, 597  
 temperature measurement, 606–610, 629–633  
 thermistors, 597, 607–608, 629–633  
 thermocouples, 597, 608–610  
 transducers, 596–610
- Data bus, 3
- Data communication equipment (DCE), 331
- Data conversion element, DAS, 593
- Data field, 694
- Data flow graphs, 12–13
- Data intervals (available/required), 157–158
- Data segment selector (DS), 90
- Data structures, 13
- Data terminal equipment (DTE), 331
- Dead zone, 601
- Deadlines, scheduling, 255–256
- Deallocation, stack implementation, 109–110
- Debouncing, 392–401  
 hardware, 392–395  
 software, 395–401
- Debugging, 134–144  
 breakpoint, 138  
 functional, 138–140  
 intrusiveness, 137  
 minimally intrusive, 137  
 nonintrusive, 137  
 performance, 140–143  
 profiling, 143–144  
 software tools for, 135–136  
 theory, 136–138
- Define 32-bit constant (d1 directive), 88
- Defuzzification, 670–671, 676
- Delay software, 98–104
- Delayed pulse generation, 309–310
- Derivative circuit, 546
- Design, 11–15, 50–55, 60–149, 217–219, 239–240, 576–577, 611–621, 629–639, 662–668, 736–739
- ADC precision, 615
- analog interfacing, 576–577
- analog signal processing, 615–620
- bottom-up, 15, 116–117
- call-graphs, 13–14
- constraints, 11–12
- data acquisition systems (DAS), 611–621, 629–639
- data flow graphs, 12–13
- embedded system parallel I/O ports, 50–55
- external interrupt approach, 217–219
- finite-impulse response (FIR) filters, 736–739
- flowcharts for, 52–53
- interrupt interfacing, 217–219, 239–240
- low power, 239–240, 576–577
- microcomputer-based systems, 11–15, 50–55
- negative predictive value (NPV), 611
- Nyquist theory for, 612–615
- positive predictive value (PPV), 611
- prevalence, 611
- printed circuit boards (PCB), 54
- process, 11–15
- proportional integral derivative (PID) controllers, 662–668
- prototypes, 53–55
- pseudocode for, 52–53
- requirements, 11–12
- sample and hold (S/H), 620–621
- sampling rate, 612–615, 736
- SCI device driver, 124–125
- sensitivity, 611
- simulation, 53–54
- software systems, 60–149
- specifications, 11
- specificity, 611
- successive refinement, 52
- top-down, 11–15, 116
- Deterministic operating system, 256
- Device drivers, 10, 123–125, 173–178, 698–700  
 9S12 details, 175–176, 698–700  
 asynchronous receiving, 174–175  
 asynchronous transmission, 173–174  
 controller area network (CAN), 698–700  
 defined, 10  
 high-level, 125  
 linking, 124  
 low-level, 123–124  
 policies of, 123–124  
 serial communications interface (SCI), 124–125, 173–178  
 synchronization software, 176–178
- Device latency, 500
- Differential input impedance, 535
- Digital filters, 612, 716–743  
 applications, 719–726  
 causal, 717  
 DAS design, 612  
 design, 736–739  
 direct form, 728–729, 739–740  
 discrete Fourier transform (DFT), 716, 736  
 finite-impulse response (FIR), 717–718, 726–729, 736–739  
 high-Q 60-Hz notch, 729–733  
 impulse response (IR), 726–729  
 infinite-impulse response (IIR), 718, 726–729  
 Laplace transform analysis, 718  
 linear, 717–719  
 median, 726–727  
 nonlinear, 717  
 response (gain and phase) examples, 718–726  
 sampling rate process, 717–718, 736–737  
 time jitter effects, 734–735  
 Z transform analysis, 718–719
- Digital logic, 16–22  
 complementary metal oxide semiconductor (CMOS)  
 devices, 16–17, 19–21  
 fan out, 16  
 open collector logic, 20–22  
 slew rate, 16  
 switch interfaces, 22  
 transistor-transistor logic, 16–18  
 transition time, 16  
 tristate logic, 18  
 voltage thresholds, 17–18
- Digital logic channel, serial I/O devices, 345
- Digital signal processing (DSP), 593, 716–743.  
*See also* Digital filters
- Digital-to-analog converters (DAC), 23, 551–564  
 analog interfacing, 551–564  
 number generation, 23  
 parameters, 551–553  
 power requirements, 558–559  
 pulse-width modulation (PWM), 563–564  
 R-2R ladder used for, 554–556  
 selection of, 557–560  
 summing amplifier used for, 553–554  
 three-bit, 552–556  
 twelve-bit, 556–557  
 waveform generation, 560–563
- Digital voltmeter (DVM), 626
- Direct addressing mode, 33
- Direct form, 728–729, 739–740
- Direct memory access (DMA), 4–5, 154, 506–511  
 burst mode, 507–508  
 computer system use of, 4–5  
 cycle steal, 507–508  
 dual-address cycle, 508–510  
 high-speed I/O interfacing, 506–511  
 initiation, 507  
 interfacing approach, 154  
 programming, 510–511  
 read cycle data, 506–507  
 single-address cycle, 508–510  
 write cycle data, 506–507
- Direct-page addressing mode, 34
- Direction register, 49–50

