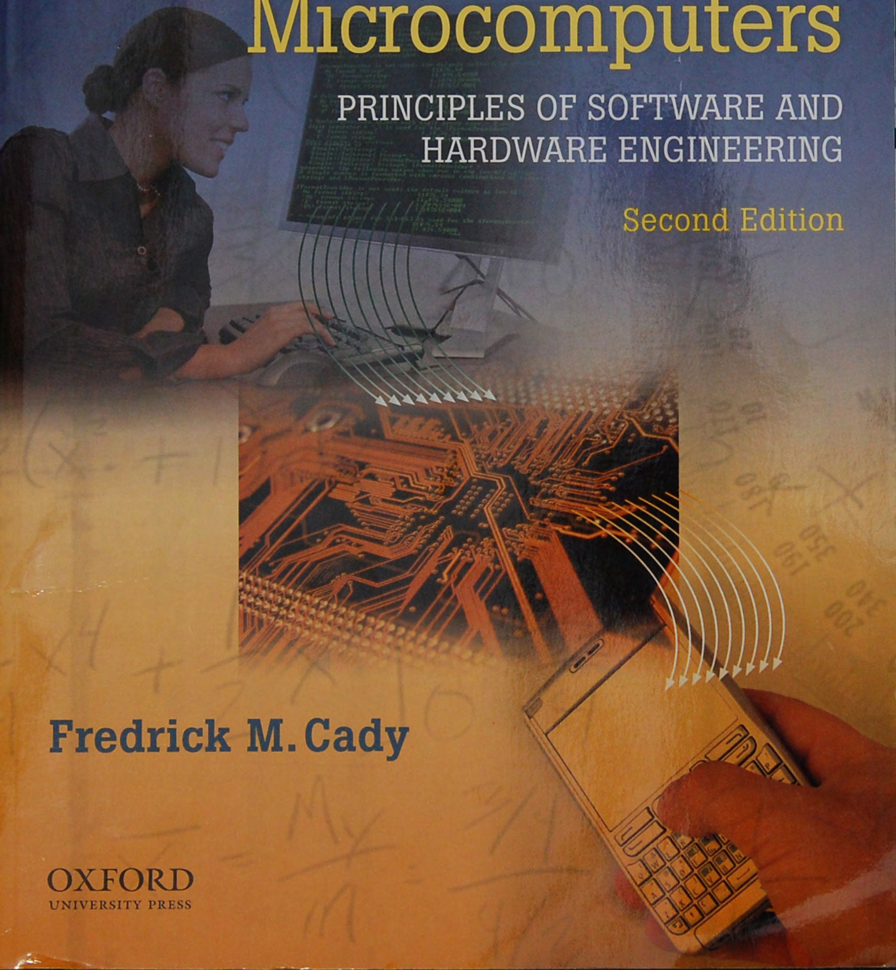
Microcontrollers and Microcomputers

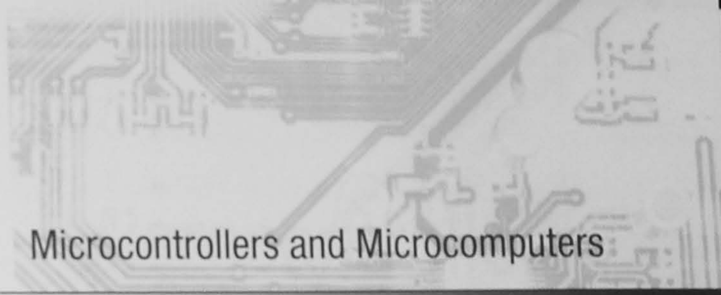
PRINCIPLES OF SOFTWARE AND HARDWARE ENGINEERING

Second Edition

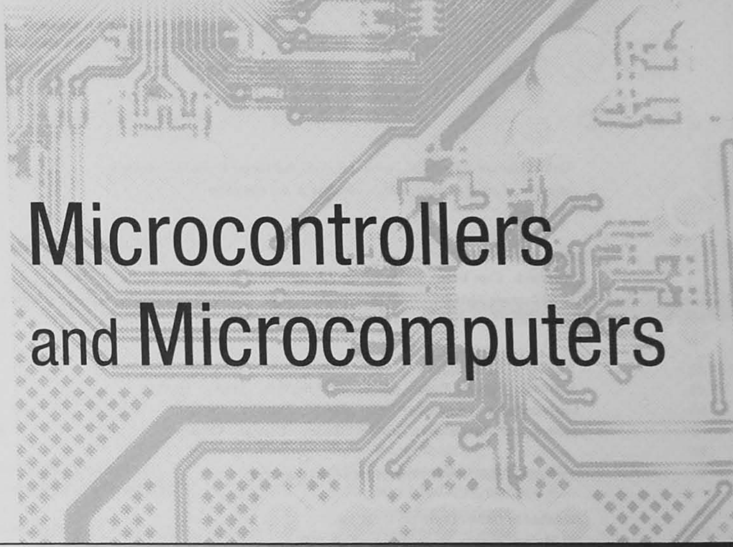


Fredrick M. Cady

OXFORD
UNIVERSITY PRESS



Microcontrollers and Microcomputers



Microcontrollers and Microcomputers

Principles of Software and Hardware Engineering

Second Edition

Fredrick M. Cady

Department of Electrical and Computer Engineering
Montana State University

New York Oxford
OXFORD UNIVERSITY PRESS
2010

Oxford University Press, Inc., publishes works that further Oxford University's objective of excellence in research, scholarship, and education.

Oxford New York
Auckland Cape Town Dar es Salaam Hong Kong Karachi
Kuala Lumpur Madrid Melbourne Mexico City Nairobi
New Delhi Shanghai Taipei Toronto

With offices in
Argentina Austria Brazil Chile Czech Republic France Greece
Guatemala Hungary Italy Japan Poland Portugal Singapore
South Korea Switzerland Thailand Turkey Ukraine Vietnam

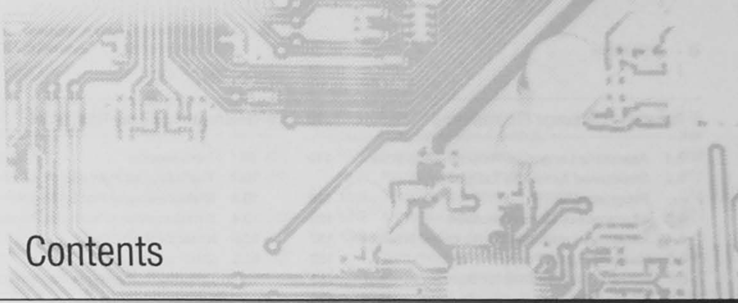
Published by Oxford University Press, Inc.
198 Madison Avenue, New York, New York, 10016
http://www.oup-usa.org

Oxford is a registered trademark of Oxford University Press

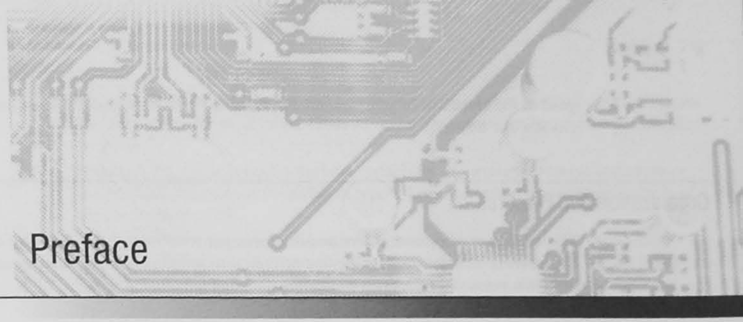
Library of Congress Cataloging-in-Publication Data

Cady, Fredrick M., 1942–
Microcontrollers and microcomputers : principles of software and hardware engineering / Fredrick M. Cady. — 2nd ed.
p. cm.
Includes bibliographical references.
ISBN 978-0-19-537161-1
1. Microcontrollers. 2. Microcomputers. 3. Software engineering. I. Title.
TJ223.P76C33 2009
004.16—dc22 2008045026

Printing number: 9 8 7 6 5 4 3 2 1
Printed in the United States of America
on acid-free paper



Contents



Preface

Much has changed since the first edition of *Microcontrollers and Microcomputers: Principles of Software and Hardware Engineering*. Many more microcontrollers are available, and advances in integrated circuit technology now allow many features needed in an embedded system to be integrated into the microcontroller, making system design much easier.

As the hardware has improved over the years, so has the software development environment. In earlier days, software was written in assembly (or C) and downloaded to RAM for testing. ROM-based debugging monitor programs allowed students to run their programs, set breakpoints, trace and inspect registers and memory to verify that the program was running correctly, and debug it if it was not. Sometimes simulators were used, but these were limited in how they simulated input and output functions. Nowadays we implement many software designs in a high-level language, primarily C. Development environments like Freescale Semiconductor's CodeWarrior® offer assembly, C, and C++ languages along with sophisticated, high-level debugging tools. Chip manufacturers, too, have integrated into their microcontroller chips development and debugging features such as Background Debugging (Freescale), On-Chip Debugging (Atmel, Microchip), Embedded ICE® (Cirrus Logic), and JTAG Debug (Maxim, TI). These hardware and software tools, along with copious amounts of Flash program memory, have made the development of embedded applications much easier than in the past. Nonetheless, students must have basic knowledge about the microcontrollers and microcomputers before they can apply the new tools in embedded systems.

This edition of *Microcontrollers and Microcomputers: Principles of Software and Hardware Engineering* is designed to be an ideal introductory text in an embedded system or microcontroller course. It is not targeted toward, nor does it describe, any specific microcontroller or microprocessor. While there are some assembly language code examples taken from the Freescale HCS12 microcontroller, these serve only as an introduction to lead students into the assembly language of their own microcontroller's. The material in this book is aimed at sophomore, junior, or senior electrical engineering, computer engineering, or computer science students taking a first course in embedded systems or microcontrollers. Although many program examples are given in C and a chapter is devoted to the use of C in an embedded application, students will benefit by having a prerequisite high-level language course, preferably C. Because the text is purposefully not processor-specific, it can be used with processor-specific material, such as manufacturers' data sheets and reference manuals, or with texts such as *Software and Hardware Engineering: Motorola M68HC11*, or *Software and Hardware Engineering: Motorola M68HC12*.

The fundamental operation of standard microcontroller features such as parallel and serial I/O interfaces, interrupts, analog-to-digital conversion, and timers is covered, with attention paid to the electrical interfaces needed. One chapter is devoted to showing how a variety of devices can be used.

rganization and Features

This edition of *Microcontrollers and Microcomputers: Principles of Software and Hardware Engineering* includes much of the first edition, with additional chapters and material developed in the ten years since the book was originally published.

1. **Introduction:** The introduction contains a description of the von Neumann and Harvard architectures and gives definitions used throughout the text. A time line gives a brief history of microcontroller development.
2. **General Principles of Microcontrollers:** The general operating principles of microcontrollers are illustrated by means of a register-transfer design of a processor, dubbed the picocontroller. Students are shown how defining a set of instructions can lead to a hardware design, including the sequential controller needed in a stored program computer. The chapter also describes the software development environment students will use in the laboratory.
3. **Structured Program Design:** The principles of top-down design are presented in this chapter. Students should be exposed to the need for software design before starting to write programs for their microcontroller.
4. **Introduction to the CPU: Registers and Condition Codes:** This chapter describes the registers found in all microcontrollers. Students are asked to describe their own processor's registers and condition codes in chapter exercises.
5. **Memory Addressing Modes:** This chapter, which will be useful in courses that teach an assembly language, describes the addressing modes commonly used in microcontrollers.
6. **Assembly Language Programming:** This chapter shows the students how they might organize an assembly language program and how to accomplish structured programming constructs in assembly. Although this text is processor independent, Freescale Semiconductor HCS12 assembly code is used in examples.
7. **C Programming for Embedded Systems:** This chapter shows how programming in C for an embedded system differs from programming for a desktop computer application. It is not an introductory instruction in C programming. We assume that students have learned C in another programming course, there are many programming examples in C throughout the text.
8. **Debugging Microcontroller Software and Hardware:** Almost all of us download our programs, push the run button, and expect the program to work the first time. Almost all of us are disappointed. Programs rarely work the first time. This chapter presents some debugging strategies and techniques and shows how a modern program debugger, such as CodeWarrior, can be used. It also shows some of the common mistakes beginning assembly and C language programmers make and gives hints on how to find them.
9. **Computer Buses and Parallel I/O:** This chapter describes parallel I/O interfaces, microcontroller I/O, and I/O software synchronization.

10. **Interrupts and Real-Time Events:** The general principles of interrupts appropriate for any microcontroller are covered. Hints for writing interrupt service routines or handlers are given.
11. **Memory:** This chapter covers the basic principles of memory elements and memory architectures. We explain the different types of memory and discuss the interaction of memory with the CPU.
12. **Serial I/O:** Many engineers have a terrible time with serial interfaces, especially the RS-232-C "standard," because they do not understand how all the signals in the standard interface are used. This chapter describes the asynchronous serial I/O and its RS-232-C standard interface. Other electrical interfaces used in serial, including RS-422, RS-423, and RS-485 I/O, are described. The synchronous serial peripheral interface (SPI) and the inter-integrated circuit (IIC or I^{2}C) interfaces are shown, and the controller area network (CAN) bus is introduced. Examples showing how to use the SPI to send data to an LCD module and the I^{2}C to read a temperature sensor are given.
13. **Analog Input and Output:** Because computers must read analog information and act upon it, this chapter considers the world of analog signals. System aspects of the analog-to-digital conversion process are given with design procedures. Both analog-to-digital and digital-to-analog converter types are described.
14. **Counters and Timers:** Many embedded applications require a timer to generate waveforms of a specific frequency, to time external events, to count events, and to generate interrupts at specific intervals. This chapter looks at the basic operation of the timer circuits found in modern microcontrollers.
15. **Single-Chip Microcontroller Interfacing Techniques:** This, the largest chapter in the book, describes a variety of real-world interfaces not covered in earlier chapters. These include simple input and output devices, switches, LEDs, and keypads. Parallel I/O expansion sometimes needed for microcontrollers with limited I/O is shown. Input and output I/O electronics show students how to protect the fragile microcontroller from the cruel world. DC and stepper motors are described, and C program stepper motor driver modules are given.
16. **Real-Time Operating Systems:** Our final chapter is a brief overview of real-time operating systems. It can serve as an introduction for students going on to a more advanced microcontroller course that will use real-time system.

The **Appendix** contains a review of binary codes and binary arithmetic.

Each chapter has a variety of relevant problems. We subscribe to POGIL, the active learni paradigm for process oriented, guided inquiry learning. (see http://www.pogil.org). The pro lem set questions are grouped into those that allow students to explore the body of knowled to stimulate their thought processes, and to challenge them to expand their knowledge a expertise. Finally, the POGIL model asks students to look back, or reflect on what they h learned. The text contains answers to selected problems.

A letter or letters in [. . .] at the end of each of the end of chapter problems signifies that problem in some way tests that the student meets ABET accreditation criteria for Outco a–k as follows:

a. An ability to apply knowledge of mathematics, science, and engineering.
b. An ability to design and conduct experiments, as well as to analyze and interpret da
c. An ability to design a system, component, or process to meet desired needs.

d. An ability to function on multi-disciplinary teams.
e. An ability to identify, formulate, and solve engineering problems.
f. An understanding of professional and ethical responsibility.
g. An ability to communicate effectively.
h. The broad education necessary to understand the impact of engineering solutions in a global and societal context.
i. A recognition of the need for, and an ability to engage in lifelong learning.
j. A knowledge of contemporary issues.
k. An ability to use the techniques, skills and modern engineering tools necessary for engineering practice.

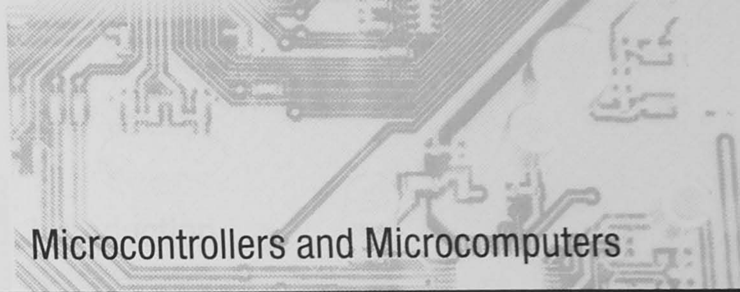
Acknowledgment

Thanks go to Al Gray, one of my first bosses and mentors, who got me started in computer engineering many, many years ago, before computer engineering was even thought of. Those were the pre microprocessor days, when computers, while far different from those we now have, were in reality much the same. Learning the fundamentals of the early computers has enabled me to learn easily a variety of others in the intervening years. It is the aim of this text to give students the same, solid grounding I gained with Al's help.

The following reviewers also provided helpful feedback on this second edition: Ji Chen, University of Houston; Steve Chiu, Idaho State University; Fernando Rios-Gutierrez, University of Minnesota, Duluth; Michael Morrow, University of Wisconsin-Madison; Albert O. Richardson, California State University, Chico; Harry W. Tyrer, University of Missouri-Columbia; Aleksandar Milenkovic, The University of Alabama in Huntsville; James M. Conrad, UNC Charlotte; Ming Yu, Florida State University; Jerome Butler, Southern Methodist University.

Thanks also go to the team extraordinaire at Oxford University Press who helped and encouraged me throughout the publication process. Danielle Christensen, as engineering editor, accepted my proposal and enthusiastically carried it forward for approval. John Challice, succeeding Danielle as engineering editor, has kept me on track through the process. Editorial assistant Adriana Hurtado patiently answered all my questions along the way, and associate editor Rachael Zimmermann has handled ancillaries and, taking over from Adriana Hurtado, fielded my questions through the home stretch.