- Directives, 65  
 Directory, file system management, 521–522  
 Discrete Fourier transform (DFT), 716, 736  
 Distortion, networks, 688  
 dl directive, 88  
 dl .1 directive, 89  
 Don't care state, 463–464  
 Double buffer scheme, 511  
 do-while operation, 84–85  
 Drift, 599  
 Dropout rate, 571  
 Drop-out error, 29  
 ds .w directive, 89  
 Dual-address DMA cycle, 508–510  
 Dual port memory, 505  
 Dual slope ADC, 567–570  
 Dynamic efficiency, 61, 141–142  
 Dynamic operating system, 256  
 Dynamic RAM (DRAM), 496–497  
 Dynamic transducers, 600
- E**  
 Edge-triggered interrupt requests, 191  
 Editor, 62  
 ediv instruction, 76–77  
 Effective address register (EAR), 6  
 Electret condenser microphone (ECM), 603, 637  
 Electrical activity measurement, 633–636  
 Electrically erasable PROM (EEPROM), 2, 91–92,  
     492–496  
     assembly language programming design, 91–92  
     flash programming, 492–496  
     memory interfacing, 492–496  
 Electromagnetic field induction, 625  
 Electromagnetic interference (EMI), 434–435  
 Electromagnetic (EM) relay interfaces, 430–432  
 emacs instruction, 77  
 Embedded computer systems, 7–10  
     analog to digital converters (ADC), 10  
     application programmer interface (API), 10  
     applications of, 7–9  
     characteristics of, 7–8  
     device driver, 10  
     general-purpose computers compared to, 9–10  
     input/output interfaces, 10  
     microcomputer, 7  
     serial communications interface (SCI), 10  
     serial peripheral interface (SPI), 10  
     software maintenance and, 7  
 emul instruction, 26, 76  
 Encapsulation, 126  
 Encoding, 708–709  
 End condition, 132  
 Entry point, 112–114  
 equ directive, 37–39, 86  
 Equate symbols to values (equ directive), 86  
 Errors, 29, 175, 612–615, 624–625, 649, 689–692  
     aliasing, 612–615  
     controller, 649  
     drop-out, 29  
     motion, noise analysis, 624–625  
     network detection, 689–692  
     number representation, 29  
     overflow, 29  
     overrun (OR), 175  
     SCI transmission checking, 691–692  
 Expanded (wide/narrow) modes, 46, 460, 480–489  
 Exponential queue, 255  
 Extended addressing mode, 33, 35  
 External bus timing, 471–479  
 External interrupt design approach, 217–219
- F**  
 Factorial, 132–133  
 Fan-in/fan-out module behavior, 16, 119  
 Fast Fourier transform (FFT), 736  
 fcb directive, 87–88  
 fcc directive, 87–88  
 fdb directive, 88  
 fdiv instruction, 74
- Fiber-optic light channel, 345  
 Field (controller) tuning, 662  
 File allocation table (FAT), 523–524  
 File system management, 519–524  
     allocation, 519–522  
     directory, 521–522  
     file allocation table (FAT), 523–524  
     fragmentation (internal/external), 524  
     free-space management, 522–523  
     high-speed I/O interfacing, 519–524  
     linked allocation, 522  
     logical-to-physical address translation, 522  
 Filters, 548–551, 611–612, 615–616, 716–743.  
     See also Digital filters  
     analog, 611, 615–616  
     analog interfacing, 548–551  
     bandpass, 550–551  
     band-reject, 550–551  
     Butterworth, 549  
     DAS design, 611–612, 615–616  
     digital, 612, 716–743  
     low-pass, 615–616  
     one-pole low-pass, 548–549  
     response (gain and phase) examples, 718–726  
 Finite-impulse response (FIR) filters, 717–718,  
     726–729, 736–739  
     convolution of, 737  
     design, 736–739  
     impulse response (IR), 726–729  
     sampling rate process, 717–178, 736–737
- Finite-state machines (FSM), 96–104  
     abstraction use of, 96–97  
     Mealy, 96–97, 101–104  
     Moore, 96–101
- First-in-first out (FIFO), 91, 118, 174, 195,  
     206–212, 271, 504–505  
     defined, 91  
     dynamics, 211–212  
     hardware, 504–505  
     high-speed I/O interfacing, 504–505  
     information hiding, 118  
     interfacing and, 174  
     interrupt synchronization, 195, 206–212  
     queues, 195, 206–212  
     real-time operating systems (RTS), 271  
     thread synchronization, 195, 271  
     two-pointer implementation, 207–210  
     two-pointer/counter implementation, 210–211  
     use of, 206–207
- Fixed-point numbers, 28–30  
 Fixed scheduler, 272–277  
 Flag actions, 193–194, 213  
 Flag register, 6  
 Flash ADC, 565–566  
 Flash EEPROM programming, 492–496  
 Flip flops, 48  
 Flowcharts, 52–53  
 Force transducers, 605–606  
 Foreground thread, 131–132, 192  
 Fork, threads, 131  
 Form constant byte (fcb directive), 87–88  
 Form constant character string (fcc directive), 87–88  
 Form double byte (fdb directive), 88  
 Fractional power containment bandwidth, 701  
 Fragmentation (internal/external), file system  
     management, 524
- Frame pointer, 110  
 Frames, 332, 694  
 Free-space management, 522–523  
 Freescale numbering schemes, 36  
 Frequency measurement, 310–317  
     conversion between period and, 312–316  
     Counter instruction, 310  
     pulse accumulators, 317
- Frequency response, 600  
 Frequency spectrum, 617  
 FSK modem, 705–708  
 Full-address decoding, 461–463  
 Full-duplex communication system, 334, 687  
 Functional cohesion, 119

- Functional debugging, 138–140  
 Fuzzification, 670  
 Fuzzy logic control, 668–681
- G**  
 Gadfly synchronization, 153, 163–166  
 Galvanic noise, 624  
 Gate, 122  
 General-purpose computers, 9–10  
 Global variables, 91–92, 105–106  
     *const* modifier, 106  
     memory allocation, 91–92  
     modular software development, 105–106  
     *static* modifier, 106  
 Graphical processing unit (GPU), 514–515  
 Graphics controller, LCD high-speed I/O interfacing, 511–515  
 Guaranteed ratio (GR), 256  
 Guided medium, 688
- H**  
 Half-duplex communication system, 334–335, 343–344, 687  
 Half-power bandwidth, 701  
 Hard real time, 252  
 Hardware abstraction layer (HAL), 121, 123  
 Hardware interfaces, 449–451  
 Hexadecimal numbers, 24  
 High instruction, 79–80  
 High-Q 60–Hz notch filters, 729–733  
 High-speed I/O interfacing, 500–530  
     applications of, 501–504  
     bank-switched memory for, 505–506  
     data acquisition, 502  
     direct memory access (DMA), 506–511  
     dual port memory for, 505  
     file system management, 519–524  
     first-in-first-out (FIFO) hardware for, 504–505  
     latency, 500–501  
     liquid crystal display (LCD) graphics controller, 511–515  
     mass storage, 501–502  
     need for, 500–501  
     network communications, 503–504  
     secure digital card (SDC), 515–518  
     seek time, 501–502  
     signal generation, 503  
     video displays, 503  
 Hold time, 158  
 Hook, RTOS, 257  
 Hysteresis, 601, 546–547
- I**  
*idiv* instruction, 74–75  
 Idle state, 151  
*if* operations, 84–85  
 Immediate addressing mode, 34  
 Impedance, 535–536, 598–599, 617–620  
     loading, DAS design, 617–620  
     op amp parameters, 535–536  
     transducers, 598–599  
 Impulse response (IR), 726–729  
*in* instruction, 5  
 Incremental controller, 657–659  
 Index registers, 6  
 Indexed addressing modes, 68–71  
 Indirect addressing modes, 70  
 Infinite-impulse response (IIR) filters, 718, 726–729  
 Information hiding, 118  
 Inherent addressing mode, 33–34  
 Inheritance, priority, 267  
*Init ()* function, 299–302  
 Input capture, 288–302  
 Input port, 2–3  
 Input/output (I/O), 2–5, 10, 47–55, 151–156, 162–172, 178–180, 277–280, 330–389, 500–530  
     application programmer interface (API), 10  
     bandwidth, 4  
     blind cycle counting synchronization, 153, 162–163  
     board support package (BSP), 277–278  
     bus cycle, 3–4  
     busy-to-done state, 151–152  
     busy-wait synchronization, 153, 163–166  
     computer architecture of, 2–5  
     control bus, 3, 5  
     device driver, 10  
     direct memory access (DMA), 4–5, 154  
     embedded computer systems, 10, 50–55  
     gadfly synchronization, 153, 163–166  
     high-speed interfacing, 500–530  
     I/O mapped, 5  
     interfacing, 3, 10, 151–156, 162–172, 178–180  
     interrupts, 153  
     memory-mapped, 2–5  
     parallel ports, 47–55, 166–172, 178–180  
     path expression, 278–280  
     periodic polling, 154  
     port availability, 154–156  
     ports, 2–3, 47–55  
     real-time operating systems (RTOS) and, 277–280  
     serial communications interface (SCI), 10  
     serial devices, 330–389  
     serial peripheral interface (SPI), 10  
     synchronization, 151–154, 163–166  
     unbuffered configuration, 152  
 Input/output parameters (arguments), 114–115  
 Institute of Electrical and Electronics Engineers (IEEE), 24  
 Instruction execution, processor, 6–7  
 Instruction pointer (IP), 90  
 Instruction register (IR), 6  
 Instrumentation amplifier, 544–545  
 Integrator circuit, 545–546  
 Intel x86 processors, memory allocation, 89–90  
 Interface, defined, 3  
 Interface latency, 199, 500–501  
 Interfacing, 3–5, 10, 126–129, 150–188, 189–250, 390–457, 458–499, 500–530, 531–590  
     analog, 3, 531–590  
     application programmer interface (API), 10  
     bandwidth, 151  
     blind cycle counting synchronization, 153, 162–163  
     busy-wait synchronization, 153, 163–166, 176–178  
     computer I/O, 3–5  
     device driver, 10  
     embedded computer I/O, 10  
     gadfly synchronization, 153, 163–166  
     hardware, 449–451  
     HD44780 controller, 178–180  
     high-speed I/O, 500–530  
     input/output (I/O) port availability, 154–156  
     input/output (I/O) synchronization, 151–154  
     interrupt, 189–250  
     key wake-up, 160–162  
     latency, 150  
     liquid-crystal displays (LCD), 178–180  
     low power design, 239–240, 576–577  
     memory, 458–499  
     multiplexed, 402  
     object-oriented, 126–129  
     parallel, 3  
     parallel I/O ports, 166–172, 178–180, 390–457  
     performance measures, 150–160  
     priority and, 151  
     real-time systems, 151  
     relays, 429–432, 442–443  
     scanned, 401–402, 424–427  
     serial, 3  
     serial communications interface (SCI), 10, 173–178  
     serial peripheral interface (SPI), 10  
     servo motors, 452–453  
     solenoids, 432–442  
     stepper motors, 443–452  
     switches, 390–410, 427–429  
     throughput, 151  
     time, 3  
     timing diagrams for, 158–160  
     timing equations for, 156–158  
 Inter-integrated circuit (I<sup>2</sup>C) interface, 368–377  
     9S12 details, 373–377  
         serial I/O devices, 368–373  
 Intermission frame space (IFS), 694