Fredrick M. Cady



Microcontrollers and Microcomputers

1 Introduction

1.1 Computers, Microprocessors, Microcomputers, Microcontrollers

A computer system is shown in Figure 1-1. We see a *CPU,* or *central processor unit, memory (ROM* and *RAM),* containing the program and data, an *I/O interface* with associated *input and output ports,* and three *buses* connecting the elements of the system together. The organization of the program and data into a single memory block is called a *von Neumann* architecture, after John von Neumann, who described this general-purpose, stored-program computer in 1945. In Figure 1-1 the data, address, and control buses consist of many wires, for example 8, 16, 32

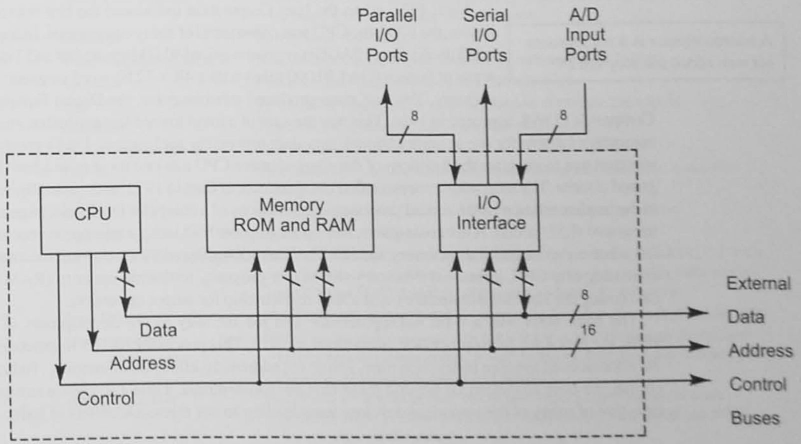


Figure 1-1 Von Neumann computer architecture.

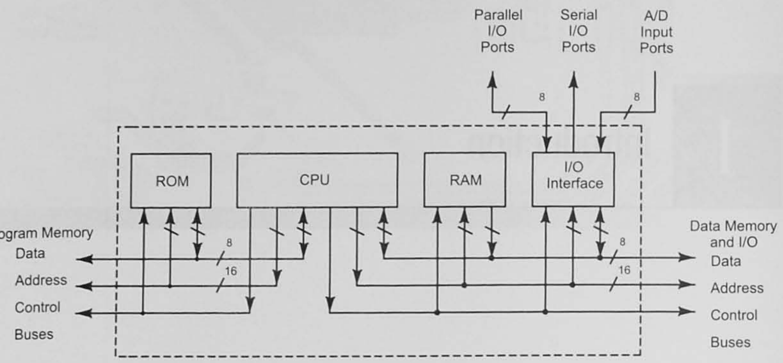


Figure 1-2 Harvard computer architecture.

or more, that carry binary signals from one place to another in the computer system. This is a classical computer system block diagram, and all computers discussed in this text have this basic architecture.

There is another major computer architecture type called the *Harvard* architecture in which two completely separate memories are used—one for the program and one for the data. This architecture is often found in digital signal processing (DSP) chips and some other microcontroller chips such as Microchip Technology PIC microcontrollers (Figure 1-2).

microcomputer is a microproces-
or with added memory and I/O.

Until 1971, when the Intel Corporation introduced the first microprocessor, the 4004, the CPU was constructed of many components. Indeed, in 1958 the Air Force SAGE computer required 40,000 square feet and 3 megawatts of power; it had 30,000 tubes with a 4K x 32 bit word magnetic core memory. The first mass-produced minicomputer, the Digital Equipment Company's PDP-8, appeared in 1964. This was the start of a trend toward less expensive, smaller computers suitable for use in nontraditional, non–data processing applications. Intel's great contribution was to integrate the functions of the many-element CPU into one (or at most a few) integrated circuits. The term *microprocessor* first came into use at Intel in 1972[1] and, generally, refers to the implementation of the central processor unit functions of a computer in a single, large scale integrated (LSI) circuit. A *microcomputer*, then, is a computer built using a microprocessor and a few other components for the memory and I/O. The Intel 4004 allowed a four-chip microcomputer consisting of a CPU, a read-only memory (ROM) for program, read/write memory (RAM) for data (using the Harvard architecture), and a shift register chip for output expansion.

The Intel 4004 was a 4-bit microprocessor and led the way to the development of the 8008, the first 8-bit microprocessor, introduced in 1972. This processor had 45 instructions, a 30-microsecond average instruction time, and could address 16 kilobytes of memory. Today, of course, we have advanced far beyond these first microcomputers. Table 1-1 gives a summary time line of many of the important developments leading to our microcontrollers of today.

[1] R. N. Noyce and M. E. Hoff Jr., *A History of Microprocessor Development at Intel*. IEEE MICRO, February 1981.

Table 1-1 Microcomputer Development Time Line

Year	Computer	Event
mid-1800s	Charles Babbage difference engine	A difference engine was completed in 1991 at the Science Museum in London to Babbage's original plans. It had around 4000 parts and weighed almost 3 tons. It successfully calculated a result to 31 digits.
1944	IBM Automatic Sequence Controlled Calculator	Also called the Harvard Mark I computer, it introduced the Harvard architecture with separate data and program memory. Built with switches, relays, and other mechanical components, it had over 700,000 components, and weighed 10,000 pounds.
1945	von Neumann machine described	While working on the EDVAC computer project, John von Neumann described a stored-program computer with data and program in the same memory.
1946	ENIAC	Electronic Numerical Integrator and Computer. With over 17,000 vacuum tubes and 7200 crystal diodes, it weighed 27 tons and consumed 150 kW of power.
1947	Point contact transistor invented	John Bardeen and Walter Brattain at AT&T Bell Labs.
1948	Junction transistor invented	William Shockley at AT&T Bell Labs.
1951	EDVAC	The Electronic Discrete Variable Automatic Computer was a successor to ENIAC. It computed in binary instead of decimal.
1951	Magnetic core memory invented	Jay Forrester at MIT based his invention on work by An Wang at Harvard University in 1949.
1958	Integrated circuit invented	Jack Kilby at Texas Instruments.
1960	MOS transistor invented	John Atalla and Dawon Kahng at AT&T Bell Labs and Robert Noyce at Fairchild Semiconductor.
1963	CMOS transistor invented	C. T. Sah and Frank Wanlass; Fairchild R & D Laboratory.
1964	First static RAM	64-bit memory, from Fairchild Semiconductor.
1964	PDP-8	Digital Equipment Corporation's first mass-produced minicomputer.
1964	Control Data Corporation CDC 6600	First reduced instruction set computer (RISC).
1965	Moore's law proposed	Gordon Moore at Fairchild Semiconductor predicted that the number of components per chip would double every one to two years.
1970	Intel 1103	First dynamic RAM chip, 1 Kbit.
1970	Three-state logic invented	National Semiconductor (now identified by trademark name Tristate)
1971	Intel 4004	First microprocessor: 2300 transistors, 740 kHz clock.
1971	Intel 1702	First erasable programmable read-only memory (EPROM); 256 x 8 bits.
1972	Intel 8008	First 8-bit microprocessor: 3500 transistors, 800 kHz clock.
1972	Hewlett-Packard HP-35	First pocket scientific calculator.
1973	IMP-16	First multichip 16-bit microprocessor; from National Semiconductor. It used five integrated circuits.
1974	PACE	First single-chip, 16-bit microprocessor; from National Semiconductor.
1974	Intel 8080	6000 transistors, 2 MHz clock.
1975	MIT's Altair 8800 computer	First hobbyist computer based on the Intel 8080. It had 4K and 8K BASIC, 4 K RAM, and introduced the S-100 bus standard. The complete kit, including extra memory and I/O, cost \$1400 (\$5800 in 2008 currency adjusted for inflation).
1976	RCA 1802	RCA COSMAC, the first CMOS microprocessor, was used in space flights in the 1970s.
1977	Commodore Pet	First all-in-one home computer with 4–8 K RAM, a 20 x 25 character display, and built-in cassette for data storage. It used the Mostek 6502 processor. It cost \$795 (\$2800 in 2008 currency adjusted for inflation).
1977	Apple II computer	Preceded by the Apple I in 1976, this became Apple's highly successful home computer. It cost \$1298 with 4 K RAM and \$2638 with 48K (\$4680 and \$9509, respectively, in 2008 currency).