- Intermittent polling, 234–235  
 Internal analog-to-digital converters (ADC), 582–586  
 International Committee for Weights and Measures (CIPM), 24  
 International Electrotechnical Commission (IEC), 24  
 Interrupt, defined, 153, 190  
 Interrupt handler, 218  
 Interrupt interfacing, 153, 189–250  
     9S12 features, 212–216, 235–240  
     critical sections, 199–205  
     edge-triggered requests, 191  
     external design approach, 217–219  
     first-in-first-out (FIFO) queues, 195, 206–212  
     interrupt service routines (ISR), 191–199, 214–217  
     interthread communication, 192–199  
     key wake-up, 222–227  
     linked lists used for, 228–230  
     low power design, 239–240  
     negative-logic requests (IRQ and XIRQ), 190–191  
     periodic, 234–239  
     polled, 219–221, 228–234  
     power systems, 227–228, 239–240  
     priority, 215–217, 230–233  
     pseudo-interrupt vectors, 221–222  
     real-time periodic interrupt (RTI), 234–239  
     reentrancy, 199–205  
     round-robin polling, 233–234  
     software need for, 190–191  
     vectored, 191, 215–217, 218–221, 231–233  
     XIRQ requests, 190–191, 227–228  
 Interrupt service routine (ISR), 131–132, 191–199, 214–217  
     interthread communication, 192–199  
     multithreading, 131–132  
     priority, 215–217  
     software execution of, 214–215  
     synchronization, 191–199, 214–217  
 Interthread communication, 192–199  
     binary semaphore, 193  
     first-in-first-out (FIFO) queues, 195  
     flag actions, 193–194  
     I/O device service, 195–199  
     mailbox synchronization, 194  
     thread synchronization, 193–195  
 Intrusiveness, 137  
 Inversion, priority, 267  
 Inverting amplifiers, 539–541  
 Invocation coupling, 118
- J**  
 Join, threads, 131  
`jxr` instruction, 81, 490  
 Junction field effect transistor (JFET), 603
- K**  
 Karnaugh maps, 463–465  
 Keep it simple stupid approach, 118  
 Key wake-up, 160–162, 222–227  
     9S12 interfacing, 160–162  
     interrupts, 222–227  
 Keyboards, switch interfacing for, 390–410
- L**  
 Label field, 32, 64–65  
 Laplace transform, 718  
 Last-in-first-out (LIFO), 80, 107–108  
 Latched input ports, 47–48  
 Latency, 150, 255, 333  
     device, 500  
     high-speed I/O interfacing, 500–501  
     interface, 150, 500–501  
     real-time operating systems (RTS), 255  
 Layered software systems, 121–123  
`ldaa` instruction, 33  
 Least significant bits (LSB), 25  
 Light-emitting diodes (LED), 50–55, 410–421  
     embedded system design for, 50–55  
     parallel port output interfaces, 410–421  
     multiple (seven-segment) interfaces, 413–421  
     single interface, 411–413  
 Linear ADC, 565  
 Linear circuits, 537–544  
 Linear digital filters, 717–719  
 Linear recursion, 132  
 Linear variable differential transducer (LVDT), 602  
 Linearity, transducers, 597  
 Linked allocation, file system management, 522  
 Linked lists, interrupt polling using, 228–230  
 Linking process, 124  
 Liquid-crystal display (LCD), 13, 178–180,  
     422–427, 511–515  
     computer design using, 13  
     double buffer scheme, 511  
     graphical processing unit (GPU), 514–515  
     graphics controller, 511–515  
     HD44780 controller interfacing, 178–180  
     high-speed I/O interfacing, 511–515  
     MC14500 driver interfacing, 424–421  
     MC14543 driver interfacing, 423–424  
     parallel port interfacing, 178–180, 422–427  
     personal digital assistant (PDA), 514–515  
 Listing file, 63  
 Little endian storage approach, 28  
 Loader, 62  
 Local variables, 105–110  
     defined, 105  
     `static` modifier, 106  
     stack creation of, 107–110  
 Logic level conversion, 377–378  
 Logical cohesion, 119  
 Logical operations, 79–80  
 Logical shift left (LSL), 78  
 Logical shift right (LSR), 77–78  
 Logical-to-physical address translation, 522  
 Longitudinal redundancy check (LRC), 691  
 Lossless data, 688  
 Lossy data, 688  
 Low instruction, 79–80  
 Low power design, 239–240, 576–577  
     analog interfacing, 576–577  
     interrupt interfacing, 239–240
- M**  
 Mailbox synchronization, 194, 271  
 Main program, interrupt software, 218  
 Malloc, 91  
 Manchester encoding, 331–332  
 Mask, defined, 288  
 Masking, 84–85  
 Mass balance, 597  
 Mass storage, high-speed I/O interfacing, 501–502  
 Master-slave configuration, 690  
 Mealy finite-state machine (FSM), 96–97, 101–104  
 Mean time between failures (MTBF), 333  
 Measurand, DAS, 592  
 Median filters, 726–727  
 Memory, 66–68, 89–92  
     allocation, 89–92  
     assembly language programming, 66–68, 89–92  
     global variables, 91–92  
     Intel x86 processors, 89–90  
     pointers, 89–90  
     register transfer operations, 66–68  
     segmentation, 89–90  
 Memory interfacing, 458–499, 505–511  
     9S12 details, 460–461, 480–496  
     address decoders for, 460–467  
     bank-switched, 505–506  
     commands, 459–461  
     direct memory access (DMA), 506–511  
     dual port, 505  
     dynamic RAM (DRAM), 496–497  
     expanded (wide/narrow) modes of, 460, 480–489  
     external bus timing, 471–479  
     flash EEPROM programming, 492–496  
     high-speed I/O, 505–511  
     MC9S12C32 details, 461, 472–479  
     paged memory, 489–492  
     read cycle data, 458–459, 467, 506–507  
     timing, 460–461, 467–479  
     write cycle data, 458–459, 467–468, 506–507