continued

Table 1-1 *Continued*

Year	Computer	Event
1977	Radio Shack TRS-80	One of the first mass-produced home computers. It cost $600 ($2030 in 2008 currency).
1978	Intel 8086	Intel's first 16-bit microcontroller: 29,000 transistors, 4.77 MHz clock.
1978	Motorola 6801	First microcontroller: 3500 transistors with 2 MHz clock. It was the first integration of an 8-bit CPU with 128 bytes of RAM, 2 Kbyte of ROM, a 16-bit timer, and serial I/O interface.
1978	First EEPROM	Intel 2816: 2 Kbyte.
1979	Motorola 68000	First 32-bit microprocessor: 68,000 transistors, 8 MHz clock. It had 32-bit registers but 16-bit internal and external data bus and 24-bit address bus.
1980	BELLMAC-32A	First single-chip, 32-bit microprocessor at AT&T Bell Labs; 146,000 transistors.
1980	Intel 8087	Math coprocessor to do floating point arithmetic.
1981	IBM Personal Computer introduced	Intel 8088 with 4.7 MHz clock, ROM BASIC, up to 640K RAM, CGA display adapter, and cassette. A 160 Kbyte floppy was optional. Its $3000 cost in 1981 is equivalent to $7400 in 2008.
1981	iAPX432	Intel's first 32-bit microprocessor. Three chips with a total of 200,000 transistors. It had an 8 MHz clock.
1981	Osborne I	First commercially successful portable computer. It weighed 23.5 pounds and had the CP/M II operating system, a 5-inch display, 64K memory, and 5.25-inch floppy disk. It cost $1795 ($4460 in 2008 currency).
1982	First RISC processor	Reduced instruction set computer produced by the RISC Project at the University of California at Berkeley; 44,500 transistors.
1982	Intel 80286	16-bit microprocessor: 134,000 transistors, 6 MHz clock.
1983	Compaq Portable	First IBM PC compatible portable computer. It cost $3950 ($8400 in 2008 currency) and weighed 28 pounds.
1984	Flash EEPROM developed	Toshiba.
1984	First Apple Macintosh computer	It used an 8 MHz Motorola 68000 microprocessor, 128K RAM, and a 400 Kbyte 3.5-inch floppy. It cost $2495 ($5130 in 2008 currency).
1984	Motorola 68020	32-bit version of the 68000 microprocessor fabricated in CMOS: 190,000 transistors and 16 MHz clock.
1985	Intel 80386	32-bit microprocessor: 275,000 transistors, 16 MHz clock.
1989	Intel 80486	32-bit microprocessor: 1.2 million transistors, 25 MHz clock.
1990	FCC Part 15, Subpart B	Rules governing radiofrequency emissions for electronic equipment including personal computers. These federal rules require testing and certification of electronic equipment.
1992	IBM PowerPC	First single-chip PowerPC reduced instruction set computer; 32 bits 2.8 million transistors, 68 MHz clock.
1996	DEC Alpha 21064	Digital Equipment Corporation, 64-bit pipelined processor, 9.7 million transistors, 500 MHz clock.
2000	Intel Pentium IV	64-bit microprocessor: 42 million transistors, 1.4 GHz clock.
2005	AMD Athlon 64	64-bit microprocessor: 200 million transistors, 2.6 GHz clock.
2008	AMD Phenom	64-bit microprocessor: 450 million transistors, 3 GHz clock.

1.2 Moore's Law

Table 1-1 shows a remarkable, exponential growth rate in the size and speed of the integrated circuits used in microprocessors and microcontrollers. In 1965 Intel cofounder Gordon Moore observed this phenomenon and predicted that the growth would continue doubling every 18 to 24 months. Although some observers claim this is a self-fulfilling prophecy because

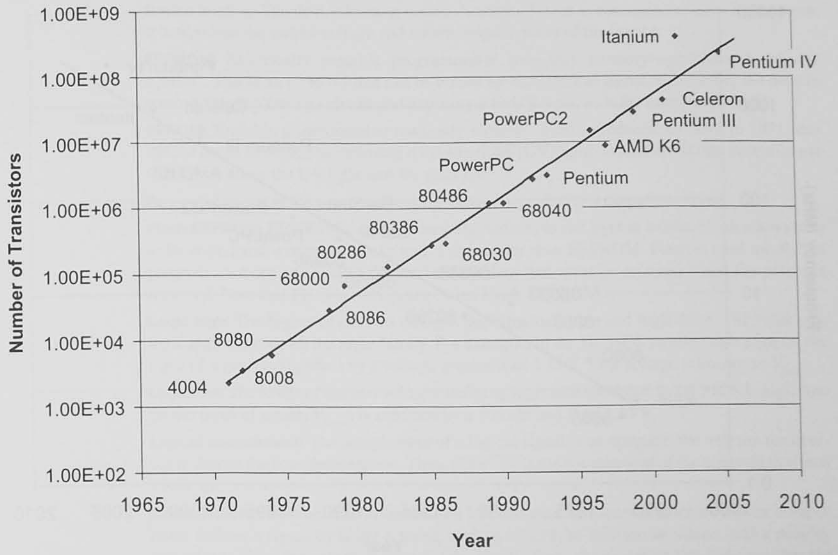


Figure 1-3 Growth in number of transistors in microprocessors from late 1960s to first decade of the twenty-first century.

manufacturers concentrate on improving their technology, Moore's now four-decade-old observation has continued to be true, as shown in Figures 1-3 and 1-4.

1.3 Microcontrollers

A *microcontroller* is a computer with *CPU, memory,* and *I/O* in one integrated circuit chip.

This text primarily is about using computers in applications where the system is dedicated to performing a single task or a single group of tasks. These are called embedded applications, and examples are found almost everywhere in products from microwave ovens and toasters to automobiles. These are often *control* applications and make use of microcontrollers. A *microcontroller* is a microcomputer with its memory and I/O integrated into a single chip. In 1991 the chip manufacturers delivered over 750 million 8-bit microcontrollers; by 2004 the industry's annual total was 6.8 billion microcontroller units.[2]

[2] http://www.instat.com/press.asp?ID=1445&sku=IN0502457SI

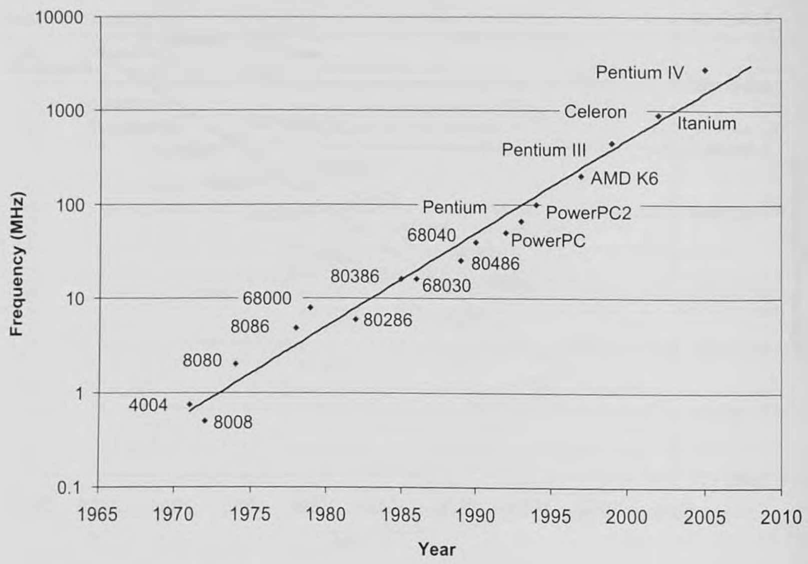


Figure 1-4 Improvements in microprocessor clock frequency for the same period.

1.4 Some Basic Definitions

Throughout this text we use the following digital logic terminology.

Active high: Used to define a signal whose assertion level is logic high.

Active low: This term defines a signal whose assertion level is logic low. For example, the signal `READ_L` is asserted low. Although many data sheets and schematic diagrams make use of an overbar or some other notation, in this text we will denote active-low signals by adding the "`_L`" suffix to the signal name.

Assembly/Compile time: The time at which our programs are assembled or compiled. Quantities known at this time can be saved as constants in program memory (ROM). In an embedded system, variable data must not be initialized at assembly/compile time.

Assert: Logic signals, particularly signals that control a part of the system, are asserted when the control, or action named by the signal, is being done. A signal may be low or high when it is asserted. For example, the signal `WRITE` indicates assertion when the signal is logic high.

Byte: A byte is 8 bits.

Device loading: The device loading is an indication of what is connected to a device's output. It determines the output voltage and current requirements of the device.

EEPROM: Electrically erasable programmable read-only memory—pronounced "*double e prom*". This is an EPROM that can be erased by an electrical signal, eliminating the need to remove the chip from its circuit and exposing it to UV light, as is the case for EPROM.

EPROM: Erasable programmable read-only memory. First introduced by Intel in 1971, this PROM could be erased by exposing it to ultraviolet (UV) light. Erasable PROMs have a quartz window to allow the UV light into the package.

Fan-out: Fan-out is the number of similar devices one device's output can drive.

Flash EEPROM: EEPROM may be erased and written to one byte at a time. Flash allows data to be erased and written in blocks and is thus faster than EEPROM. Flash is used mostly for program memory and EEPROM for variable data that must be retained when the power is removed. Note that Flash is sometimes called Flash EEPROM.

Logic high: The higher of the two voltages defining logic true and logic false. The value of a logic high depends on the logic family. For example, in the HCMOS family, logic high (at the input of a gate) is signified by a voltage greater than 3.15 V. This voltage is known as V_{ihmin}.

Logic low: The lower of the two voltages defining logic true and false. In HCMOS, a logic low (at the input of a gate, V_{ilmax}) is signified by a voltage less than 1.35 V.

Logical complement: The complement of a logical signal is an operator. We will use the overbar to donate the complementation. Thus, $\overline{\texttt{PUMP\_ON}}$ is the complement of the active-high signal `PUMP_ON`.

Mixed-polarity notation: The notation used by most manufacturers of microcomputer components defines a signal by using a name, such as `WRITE`, to indicate an action, and a polarity indicator to show the assertion level for the signal. Thus, the signal `WRITE` indicates that the CPU is doing a write operation when the signal is high. `READ_L` denotes that a read operation is going on when the signal is low.