- Memory-mapped I/O, 2–5  
 buses, 3–4  
 computer system, 2–5  
 direct memory access (DMA), 4–5
- Messages, networks, 687, 694–695
- Microcomputer, defined, 2
- Microcomputer-based systems, 1–59, 648–685  
 9S12 architecture, 30–46  
 computer architecture, 2–7  
 control systems, 648–685  
 defined, 7  
 design process, 11–15  
 digital logic, 16–22  
 embedded, 7–10, 50–55  
 input/output (I/O), 2–5, 10, 47–55  
 microcontrollers, 2, 55–56  
 number representation, 22–30  
 open collector logic, 20–22  
 phase-lock-loop (PLL), 36, 46  
 tristate logic, 18
- Microcontrollers, 2, 55–56
- Microphones, 603, 637–639
- Microprocessor, 2
- Minimal-cost address decoding, 463–465
- Mixed-signal design, 540–541
- Modems, 344–345  
 baud rate, 708  
 carrier frequency, 706  
 compression ratio, 709  
 defined, 705  
 encoding, 708–709  
 FSK, 705–708  
 phase-shift keying (PSK), 708–709  
 quadrature amplitude (QAM), 709–710
- Modular software development, 104–121  
 9S12 stack implementation, 107–110  
 assembly language rules for, 119–121  
 C functions, 106–107, 110–111  
 modules, 111–119  
 task division, 115–119  
 variables, 104–111
- Module routing register, 363–368
- Modules, 111–119, 191–199  
 bottom-up programming approach for, 116–117  
 call graphs, 115–116  
 cohesion, 118–119  
 coupling, 115, 118  
 dividing software tasks into, 115–119  
 entry point, 112–114  
 fan-in/fan-out behavior, 119  
 information hiding, 118  
 input/output parameters (arguments), 114–115  
 interrupt service routines (ISR), 191–199  
 keep it simple stupid approach, 118  
 program, 111–115  
 top-down programming approach for, 116
- Monotonic ADC, 565
- Moore finite-state machine (FSM), 96–101
- Most significant bits (MSB), 25
- Motion errors, noise analysis, 624–625
- Motor interfacing, 432–453  
 brushed DC motor, 433  
 brushless DC (BLDC) motor, 434, 440–441  
 servo motor, 452–453  
 solenoids, 432–442  
 stepper motor, 443–452
- mul instruction, 26
- Multi-drop systems, 691–692
- Multi-media card (MMC), 515
- Multiple-access circular queue (MACQ), 580–582
- Multiplexed interface, 402
- Multiplexer, DAS design, 611–612
- Multithreading, 130–132
- Mutual exclusion, thread, 271
- N**
- Naming convention, 95–96
- National Institute of Standards and Technology (NIST), 593
- N-channel metal-oxide semiconductor (NMOS)  
 technology, 30–31
- Negative logic, 157
- Negative-logic interrupt requests (IRQ and XIRQ), 190–191
- Negative predictive value (NPV), 611
- Networks, 503–504, 686–715  
 attenuation, 688  
 channel capacity, 688  
 communications, 503–504, 690–693, 701–710  
 controller area network (CAN), 692–700  
 distortion, 688  
 error detection, 689–692  
 guided medium, 688  
 high-speed I/O interfacing, 503–504  
 messages, 687, 694–695  
 modem communications, 705–710  
 noise, 688  
 parity bits, 689–690  
 SCI serial port communication systems, 690–692  
 unguided medium, 688  
 use of, 686–690  
 wireless communication, 701–705
- Next parameters, 99–100
- Noise, 565, 598, 621–628, 688  
 analysis, 623  
 crosstalk, 688  
 data acquisition systems (DAS), 598, 621–628  
 electromagnetic field induction, 625  
 galvanic, 624  
 measurement of, 625–627  
 motion errors, 624–625  
 networks, 688  
 pink (1/f), 624  
 reduction of, 627–628  
 shot, 624  
 signal-to-noise and distortion ratio (SINAD), 565  
 signal-to-noise ratio (SNR), 598, 688  
 thermal, 621–623  
 white, 622
- Non-return-to-zero (NRZ) encoding, 331, 350
- Non-return-to-zero-inverted (NRZI) encoding, 331, 379
- Nonatomic operations, 202–203
- Noninverting amplifier, 541–544
- Nonlinear digital filters, 717
- Nonlinear transducers, 601
- Normalized mean response time (NMRT), 256
- Null-to-null bandwidth, 701
- Number representation, 22–30  
 16-bit, 27–28  
 8-bit, 25–26  
 abbreviations of, 24–25  
 alternative values, 23  
 American Standard Code for Information Interchange (ASCII) code, 26–27  
 basis, 25  
 big and little endian storage of, 28  
 binary, 22–223  
 characters, 36–27  
 errors, 29–30  
 fixed-point, 28–30  
 hexadecimal, 24  
 one's complement, 25–26  
 positive logic, 22–32  
 precision values, 23  
 two's complement, 26
- Nyquist theory, DAS design, 612–615
- O**
- Object code, 62–64
- Object-oriented interfacing, 126–132
- Offset addressing modes, 70–71
- Offset binary code, 556–557
- Offset current, 535–536
- Offset voltage, 535–536
- One-pole low-pass filters, 548–549
- One's complement numbers, 25–26
- Opcode field, 32–33, 65
- Open collector logic, 20–22

- Open collector programs, 49  
 Open-loop control systems, 649–655  
 Open-loop output impedance, 535  
 Operand field, 33, 65–66  
 Operating modes, 45–46  
 Operation code (opcode), 6, 65  
 Operational amplifiers (op amps), 534–548
  - analog interfacing, 534–548
  - analog isolation, 547–548
  - current-to-voltage circuit, 545
  - derivative circuit, 546
  - hysteresis and, 546–547
  - instrumentation amplifier, 544–545
  - integrator circuit, 545–546
  - inverting amplifier, 539–541
  - linear circuits, 537–544
  - mixed-signal design, 540–541
  - noninverting amplifier, 541–544
  - parameters, 534–536
  - threshold detectors, 537
  - voltage comparators, 546–547
  - voltage-to-current circuit, 545
- Optical channel, serial I/O devices, 345  
`org` directive, 88–89, 103  
 Oscilloscopes, 626–627  
`out` instruction, 5  
 Output compare, 302–310  
 Output port, 3  
 Overflow error, 29  
 Overrun error (OR), 175  
 Overshoot, 663
- P**
- Packages, digital-to-analog converters (DAC), 559–560  
 Packets, 687. *See also* Messages  
 Paged memory, 489–492  
 Parallel I/O ports, 47–55, 166–172, 178–180, 390–457
  - computer-controlled devices, 427–443
  - computer to switch interfacing, 390–391
  - direction register, 49–50
  - embedded system design, 50–55
  - flip flops, 48
  - HD44780 controller LCD interface, 178–180
  - input switch interfaces, 390–410
  - interfacing, 166–172, 178–180, 390–457
  - keyboard interfaces, 390–410
  - latched, 47–48
  - light-emitting diode (LED) output interfaces, 410–421
  - liquid crystal display (LCD) interfaces, 422–427
  - open collector programs, 49
  - relay interfaces, 429–432, 442–443
  - ritual program, 48
  - servo motor interfaces, 452–453
  - solenoid interfaces, 432–442
  - solid-state relay (SSR) interfaces, 442–443
  - stepper motor interfaces, 443–452
  - switch interfaces, 390–410, 427–428
  - transistors and, 427–429
- Parallel programming, 131  
 Parameters, 534–536, 551–553, 564–565, 593–596
  - analog-to-digital converters (ADC), 564–565
  - data acquisition systems (DAS), 593–596
  - digital-to-analog converters (DAC), 551–553
  - operational amplifiers (op amps), 534–536
- Parity, 332–333, 689–690
  - baud rate, 333, 339–340
  - bit time, 332
  - even, 332, 689
  - network error detection, 689–690
  - odd, 332
    - serial I/O devices, 332–333
- Path expression, 278–280  
 PC-relative addressing mode, 33–36  
 Peak bandwidth, 333  
 Performance debugging, 140–143  
 Period measurement, 293–298, 312–316
  - conversion between frequency and, 312–316
- Count instruction, 295–298
- precision, 293  
 range, 293  
 resolution, 293  
 Periodic interrupts, 234–239  
 Periodic polling, 154  
 Periodic thread, 252–253  
 Personal digital assistant (PDA), 514–515  
 Phase-lock-loop (PLL), 36, 46  
 Phase-shift keying (PSK) modem, 708–709  
 Physical plant, 648, 669  
 Pink (1/f) noise, 624  
 Pointers, 89–90, 107–110
  - base (BP), 90
  - code segment selector (CS), 89
  - data segment selector (DS), 90
  - frame, 110
  - instruction (IP), 90
  - memory allocation, 89–90
  - stack (SP), 90, 107–110
  - stack segment selector (SS), 90
  - variable creation using, 107–110
- Polled interrupts, 219–221, 228–234
  - linked lists used for, 228–230
  - priority, 231–232
  - round-robin polling, 233–234
  - vectored interrupts compared to, 219–221
- Polymorphism, 126  
 Position measurement, 636–637  
 Position transducers, 601–603  
 Positive logic, 22–23, 157  
 Positive predictive value (PPV), 611  
 Post processor, 669–670  
 Potentiometers, 636–637  
 Power, 239–240, 558–559, 575–580
  - analog interfacing, 558–559, 575–580
  - analog-to-digital converters (ADC), 575
  - battery, 578–580
  - digital-to-analog converters (DAC), 558–559
  - interrupt interfacing, 239–240
  - low power design, 239–240, 576–577
  - regulators, 575–576
- Power gain, 536  
 Power system interrupt interfacing, 227–228, 239–240  
 Precision, 29, 293, 551, 557, 595–596, 615
  - analog-to-digital converters (ADC), 615
  - data acquisition systems (DAS), 595–596, 615
  - digital-to-analog converters (DAC), 551, 557
  - digital value of, 29
  - period measurement, 293
- Preemptive scheduler, 254  
 Prevalence, DAS design, 611  
 Primary sensing element, DAS, 592  
 Primitive data, 118  
 Printed circuit boards (PCB), 54  
 Printer interface, 162–163  
`printf` statements, 117  
 Priority, 215–217, 230–233, 255–257, 267
  - defined, 215
  - inheritance, 267
  - interrupt order of service examples, 230–233
  - inversion, 267
  - polled interrupts, 231–232
  - rate monotonic theorem, 255
  - schedulers, 255–257, 267
  - vectored interrupts, 215–217, 232–233
- Private variable, 105  
 Procedural cohesion, 119  
 Process, defined, 190  
 Process reaction curve approach, 662–663  
 Process reaction rate, 663  
 Processor, 2, 5–7
  - arithmetic logic unit (ALU), 6
  - bus interface unit (BIU), 5–6
  - control unit (CU), 6
  - effective address register (EAR), 6
  - instruction execution, 6–7
  - operation code (opcode), 6
  - program counter (PC), 6
  - registers, 6
- Profiling, 143–144

- Program counter (PC), 6  
 Program modules, 111–115  
 Programmable read only memory (PROM), 2  
 Programming, 62–92, 119–121, 492–496, 510–511  
     arithmetic operations, 71–77  
     assembly language, 62–92, 119–121  
     branch operations, 83–85  
     direct memory access (DMA), 510–511  
     flash EEPROM, 492–496  
     high-speed I/O interfacing, 510–511  
     indexed addressing modes, 68–71  
     introduction to, 62–64  
     logical operations, 79–80  
     memory allocation, 89–92  
     memory interfacing, 492–496  
     memory operations, 66–68  
     pseudo-ops, 65, 85–89  
     register transfer operations, 66–68  
     shift operations, 77–79  
     stack, 80–83  
     subroutines, 80–82  
     syntax, 64–66  
 Promotion, 29  
 Proportional integral derivative (PID) controllers, 96, 659–668  
     components of, 659  
     derivative term for, 660–662  
     design, 662–668  
     integral term for, 661  
     proportional term for, 661  
 Prototypes, 53–55  
 Pseudocode, 52–53  
 Pseudo-interrupt vectors, 221–222  
 Pseudo-operation code (pseudo-op), 37–39, 65, 85–89  
     9S12C32 microcontroller architecture, 37–39  
     assembly directives, 85–89  
     assembly language programming, 65, 85–89  
     syntax and, 65  
     TExaS assembler applications, 85–86  
 psha instruction, 80–81  
 Public variable, 105  
 pula instruction, 80–81  
 Pulse accumulators, 316–318  
 Pulse-width measurement, 298–302, 318  
 Pulse-width modulation (PWM), 36–37, 307–309,  
     318–322, 563–564  
     9S12 architecture, 36–37  
 MC9S12C32 microcontroller, 318–322  
 digital-to-analog converters (DAC), 563–564  
 timers, 307–309, 318–322
- Q**  
 Quadrature amplitude modem (QAM), 709–710  
 Qualitative performance measurements, 61  
 Quality programming, 60–61  
 Quantitative performance measurements, 61  
 Quantizing, DAS design, 612
- R**  
 R-2R ladder, DAC analog interfacing using, 554–556  
 Random access memory (RAM), 2–5, 91–92, 496–497  
     allocation, 91–92  
     bus cycle, 3–4  
     computer system function of, 2  
     direct memory access (DMA), 4–5  
     dynamic (DRAM), 496–497  
     volatility of, 2  
 Range, 293, 551, 557  
     DAC requirements, 551, 557  
     period measurement, 293  
 Rate monotonic theorem, 255  
 Read cycle data, 458–459, 467, 506–507  
 Read only memory (ROM), 2–4, 91–92  
     allocation, 91–92  
     bus cycle, 3–4  
     computer system function of, 2  
     nonvolatility of, 2  
 Read operation, stacks, 81  
 Real-time operating systems (RTOS), 151, 251–287  
     board support package (BSP), 277–278  
     certification, 257  
     communication, 270–272  
     defined, 151  
     fixed scheduling, 272–277  
     hard, 252  
     hook, 257  
     I/O devices and, 277–280  
     path expression, 278–280  
     round robin scheduler, 254–255, 257–264  
     .schedulers for, 254–264, 272–277  
     semaphores for, 264–270  
     soft, 151, 252  
     synchronization, 264–272  
     thread and, 252–257, 270–272  
     time-jitter, 255–256, 276  
     utilization, 256  
 Real-time periodic interrupt (RTI), 234–239  
 Receiver, controller area network (CAN), 695–698  
 Recursion, 132–134  
 Redefinable equate symbols to values  
     (`set` directive), 86  
 Reed relay interfaces, 432  
 Reentrancy, 130–132, 199–205  
     interrupt synchronization, 199–205  
     threads, 130–132  
 Registers, 6, 31–32, 49–50, 66–68, 71–75  
     9S12 features, 31–32  
     accumulators, 6, 71  
     computer processor function of, 6  
     condition code (CC, CCR), 71–75  
     defined, 71  
     direction, 49–50  
     index registers, 6  
     pointers (addresses), 21  
     program counter (PC), 6, 31  
     stack pointer (SP), 6, 21  
     transfer operations, 66–68  
 Relays, 429–432, 442–443  
     defined, 429  
     electromagnetic (EM), 430–432  
     parallel port interfaces, 429–432, 442–443  
     reed, 432  
     solid-state (SSR), 442–443  
     use of, 430  
 Reliability, serial I/O devices, 333  
 Rendezvous, thread, 270  
 Reproducibility, DAS requirements, 596  
 Reserve multiple 32-bit words (`d1 . 1` directive), 89  
 Reserve multiple bytes (`rmb` directive), 89  
 Reserve multiple words (`ds . w` directive), 89  
 Resistors, analog interfacing uses, 531–532  
 Resolution, 29, 293, 551, 557, 595  
     data acquisition systems (DAS), 595  
     digital value of, 29  
     digital-to-analog converters (DAC), 551, 557  
     instrument, 595  
     period measurement, 293  
     spatial, 595  
 Response time, 663  
 Ring network, 691  
 Ritual program, 48, 218  
 rmb directive, 89  
 Roll (shift) operations, 78–79  
 Round-robin interrupt polling, 233–234  
 Round-robin scheduler, 254–255, 257–264  
 rtc instruction, 491  
 rtci instruction, 192–193, 218  
 rts instruction, 81–83
- S**  
 Sample and hold (S/H), 570–571, 612, 620–621  
     analog interfacing, 570–571  
     DAS design, 612, 620–621  
 Sampling rate, 612–615, 717–718, 736–737  
     DAS design, 612–615  
     digital filter response, 717–718, 736–737  
     FIR filter design, 736–737  
 Saturation, 601  
 Scanned interface, 401–402