Nibble: A nibble is 4 bits. There are two nibbles for each byte.

OTP EPROM: One-time-programmable EPROM. This is an EPROM without the quartz window; thus it cannot be erased after it has been programmed.

Positive and negative edge trigger: Data latches may operate on a level or edge-triggered basis. There are positive (rising) and negative (falling) edge-triggered devices.

PROM: Programmable read-only memory. Memory that can be programmed by the user instead of at the factory, as must be done for ROM.

RAM: Random access memory. This memory can be read from and written to and is used in the microcontroller for variable data storage. The memory contents are lost when the power is removed. Therefore the memory is said to be volatile.

ROM: Read-only memory. The contents of this memory is programmed once, at the time of manufacture, and is nonvolatile. That is, the memory contents persist when the power is removed. ROM is used in microcontrollers for program storage.

Run time: This is when our program executes. Any variable data with initial values must be initialized at run time.

Tristate or three-state: A logic signal that can neither source nor sink current. It presents a high impedance load to any other logic device to which it is connected.

Word: A word is 16 bits.

Table 1-2 Notation

0x	Hexadecimal numbers are denoted by a leading 0x (e.g., 0xFFFF is the hexadecimal number FFFF).
	When two memory locations are to be identified, the starting and ending addresses are given as 0xFFFE:FFFF.
$	Hexadecimal numbers in Freescale assembly language examples use a $ to denote a hexadecimal number. $0F = 15.
%	Binary numbers are denoted by a leading %. For example, 0xF may be written %1111.
@	A base-8 or octal number is preceded by @. Thus 0xF = @17.
	Base 10 is the default base; unlike hexadecimal, binary or octal, it has no base indicator. Thus 0xF = 15.
0b	In C programs, the 0b prefix is used to signify a binary number.
x	An "x" indicates a don't-care bit—that is, the bit may be zero or one.
*	The "*" indicates a pointer in a C program.
_L	A signal whose assertion level is low is followed by "_L."

1.5 Notation

Throughout this text, the notation shown in Table 1-2 is used.

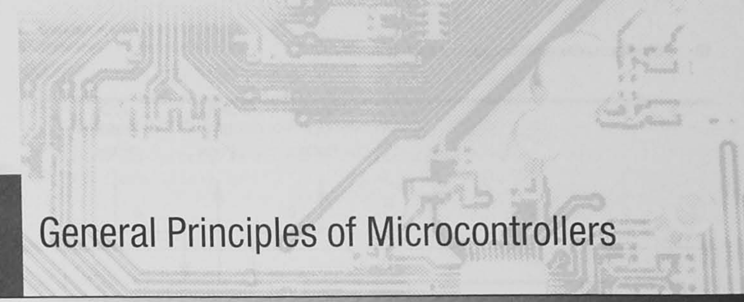
1.6 Study Plan

The designs of embedded application systems and other more general-purpose computers are very similar. Our goal for this course is not to make you an expert in using a specific processor, but to give you the knowledge and tools to be able to effectively apply any processor in any application. We will do that by first studying the general principles necessary to understand each part of the system. You may then turn to the user's manual for a specific processor and be able to more easily understand the information there and apply it in an application.

The basic operation of a stored-program, general-purpose computer is to be studied first. You'll learn about registers, the arithmetic and logic unit, and how a computer works. Because much of your work in an introductory microprocessor/microcontroller course is likely to be learning the language and programming exercises, we introduce you to structured program design in Chapter 3. Designing software before writing it is vital in developing debuggable application software. We will guide you through an introduction to the central processor unit and how it addresses memory in Chapters 4 and 5 and introduce assembly language programming in Chapter 6. You will need to study your own processor in parallel while reading these chapters.

Many embedded applications are written in C, which you may have learned in another programming class. A program written in C for an embedded application, however, has some significant differences from one written for a desktop computer. Chapter 7 will help you learn about these differences. Chapter 8 discusses debugging techniques helpful for assembly and C language programs.

Chapters 9 through 15 cover the basics of parallel and serial I/O, interrupts, memory, analog I/O, timers, and interfacing techniques for single-chip microcontrollers. Chapter 16 touches on real-time operating systems.



2 General Principles of Microcontrollers

Objectives

This chapter introduces the principles of a stored program computer and shows how we develop the software for an embedded microcontroller system. The material should enable you to understand the hardware of a typical system. You will see the importance of the instruction *fetch*, how the sequence controller works, and how to determine system timing. You will understand how memory operates and how it affects the design of the computer. We also consider the software needed and introduce the idea of a tool set to produce the code that ultimately resides in the microcontroller's read-only memory.

2.1 Introduction

In this chapter we will investigate the operation of a typical microprocessor or microcontroller. Our goal is to have you see that a computer is *not* a mysterious box but, rather, a collection of basic digital logic components that you could design. By the end of this chapter you will appreciate that a computer works in a predictable way and that you have complete and absolute control over what it does at all times.

2.2 A Typical Microcontroller

A typical microcontroller is shown in Figure 2-1. It consists of the following elements

- A *central processor unit (CPU)*, that contains registers, an arithmetic and logic unit (*ALU*), and a sequence controller to control all activities of the microcontroller.
- *Read-only memory (ROM)*, to hold our program and any constant data. Modern microcontrollers have reprogrammable types of read-only memory such as Flash

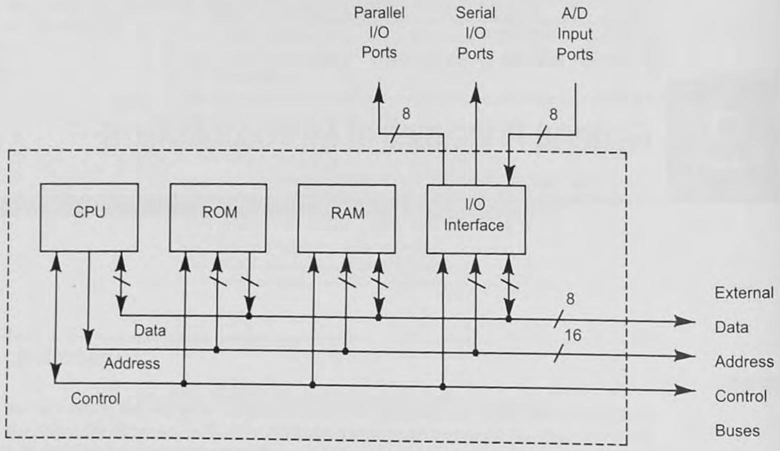


Figure 2-1 Typical microcontroller.

memory, which is a particular type of electrically erasable programmable read-only memory (EEPROM).

- *Random access memory (RAM)*, to store variable data.
- An *input/output (I/O) interface* to connect the microcontroller to the real world. The I/O interface in most microcontrollers contains other useful functions such as timers, pulse-width modulators, and other special I/O functions.
- Connecting these blocks are three buses: the *data bus*, the *address bus*, and the *control bus*. Often these buses are available outside the microcontroller to allow additional memory and I/O to be used.

The Program

Any program in an embedded system, such as the famous C program that prints the message "Hello World!" as shown in Example 2-1, must be in the memory (normally ROM). This C program, however, hides some of the important details of what is really in the memory of the microcontroller. Our microcontroller has instructions, called machine or assembly language instructions. The instructions in our programs are converted to binary codes that *instruct* the microcontroller what to do to *execute* the program. Example 2-2 shows an equivalent Hello World! program written in the assembly language of a typical microcontroller, such as a Freescale HCS12. Other microcontrollers will have similarly encoded instructions. No matter what language you use to write your programs, they will all be converted to the particular microcontroller's instruction set to be placed into the program memory.

Example 2-1 C Program to Print Hello World!

```
/* Example Program to print "Hello World!" */
#include <stdio.h>

void main(void) {
  printf("Hello World");
}
```

Example 2-2 Assembly Language Hello World! Program

```
; Example program to print
; "Hello World"
; Constant equates
CR:     EQU  0x0d   ; Carriage return
LF:     EQU  0x0a   ; Line feed
EOS:    EQU  0      ; End of string
; Memory map equates
PROG:   EQU  0x8000 ; Flash memory
STACK:  EQU  0x0a00 ; Stack pointer
        ORG  PROG   ; Locate program
Entry:
; Initialize stack pointer
        lds  #STACK
loop:
; Print Hello World! string
        ldd  #HELLO
        jsr  printf
; Do it forever
        bra  loop
; Define the string to print
HELLO:  DC.B 'Hello World!',CR, LF, EOThe Picocontroller
```

2.3 The Picocontroller

To understand how computer instructions work, let us consider the design of a very simpl microcontroller. It is so simple it can be called a picocontroller.[1]

[1] One picocontroller = 10^{-6} microcontroller.