- Schedulers, 254–257  
 cooperative (non-preemptive), 254  
 deadlines, 255–256  
 defined, 254  
 fixed, 272–277  
 preemptive, 254  
 priority, 255–256  
 Rate Monotonic theorem, 255  
 real-time operating systems (RTOS), 254–257  
 round robin, 254–255, 257–264  
 thread and, 254–257  
 utilization, 256–257
- Scope of a variable, 105
- Secure digital card (SDC), high-speed  
 I/O interfacing, 515–518
- Seek time, 501–502
- Segmentation, 89–90
- Self-documenting code, 92–96  
 comments, 92–94  
 naming convention, 95–96
- Semaphores, 193, 264–270  
 binary, 193  
 blocking implementation, 267–270  
 interthread communication, 193  
 real-time operating systems (RTOS), 264–270  
 spin-lock implementation, 265–267
- Sensitivity, DAS design, 611
- Sensitivity drift, 599
- Sequential cohesion, 119
- Serial channel, 330–331
- Serial communications interface (SCI), 10,  
 124–125, 173–178, 346–357, 690–692  
 9S12 details, 175–176, 350–357  
 asynchronous mode, 173–175, 346–350  
 busy-wait synchronization, 176–178  
 checksum, 691  
 computer I/O interfacing, 10  
 data transmission, 173–174, 346–347, 691–692  
 design of device driver, 124–125  
 device driver interfacing, 173–178  
 longitudinal redundancy check (LRC), 691  
 master-slave configuration, 690  
 multi-drop systems, 691–692  
 receiving data, 174–175, 347–350  
 ring network, 691  
 serial port communication systems, 690–692  
 synchronization software, 176–178  
 TExaS simulator functions, 178  
 transmission error checking, 691–692
- Serial I/O devices, 330–389. See also Serial  
 communications interface (SCI)  
 9S12 details, 350–357, 363–368, 373–377  
 asynchronous communication systems, 336, 346–350  
 asynchronous communications interface adapter  
 (ACIA), 330  
 balanced differential line protocols, 339–344  
 bandwidth, 333–334  
 baud rate, 333, 339–340  
 channels, 330–331, 343–345  
 current loop channel, 344  
 data communication equipment (DCE), 331  
 data terminal equipment (DTE), 331  
 digital logic channel, 345  
 frames, 332  
 full-duplex communication system, 334  
 half-duplex communication system, 334–335, 343–344  
 inter-integrated circuit ( $I^2C$ ) interface, 368–377  
 logic level conversion, 377–378  
 Manchester encoding, 331–332  
 MC9S12C32 details, 360–363  
 modems, 344–345  
 module routing register, 363–368  
 non-return-to-zero (NRZ) encoding, 331, 350  
 non-return-to-zero-inverted (NRZI) encoding, 331, 379  
 optical channel, 345  
 parity, 332–333  
 RS422 specifications, 331–332, 339–340, 342–343  
 RS485 specifications, 343–344  
 RS423 specifications, 340  
 RS232 specifications, 331–332, 337–341, 350–357
- serial communications interface (SCI), 346–357
- serial peripheral interface (SPI), 357–368
- simplex communication system, 336
- synchronous communication systems, 336–337,  
 357–368
- universal asynchronous receiver/transmitter  
 (UART), 330–331
- universal serial bus (USB), 341–342, 378–383
- Serial peripheral interface (SPI), 10, 357–368  
 9S12 module routing register, 363–368  
 computer I/O interfacing, 10  
 MC9S12C32 details, 360–363  
 synchronous transmission and receiving using,  
 357–360
- Servo motor interfaces, 452–453
- Set counter origin (`ORG` directive), 88–89
- `SET` directive, 86
- Setup time, 158, 166
- Shaft encoder, 451–452
- Shannon-Hartley channel capacity theorem, 701
- Shift operations, 77–79
- Shot noise, 624
- Signal-to-noise and distortion ratio (SINAD), 565
- Signal-to-noise ratio (SNR), 598, 688
- Signals, 460–461, 471–472, 503, 592–593, 612–620  
 9S12 details, 460–461  
 aliasing error, 612–615  
 analog signal processing, 615–620  
 command and timing, 460–46, 471–472  
 data acquisition systems (DAS), 592–593,  
 612–620
- generation, high-speed I/O interfacing, 503
- memory interfacing, 460–461, 471–472
- processing, 592–593
- sampling rate, 612–615
- synchronized versus unsynchronized, 471–472
- Simplex communication system, 336
- Simulation, 53–54
- Single-address DMA cycle, 508–510
- Single chip mode, 45–46
- Slack time, scheduling, 255
- `SLEEP` instruction, 264
- Sleep state, thread, 254
- Slew rate, 16, 536
- Snubber diode, 435
- Soft real time, 151, 252
- Software, defined, 2
- Software maintenance and, 7
- Software overhead, 400
- Software system design, 60–149  
 abstraction, 96–104  
 assembly language programming, 62–92,  
 119–121  
 debugging, 134–144  
 device drivers, 123–125  
 layered software systems, 121–123  
 modular software development, 104–121  
 object-oriented interfacing, 126–129  
 quality programming, 60–61  
 recursion, 132–134  
 self-documenting code, 92–96  
 threads, 130–132
- Solenoid interfaces, 432–442
- Solid-state relay (SSR) interfaces, 442–443
- Sound input/output, 637–639
- Source code, 62–64
- Specificity, DAS design, 611
- Spectrum analyzers, 627
- Speed, analog interfacing parameters,  
 558, 565
- Spin-lock semaphore implementation,  
 265–267
- Sporadic thread, 253
- `STAA` instructions, 68–69, 80
- `STAA PTT` instruction, 39
- Stack pointer (SP), 6, 90, 107–110  
 defined, 6  
 memory allocation and, 90  
 variable creation using, 107–110
- Stack segment selector (SS), 90

- Stacks, 80–83, 107–110  
     9S12 implementation, 107–110  
     access, 109  
     allocation, 109  
     binding, 108  
     deallocation, 109–110  
     frame pointer, 110  
     last-in-first-out (LIFO), 80, 107–108  
     pull operation, 80–81, 107–108  
     push operation, 80–81, 107–108  
     read operation, 81  
     subroutines, 81–83  
     variable creation using, 107–110  
     write operation, 81
- Stamp data, 118
- Starvation, scheduling, 255
- State variables (real and desired), 648–649
- Static efficiency, 61
- static** modifier, 106
- Static operating system, 256
- Static transducers, 596–600
- Statistical control, DAS, 596
- Status flag, 194
- Steady-state controller accuracy, 663
- Stepper motors, 443–452  
     hardware interfaces, 449–451  
     operation of, 446–449  
     parallel port interfacing, 443–452  
     shaft encoder, 451–452  
     use of, 443
- Strain gauges, 605–606, 597
- Subroutines, 80–83, 199–205  
     assembly language programming, 80–83  
     reentrancy and, 199–205  
     stack, 80–83
- Successive approximation ADC, 566–567
- Successive refinement, 52
- Sustained bandwidth, 333
- Switches, 22, 390–410, 427–428  
     capacitors for debouncing, 392–395  
     computer-controlled, 427–428  
     current, 427–428  
     debouncing, 392–401  
     hardware debouncing, 392–395  
     input computer interfaces, 390–410  
     interfaces, 22, 390–410, 427–429  
     multiple keys, 401–410  
     open collector logic, 22  
     parallel port interfaces, 390–410, 427–429  
     software debouncing, 395–401  
     transistors and, 427–428
- Symbol table, 63–64
- Synchronization, 151–154, 162–166, 176–178, 189–250, 264–272  
     blind cycle counting, 153, 162–163  
     busy-to-done state, 151–152  
     busy-wait, 153, 163–166  
     first-in-first-out (FIFO) queues, 195, 206–212, 271  
     flag actions, 193–194  
     gadfly, 153, 163–166  
     input/output interfacing, 151–154, 162–166  
     interrupts, 153, 189–250  
     interthread communication, 193–195  
     mailbox scheme, 194, 271  
     periodic polling, 154  
     real-time operating systems (RTOS), 264–272  
     reentrancy and, 199–205  
     SCI device drivers, 176–178  
     semaphores, 193, 264–270  
     states of I/O, 151–152  
     thread, 193–195, 270–272  
     unbuffered configuration, 152
- Synchronous bus timing, 460, 468
- Synchronous communication systems, 336–337, 357–368  
     serial I/O devices, 336–337  
     serial peripheral interface (SPI), 357–368  
     transmission and receiving using, 357–360
- Syntax, 64–66  
     label field, 64–65  
     operand field, 65–66  
     operation field, 65
- T**
- Tail recursion, 132
- TCNT counter, 97–98, 289–291
- Temperature measurement, 606–610, 629–633
- Temporal cohesion, 119
- Test and set functions, 205
- TExaS assembler, 62–64, 85–86, 178  
     busy-wait SCI functions, 178  
     assembly language programming and, 62–64  
     CodeWarrior compared to, 62, 64  
     pseudo-op applications, 85–86
- Thermal noise, 621–623
- Thermistors, 597, 607–608, 629–633
- Thermocouples, 597, 608–610
- Thread switch, 214
- Thread, 130–132, 144, 190, 192–199, 252–257, 270–272  
     active state, 253  
     aperiodic, 253  
     background, 131, 192  
     blocked state, 253  
     defined, 190  
     first-in-first-out queue for, 271–272  
     foreground, 131, 192  
     interrupt synchronization and, 190, 192–199  
     interthread communication, 192–199  
     mailbox communication, 271  
     multithreading, 130–132  
     periodic, 252–253  
     profiling, 144  
     real-time operating systems (RTS) and, 252–257  
     reentrancy and, 130–132  
     rendezvous, 270  
     resource sharing, 271  
     .schedulers and, 254–257  
     single execution, 130  
     sleep state, 254  
     sporadic, 253  
     synchronization, 193–195, 270–272
- Threshold detectors, 537
- Throughput, 151, 333
- Time quantizing, 612
- Time-jitter, 255–256, 276, 734–735  
     digital filter effects, 734–735  
     real-time operating systems, 255–256, 276
- Time-to-deadline, 255
- Timer\_Wait directive, 98
- Timers, 97–104, 288–329  
     9S12 details, 97–98, 288–291, 316–318  
     built-in, 98–104  
     C code implementation, 294–295, 306–307, 311–312  
     conversion between frequency and period, 312–316  
     delay software, 98–104  
     delayed pulse generation, 309–310  
     finite-state machines (FSM) for, 97–104  
     frequency measurement, 310–317  
     input capture, 288–302  
     MC9S12C32 details, 318–322  
     output compare, 302–310  
     period measurement, 293–298, 312–316  
     pulse accumulators, 316–318  
     pulse-width measurement, 298–302, 318  
     pulse-width modulation (PWM), 307–309, 318–322  
     TCNT counter, 97–98, 289–291  
     time generation, 288–310
- Timing, 156–160, 460–461, 467–479  
     9S12 details, 460–461  
     asynchronous bus, 469–471  
     bus, 460–461, 467–479  
     diagrams, 158–160  
     equations, 156–158  
     external bus, 471–479  
     interfacing, 156–160  
     MC9S12C32 details, 472–479  
     memory interfacing, 460–461, 467–479

- signals, 460, 471–472  
 synchronized versus unsynchronized signals, 471–472  
 synchronous bus, 460, 468  
 Top-down design, 11–15, 116  
   analysis phase, 11  
   engineering phase, 13  
   high-level phase, 12–13  
   implementation phase, 14  
   maintenance phase, 15  
   modular software development and, 116  
   testing phase, 14–15  
 Total harmonic distortion (THD), 565  
 Transducer, DAS, 592  
 Transducers, 596–611  
   data acquisition systems (DAS), 596–611  
   drift, 599  
   dynamic, 600  
   force, 605–606  
   impedance, 598–599  
   linearity of, 597  
   manufacturing issues, 599–600  
   nonlinear, 601  
   position, 601–603  
   sensitivity of, 597–598  
   specificity of, 598  
   static, 596–600  
   temperature, 606–610  
   thermistors, 607–608  
   thermocouples, 608–610  
   velocity measurements, 603–605  
 Transient response, 600  
 Transistors, current switch interfacing using, 427–428  
 Transistor-transistor logic (TTL), 16–18  
 Transition time, 16  
 Transmission error checking, 691–692  
 Trellis-coded quadrature amplitude modem (TCQAM), 709–710  
 Tristate logic, 18  
`tsx` instruction, 81, 107, 110  
 Two's complement numbers, 26
- U**  
 Unbuffered configuration, 152  
 Undershoot, 663  
 Unguided medium, 688  
 Universal asynchronous receiver/transmitter (UART), 330–331  
 Universal serial bus (USB), 341–342, 378–383  
   integrated interface, 383  
   modular interface, 382–383  
   non-return-to-zero-inverted (NRZI) encoding, 379  
   serial I/O devices, 341–342, 379–382  
 Utilization, RTOS, 256–257
- V**  
 Variables, 104–111, 592, 648–649  
   9S12 stack implementation fpr, 107–110  
   C functions, 106–107, 110–111  
   const modifier, 106  
   conversion element, DAS, 592  
   creation of, 106–110  
   control systems, 648–649  
   global, 91–92, 105–106  
   local, 105–110  
   private, 105  
   public, 105  
   scope of, 105  
   stack pointers (SP) for, 107–110  
   state (read and desired), 648–649  
   static modifier, 106  
 Vectored interrupts, 191, 215–217, 218–222, 231–233  
 defined, 191  
 external interrupt design using, 218–219  
 interrupt service routines (ISR), 214–217  
 polled interrupts compared to, 219–221  
 priority, 215–217, 232–233  
 pseudo, 221–222  
 Velocity measurements, transducers, 603–605  
 Video displays, high-speed I/O interfacing, 503  
 Voltage comparators, 546–547  
 Voltage, op amp parameters, 535–536  
 Voltage noise, 536  
 Voltage quantizing, 612  
 Voltage thresholds, 17–18  
 Voltage-to-current circuit, 545  
 Vulnerable windows, 199–205
- W**  
 Waveform generation, 560–563  
 Weighted round robin scheduler, 254–255  
`while` operation, 84–85  
 White noise, 622  
 Wireless communication, 701–705  
 Write cycle data, 458–459, 467–468, 506–507  
 Write operation, stacks, 81
- X**  
 x86, 89–90  
 XGate, 36  
 XIRQ interrupt requests, 190–191, 227–228
- Z**  
 Z transform, 718–719  
 Zero drift, 599  
 Zero-page addressing mode, 34  
 Zero-time reference, 156  
 ZigBee, 702–703