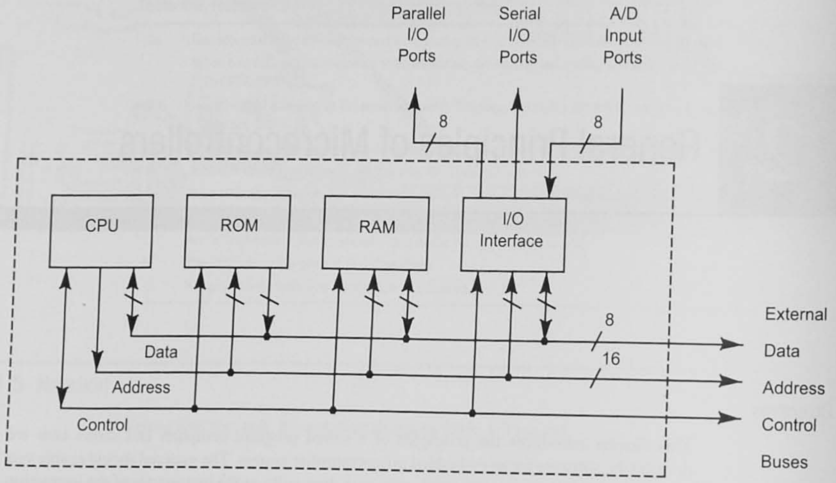


Figure 2-1 Typical microcontroller.

memory, which is a particular type of electrically erasable programmable read-only memory (EEPROM).

- *Random access memory (RAM),* to store variable data.
- An *input/output (I/O) interface* to connect the microcontroller to the real world. The I/O interface in most microcontrollers contains other useful functions such as timers, pulse-width modulators, and other special I/O functions.
- Connecting these blocks are three buses: the *data bus,* the *address bus,* and the *control bus.* Often these buses are available outside the microcontroller to allow additional memory and I/O to be used.

The Program

Any program in an embedded system, such as the famous C program that prints the message "Hello World!" as shown in Example 2-1, must be in the memory (normally ROM). This C program, however, hides some of the important details of what is really in the memory of the microcontroller. Our microcontroller has instructions, called machine or assembly language instructions. The instructions in our programs are converted to binary codes that *instruct* the microcontroller what to do to *execute* the program. Example 2-2 shows an equivalent Hello World! program written in the assembly language of a typical microcontroller, such as a Freescale HCS12. Other microcontrollers will have similarly encoded instructions. No matter what language you use to write your programs, they will all be converted to the particular microcontroller's instruction set to be placed into the program memory.

Example 2-1 C Program to Print Hello World!

```
/* Example Program to print "Hello World!" */
#include <stdio.h>

void main(void) {
  printf("Hello World");
}
```

Example 2-2 Assembly Language Hello World! Program

```
; Example program to print
; "Hello World"
; Constant equates
CR:     EQU  0x0d   ; Carriage return
LF:     EQU  0x0a   ; Line feed
EOS:    EQU  0      ; End of string
; Memory map equates
PROG:   EQU  0x8000 ; Flash memory
STACK:  EQU  0x0a00 ; Stack pointer
        ORG  PROG   ; Locate program
Entry:
; Initialize stack pointer
        lds  #STACK
loop:
; Print Hello World! string
        ldd  #HELLO
        jsr  printf
; Do it forever
        bra  loop
; Define the string to print
HELLO:  DC.B 'Hello World!',CR, LF, EOThe Picocontroller
```

2.3 The Picocontroller

To understand how computer instructions work, let us consider the design of a very simple microcontroller. It is so simple it can be called a picocontroller.[1]

[1] One picocontroller = 10^{-6} microcontroller.

Table 2-1 Picocontroller Operations and Opcodes

Operation	Operation Code
ADD	00
SUB	01
IN	10
OUT	11

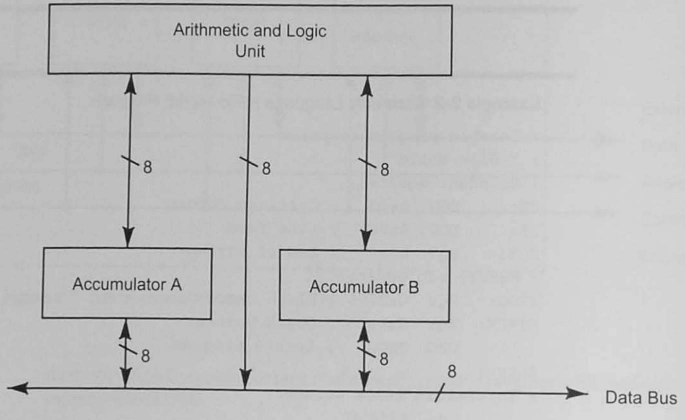


Figure 2-2 Accumulator registers (A and B), arithmetic logic unit (ALU), and a data bus.

Computer Operation Codes

The first step in the design of a computer is to define the set of executable operations. Our simple computer is to be capable only of inputting and outputting 8-bit binary numbers and adding or subtracting them. The input, output, adding, and subtracting capabilities are called *operations*, and we encode them by using *operation codes (opcodes)*. Because computers are digital devices, all information is encoded in binary—1s and 0s. If there are four operations, 2 bits are needed to provide a unique code for each. Table 2-1 shows the codes that are selected.

Basic Computer Hardware

Hardware for Addition and Subtraction

An *accumulator* is a register that may hold one operand for an ALU operation and may be used for the answer, as well.

Let us look at the hardware required to add or subtract two 8-bit binary numbers. These operations require two *operands*—the two binary numbers that are added or subtracted. For the adder or the subtracter to work, these binary numbers are held in registers while the addition or subtraction is being carried out. Registers are arrays of memory

elements, usually flip-flops, which may be loaded with binary values. Two registers—called the A and B registers—hold the operands. In your logic class you probably learned how to design a ripple-carry full-adder to add 8-bit numbers and produce an 8-bit result plus a carry. Similar hardware could be designed to subtract two numbers. As the design of this computer progresses, we will probably want to add more capabilities, perhaps logic operations like AND and OR. The hardware for these arithmetic and logic operations can be placed into a black box called the *arithmetic and logic unit (ALU)*. The specific hardware within the ALU is not a concern at this time; it is sufficient to know that hardware can be designed to do addition and subtraction. The design at this stage is shown in Figure 2-2, where arrows show that the numbers to be added or subtracted come from a *data bus* and flow from the registers to the ALU. The answer flows from the ALU back to the data bus. The registers are called *accumulators* because they can accumulate answers.

At this stage of the design some details of using registers can be ignored. For example, a register needs a clock signal, and there is a carry signal that is produced by the adder circuit in the ALU. These design details can be postponed for now. See Example 2-3 and 2-4.

Example 2-3 8-Bit Register

Show how to use eight D-type latches to construct an 8-bit register.

Solution

See Figure 2-3.

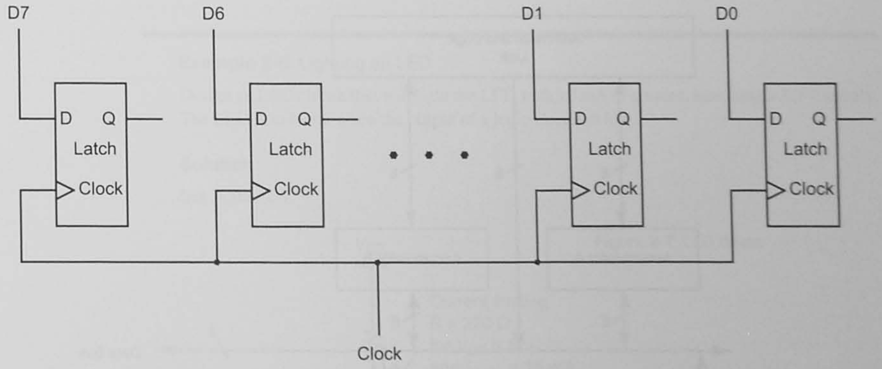


Figure 2-3 An 8-bit register.

Example 2-4 8-Bit Ripple-Carry Adder

Show how to make an 8-bit ripple-carry adder by using seven full adders and one half-adder.

Solution

See Figure 2-4.

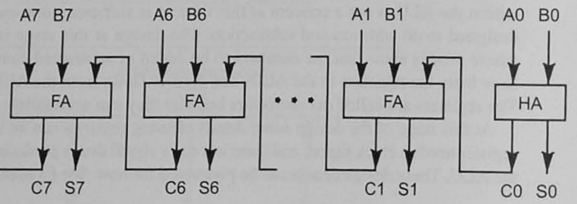


Figure 2-4 An 8-bit ripple-carry adder.

Input and Output Hardware

At this point, there are registers to hold numbers and an ALU to add or subtract them. There must be a *source* for the numbers and a *destination* for the answer. Let's use a set of eight switches to enter the numbers and eight light-emitting diodes (LEDs) to display the result.

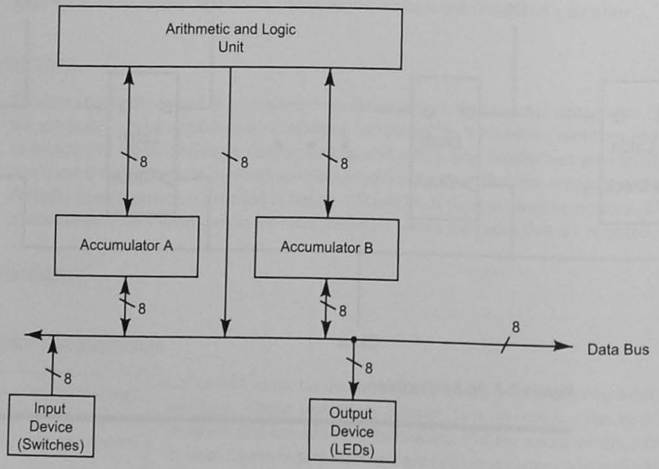


Figure 2-5 Adding input and output devices to the registers and ALU.

Figure 2-5 shows an input device, the switches, and an output device, the LEDs, added to the registers and ALU. See Examples 2-5 and 2-6.

Example 2-5 Binary Switch

Show how to use a switch to produce a logic high or logic low for the two positions of the switch.

Solution

See Figure 2-6.

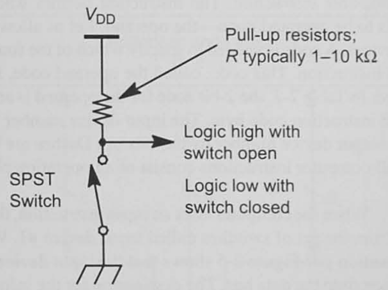

V_{DD}
Pull-up resistors; R typically 1–10 kΩ
Logic high with switch open
Logic low with switch closed
SPST Switch

Figure 2-6 Binary input switch.

Example 2-6 Lighting an LED

Design an LED circuit that will light the LED with 10 mA of current, assuming a 3.3 V supply. The LED is to be on when the output of a logic circuit is low.

Solution

See Figure 2-7.

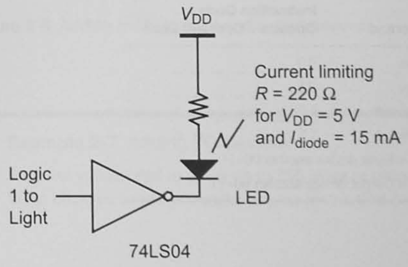

V_{DD}
Current limiting $R = 220\ \Omega$ for $V_{DD} = 5$ V and $I_{diode} = 15$ mA
Logic 1 to Light
LED
74LS04

Figure 2-7 LED driver.

Operand Codes

The hardware to input numbers and display the results is adequate for now, but let's look ahead and design in some additional capabilities. Most computers have many sources of information. For example, a binary number may be input from an analog-to-digital converter, or the result of the addition may be transferred to a digital-to-analog converter instead of the display LEDs. To do that, we must specify which of several devices the computer must input from or output to. A design decision must be made. How can the computer input information from more than one source? There are two choices. There could be separate operation codes to do the separate operations. For example, we could define two new input operations, `IN1` and `IN2`, where `IN1` inputs information from the switches and `IN2` from an analog-to-digital converter. Another choice is to include an *operand* with the *operation*. The combination of an operation and an operand is called a *computer instruction*. The instruction defines what is to be done—the operation—and what is to be operated upon—the operand. Let us allow up to four of each of the input and output devices. A code is needed to specify which of the four devices is to be used for the input or output instruction. This code, called the operand code, is added to the operations previously defined. In Table 2-2, the 2-bit code for the operand is arbitrarily placed in the last two bits of an 8-bit instruction code byte. The input device number is to be encoded with the two bits *ii* and the output device number by the bits `oo`. Dashes are bits that haven't been assigned yet. Almost all computer instructions consist of an operation plus one or more operands.

A computer *instruction* is an *operation* plus and *operand*.

When the computer does an input instruction, the information comes from the set of switches called input device #1. Where does the information go? Figure 2-5 shows that the input device *sources* its information onto the data bus. The *destination* for the information can be either Accumulator A or Accumulator B. To allow the programmer to choose one or the other, we can simply add a destination operand code for the input operation and, using similar arguments, a source operand to be able to choose which accumulator contains data to be output to the output device. Table 2-3 shows our growing instruction set and Figure 2-8 gives an updated diagram showing up to four potential input and output devices. See Example 2-7.

All data transfer operations have a *source* operand and a *destination* operand.

Table 2-2 Adding Operands to the IN and OUT Operations

Operation	Operand	Instruction Code = Opcode + Operand Code
`ADD`	None	`00 _ _ _ _ _ _`
`SUB`	None	`01 _ _ _ _ _ _`
`IN`	Device #	`10 _ _ _ _ i i`
`OUT`	Device #	`11 _ _ _ _ o o`
	`ii` = Input device number 00–11	
	`oo` = Output device number 00–11	

Table 2-3 Adding Source and Destination Operands to the IN and OUT Operations

Operation	Operand	Instruction Code = Opcode + Operand Code
`ADD`	None	`00 _ _ _ _ _ _`
`SUB`	None	`01 _ _ _ _ _ _`
`IN`	Device # destination register	`10 _ _ i i r r`
`OUT`	Source register, device#	`11 _ _ r r o o`
	`ii` = Input device number 00–11	
	`oo` = Output device number 00–11	
	`rr` = Register address: A = 00, B = 01	

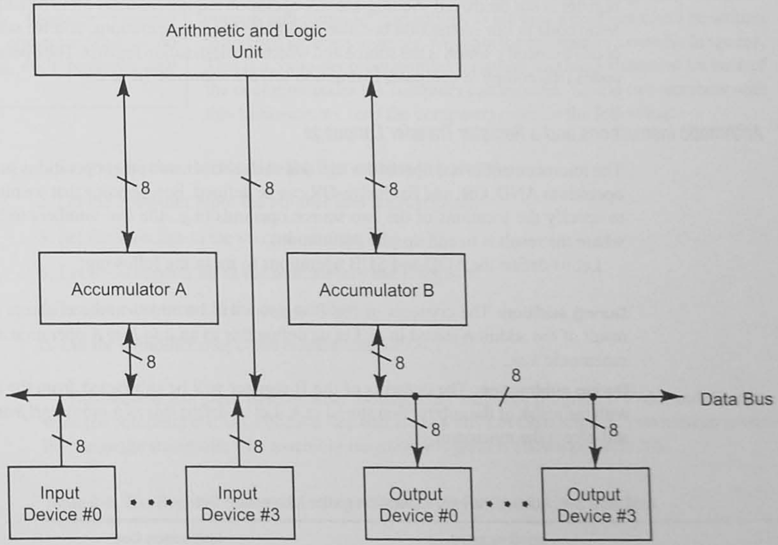


Figure 2-8 Adding multiple input and output devices.

Example 2-7 Adding I/O Devices

Suppose you needed to allow up to 256 input or output devices in your picocontroller design. What changes would you make to your instruction codes?

Solution

For 256 input or output devices, there must be an 8-bit I/O address. You must add another byte to the operation code.

The Move Operation

Another very useful operation will give the microcontroller the capability to transfer data between the two accumulators. This allows us to use one for temporary storage while using the other. Microcontroller instruction sets call these *move* (`MOV`), *transfer* (`TFR`), or *load* (`LD`) operations. Let's define a `MOV` operation to move information from one register to another. The `MOV` operation *copies* the data from the source to the destination; the information in the source register is not destroyed. Adding another operation requires another bit to be added to the operation code in our instruction because now there are five operations, and we will add operand codes to specify which is the source and which the destination register. The operations and their codes (the complete computer instruction set) are shown in Table 2-4.

Arithmetic Instructions and a Register Transfer Language

The microcontroller has operations that add and subtract, and other operations such as the logic operations AND, OR, and Exclusive-OR can be defined. Before doing that we must decide how to specify the locations of the two source operands (e.g., the two numbers to be added) and where the result is to end up (the destination).

Let us define the ADD and SUB operations to mean the following:

During addition: The contents of the B register will be added to the contents of A with the result of the addition stored in A. Let us define this as an *add B to A* operation and give it the mnemonic `ABA`.

During subtraction: The contents of the B register will be subtracted from the contents of A with the result of the subtraction stored in A. Let us define this as a *subtract B from A* operation and give it the mnemonic `SBA`.

Table 2-4 Adding the `MOV` Operation to the Instruction Set

Operation	Operand	Instruction Code = Opcode + Operand Code
`ADD`	None	`001 _ _ _ _ _`
`SUB`	None	`011 _ _ _ _ _`
`IN`	Device #, destination register	`101 _ i i d d`
`OUT`	Source register, device #	`111 _ s s o o`
`MOV`	Source register, destination register	`010 _ s s d d`

`ii` = Input device number 00–11
`00` = Output device number 00–11
`ss` = Source register address: A = 00, B = 01
`dd` = Destination register address: A = 00, B = 01

Although these descriptions define what the computer instruction does, we need a shorthand way to succinctly describe them. A *register transfer language*, or notation, is commonly used. For example, we can describe the ABA instruction by showing

$$A + B \rightarrow A$$

which means that the contents of A are replaced by the sum of the present contents of A and B. Some register transfer languages reverse the order to show the replacement operation

$$A \leftarrow A + B$$

We can now use the register transfer language to describe the result of each instruction (Table 2-5).

Adding Two Numbers

An *assembly language program* instructs the computer what to do by specifying each operation and operand.