Pin assignments for the RS232 EIA-574 and EIA-561 protocols

<b>DB25 Pin</b>	<b>RS232 Name</b>	<b>DB9 Pin</b>	<b>EIA-574 Name</b>	<b>RJ45 Pin</b>	<b>EIA-561 Name</b>	<b>Signal</b>	<b>Description</b>	<b>True (V)</b>	<b>DTE</b>	<b>DCE</b>
1						FG	Frame Ground/Shield			
2	BA	3	103	6	103	TxD	Transmit Data	-12	out	in
3	BB	2	104	5	104	RxD	Receive Data	-12	in	out
4	CA	7	105/133	8	105/133	TSR	Request to Send	+12	out	in
5	CB	8	106	7	106	CTS	Clear to Send	+12	in	out
6	CC	6	107			DSR	Data Set Ready	+12	in	out
7	AB	5	102	4	102	SG	Signal Ground			
8	CF	1	109	2	109	DCD	Data Carrier Detect	+12	in	out
9							Positive Test Voltage			
10							Negative Test Voltage			
11							Not Assigned			
12						sDCD	Secondary DCD	+12	in	out
13						sCTS	Secondary CTS	+12	in	out
14						sTxD	Secondary TxD	-12	out	in
15	DB					TxC	Transmit Clk (DCE)		in	out
16						sRxD	Secondary RxD	-12	in	out
17	DD					RxC	Receive Clock		in	out
18	LL						Local Loopback			
19						sRTS	Secondary RTS	+12	out	in
20	CD	4	108	3	108	DTR	Data Terminal Rdy	+12	out	in
21	RL					SQ	Signal Quality	+12	in	out
22	CE	9	125	1	125	RI	Ring Indicator	+12	in	out
23						SEL	Speed Selector DTE		in	out
24	DA					TCK	Speed Selector DCE		out	in
25	TM					TM	Test Mode	+12	in	out

General specification of various types of capacitor components<sup>2, 3</sup>

<b>Type</b>	<b>Range</b>	<b>Tolerance</b>	<b>Temperature coef</b>	<b>Leakage</b>	<b>Frequencies</b>
Polystyrene	10 pF to 2.7 $\mu$ F	$\pm 0.5$	Excellent	10 G $\Omega$	0 to $10^{10}$ Hz
Polypropylene	100 pF to 50 $\mu$ F	Excellent	Good	Excellent	
Teflon	1000 pF to 2 $\mu$ F	Excellent	Best	Best	
Mica	1 pF to 0.1 $\mu$ p	$\pm 1$ to $\pm 20\%$		1000 M $\Omega$	$10^3$ to $10^{10}$ Hz
Ceramic	1 pF to 0.01 $\mu$ F	$\pm 5$ to $\pm 20\%$	Poor	1000 M $\Omega$	$10^3$ to $10^{10}$ Hz
Paper (oil-soaked)	1000 pF to 50 $\mu$ F	$\pm 10$ to $\pm 20\%$		100 M $\Omega$	100 to $10^8$ Hz
Mylar (polyester)	5000 pF to 10 $\mu$ F	$\pm 20\%$	Poor	10 G $\Omega$	$10^3$ to $10^{10}$ Hz
Tantalum	0.1 $\mu$ F to 220 $\mu$ F	$\pm 10\%$	Poor		
Electrolytic	0.47 $\mu$ F to 0.01 F	$\pm 20\%$	Ghastly	1 M $\Omega$	10 to $10^4$ Hz

<sup>2</sup> Modified from Wolf and Smith, *Student Reference Manual*, Prentice Hall, pg. 302, 1990.

<sup>3</sup> From Horowitz and Hill, *The Art of Electronics*, Cambridge University Press, pg. 22, 1989.

## Examples of systems and component interfaces

---

- 60 Hz digital notch filter, section 15.4, page 729  
ADC device driver, program 11.13, page 583  
ADC, 12-bit Max1247, example 7.5, page 365  
Analog amp, inverter, example 11.1, page 540  
Analog amp, multiple inputs, example 11.2, page 543  
Battery charger, Li-ion, figure 11.74, page 580  
Battery charger, NiMH, figure 11.73, page 579  
Brushless DC motor, example 8.10, page 440  
Butterworth low pass filter, figure 11.29, page 549  
CAN controller network, program 14.1, page 699  
Count edges, program 6.1, page 292  
DAC, 12-bit DAC8043, example 7.4, page 368  
DAC, using PWM, example 11.3, page 563  
DC motor H bridge, figure 8.71, page 440  
DC motor interface, example 8.9, page 437  
DC motor isolated interface, figure 8.65, page 438  
DC motor MOSFET interface, figure 8.64, page 438  
Debugging dump, program 2.26, page 139  
Debugging monitor, program 2.28, page 140  
Debugging profile, program 2.35, page 143  
Digital filter, FIR, section 15.7, page 736  
Digital logic interface, section 1.4, page 22  
Discrete Fourier Transform, section 15.6, page 736  
DTR communication, example 7.2, page 355  
EKG data acquisition, example 12.2, page 633  
Fifo queue, program 4.14, page 209  
File system, section 10.7, page 519  
Finite state machine (Mealy), example 2.2, page 101  
Finite state machine (Moore), example 2.1, page 98  
Fixed thread scheduler, program 5.21, page 275  
Frequency measurement, 100 Hz, example 6.10, page 314  
Frequency measurement, 0.1 Hz, example 6.8, page 310  
Fuzzy logic controller, example 13.7, page 671  
I<sup>2</sup>C interface, program 7.9, page 377  
Instrumentation amp, figure 11.17, page 544  
Integration amp, figure 11.20, page 546  
Keyboard, matrix scan, example 8.5, page 404  
LCD graphics controller, figure 10.18, page 512  
LCD using HD44780, section 3.7, page 178  
LED display, 3-digit, example 8.7, page 415  
LED display, 5-digit, example 8.8, page 419  
LED interface, figure 8.25, page 411  
Logic level conversion, figure 7.54, page 378  
Mail box synchronization, Figure 4.5, page 194  
Median filter, program 15.6, page 727  
Microphone, electret, figure 12.11, page 603  
Multi-drop network, figure 14.5, page 692  
Multiple access circular queue, program 11.12, page 581  
Output port expander, example 7.6, page 367  
Path expression, section 5.6.2, page 278  
Period measurement, 32-bit, example 6.3, page 295  
Period measurement, example 6.1, page 291  
Period measurement, input capture, example 6.2, page 293  
Periodic interrupt, program 4.31, page 239  
Phase lock loop (PLL), section 1.8, page 46  
PI position controller, example 13.6, page 665  
Piano keyboard, example 8.4, page 402  
PID velocity controller, example 13.5, page 664  
Position controller, example 13.4, page 657  
Position measurement, example 12.3, page 636  
Programming flash, section 9.7, page 492  
Pulse width measurement, example 6.6, page 301  
Pulse-width modulation, 16-bit, program 6.16, page 322  
Pulse-width modulation, 8-bit, program 6.15, page 321  
Pulse-width modulation, section 6.2.3, page 307  
RAM interface, 16-bit, example 9.7, page 488  
RAM interface, 8-bit, example 9.6, page 481  
Real-time OS, program 5.7, page 261  
Resistance measurement, example 6.4, page 298  
Ring network, figure 14.4, page 691  
Robotic arm, example 13.2, page 650  
RTI periodic interrupt, program 4.29, page 237  
SCI serial using busy waiting, program 3.10, page 177  
SCI serial using interrupts, example 7.1, page 352  
Secure digital card (SDC), section 10.6, page 515  
Servo motor interface, section 8.7, page 452  
Solid-state relay interface, figure 8.75, page 443  
Sound recording, example 12.4, page 637  
Square wave generation, example 6.7, page 306  
Stepper motor controller, example 8.11, page 443  
Stepper motor, bipolar, figure 8.83, page 451  
Stepper motor, unipolar, figure 8.81, page 450  
Switch debounce, example 8.1, page 396  
Temperature controller, example 13.3, page 655  
Thermistor-based thermometer, example 12.1, page 629  
Time delay, section 2.4.3, page 98  
TOF periodic interrupt, program 4.30, page 238  
USB interface, figure 7.59, page 382  
Voltage comparator, figure 11.23, page 547  
Voltage regulation, figure 11.69, page 576  
Waveform generation, figure 11.50, page 560  
Wireless communication, example 14.1, page 701  
XON/XOFF communication, example 7.3, page 356  
ZigBee communication, figure 14.12, page 703