Although the design is far from complete, there are enough hardware components and computer instructions to see how a program could be written to add two numbers together. We write the program in *assembly language*. This is a computer programming language that has a statement for each of the operation codes the computer can execute. To add two numbers with this hardware, we (and the computer) must do the following:

1. Set the switches (by hand) to the first number to be input.
2. Let the computer input the number into the A register.
3. Set the switches to the second number.
4. Let the computer input the number into the B register.
5. Let the computer add the two numbers.
6. Let the computer output the result to the LEDs.

This sequence of steps defines what the assembly language program is to do, and therefore what the computer is to do. For each step that starts with "Let the computer," we need an assembly language statement. The assembly language program is shown in Table 2-6.

Table 2-5 The Register Transfer Language Shows How Each Instruction Operates

Operation	Operand	Instruction Code = Opcode + Operand Code	Register Transfer
`ABA`	None	`001 _ _ _ _ _`	$A + B \rightarrow A$
`SBA`	None	`011 _ _ _ _ _`	$A - B \rightarrow A$
`IN`	Device #, destination register	`101 _ i i d d`	ii → dd
`OUT`	Source register, device #	`111 _ s s o o`	ss → oo
`MOV`	Source register, destination register	`010 _ s s d d`	ss → dd

`ii` = Input device number 00–11
`00` = Output device number 00–11
`ss` = Source register address: A = 00, B = 01
`dd` = Destination register address: A = 00, B = 01

Table 2-6 A Program to Input Two Numbers, Add Them Together, and Display the Result

Step	Operation Field	Operand Field	Comment Field
2	`IN`	`1,A`	; Switches → A. The contents of A are replaced by the contents of the switch bank.
4	`IN`	`1,B`	; Switches → B. The contents of B are replaced by the contents of the switch bank.
5	`ABA`		; A + B → A. The contents of A are replaced by the sum of the contents of A and B.
6	`OUT`	`A,2`	; A → LEDs. The contents of the LEDs are replaced by the contents of A.

A *mnemonic* is the English-like shorthand for an operation.

The format shown in Table 2-6 is typical of an assembly language program, and there are several fields on each line. First is the *operation field* where the *mnemonic*, an English-like code for the computer operation, is written. Following that is the *operand field*, where the operand, if there is one, is written. A *comment field* may follow the operand field.[2] Usually comments explain the design or the purpose of the program. Here we have chosen to show a shorthand notation that many manufacturers use to state exactly what the instruction is to do. For example, in line 2, the operation mnemonic is `IN`, the operands are device #1 (the bank of eight switches) and the destination register (A), and the comment shows that the contents of the A register are to be replaced by the contents of the switches.[3]

The Program in Memory

As you know from your own experience with computers, programs go into memory. You can envision the memory as an array of flip-flops that store data. In a typical memory there are 8 bits (a byte), D7...D0, in each memory location, and there are 65,536 (64 kilobytes)[4] locations. To access one specific location in these 64K locations, a 16-bit address, A15...A0, must be supplied. When this is done and when certain control signals are activated, information can be read from the program memory.

The computer memory contains binary information or data. Therefore, the program shown in Table 2-6 must be *encoded*, using the codes defined in Table 2-5. This is called *assembling* a program, a process that turns the instructions into the 1s and 0s that go into the computer memory. *All* computer programs, no matter the language in which they are written, must be assembled, or converted, into binary words called the *machine code*. The result of this is shown in Table 2-7.

The first location in the memory is *address zero*. For each memory location you can look at the binary codes and, referring to Table 2-5, decode them to find out what the computer is going to do. Before designing hardware to do just this, let us return to our more realistic Hello World! program and see how it looks in a realistic microcontroller's memory.

[2] There is another field to the left of the operation field not shown in this example. This is the label field, and we will see how to use this later, in Chapter 5, and when you learn how to use an assembler.

[3] Chapter 6 will show that comments that merely tell what the instruction is doing are not very useful. Comments related to the design of the program showing why the instruction is there are far more valuable.

[4] A kilobyte (Kbyte) is $2^{10} = 1024$ bytes.

Table 2-7 How the Program Looks in Memory

Memory Location Address	Contents (Machine Code)	Assembly Program Statement
0:	`1010 0100`	`IN 1,A`
1:	`1010 0101`	`IN 1,B`
2:	`0010 0000`	`ABA`
3:	`1110 0010`	`OUT A,2`

Table 2-8 Contents of Memory for Hello World!

Memory Addresses	Instruction Code Bytes	Instruction
`8000 - 8002`	`CF 0A 00`	`LDS #0x0A00`
`8003 - 8005`	`CC 80 0B`	`LDD #0x800B`
`8006 - 8008`	`16 80 18`	`JSR 0x8018`
`8009 - 800A`	`20 F8`	`BRA 0x8003`
`800B - 8017`	`48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0D 0A 00`	`Constant data for the Hello World! string`

2.4 The Microcontroller's Memory

The assembly language program in Example 2-2 is assembled and loaded into the microcontroller's memory. It looks something like that shown in Table 2-8. This example illustrates that even though you may write a program in a high-level language like C, it is converted to bytes representing the *operation* that must be done and the *operands* that are being operated upon. The memory addresses shown correspond to the ROM in the embedded system. Note that all addresses and instruction code bytes are in hexadecimal. The right-hand column shows each instruction in the program in assembly language. Don't worry about what these instructions are at this stage; you will learn your particular microcontroller's instruction set later.

Look closely at the Instruction Code Bytes column in Table 2-8. In each case, the first byte in each line (CF, CC, 16, etc.) is a unique code for each *operation* to be *executed* by the microcontroller. For example, *CF* is a code for the LDS (*immediate load stack pointer register*) operation.[5] This is the *opcode* byte.[6] The following two bytes (0A 00) are the bytes for the *operands* for this operation. A computer *instruction* is the combination of an operation (what the computer is to do) and zero, one, or more operands (what the computer is going to do it to). For this instruction, the microcontroller will load, or initialize, the stack pointer register with the value 0x0A00.

Constant data also may be stored in the ROM. The data for this program is the string "Hello World!" that is to be printed. Data constants may be defined in an assembly language program like you have done in other programming languages. The line "`HELLO: DC.B`" in Example 2-2 shows how this is done in the assembly program. In Table 2-8 the memory locations 0x800B–0x8017 contain the *constant* data used by the program. These bytes are the ASCII

[5] These are Freescale HCS12 instructions. Your own microcontroller will have different operations and opcode bytes.

[6] Some computer operations may need to be specified by more than one byte.

codes for the characters in the message that will be printed on the screen by a *printf* routine. See Examples 2-8 through 2-10.

Example 2-8 Opcodes

For each of the instructions in Table 2-8, give the memory locations and the hexadecimal value for each opcode.

Solution

Memory	Opode Byte
8000	CF
8003	CC
8006	16
8009	20

Example 2-9 Operand Codes

For each of the instructions in Table 2-8, give the memory locations and the hexadecimal value for each operand code.

Solution

Memory	Operand Code Bytes
8001, 8002	0A, 00
0804, 0805	80, 0B
0807, 0808	80, 18
800A	F5

Example 2-10 Constant Bytes

For the program in Table 2-8, give the memory locations and the hexadecimal value for the constant data bytes in the string "Hello World!"

Solution

Memory	Constant Data Byte
800B	48
800C	65
800D	6C
800E	6C
800F	6F

Memory	Constant Data Byte
8010	20
8011	57
8012	6F
8013	72
8014	6C
8015	64
8016	21
8017	0D
8018	0A
8019	00

The Memory Map

A *memory map* shows you where each kind of memory, or no memory, is located.

A *memory map*, which shows what memory addresses are used for what type of memory, is used to show the memory organization in a computer. A typical microcontroller's memory map may contain RAM, EEPROM, Flash, and even spaces without memory, as shown in Figure 2-9. Notice in Figure 2-9 that part of the memory space is used for I/O control registers, and there are some memory addresses that contain no memory.

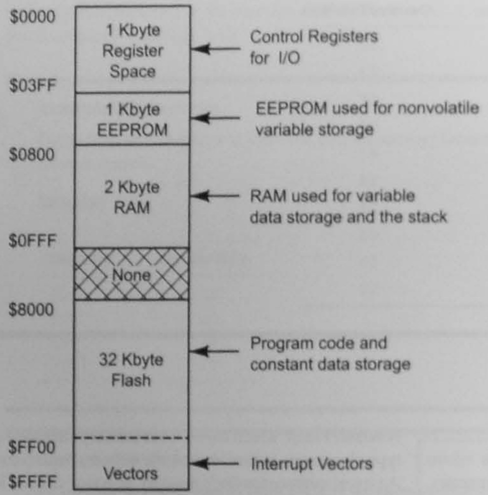
There are two types of memory shown in Figure 2-9. The contents of the read-only memory, ROM, are not lost when the power is removed from the microcontroller. This is *nonvolatile* memory. The various types of ROM include factory programmed (normally just called ROM), and field-programmable ROM, or PROM. PROM comes in three varieties including UV erasable EPROM, electrically erasable EEPROM, and Flash (sometimes called Flash EEPROM). EEPROM and Flash are very similar in that the memory can be erased and then reprogrammed without removing it from the circuit to place in a UV PROM erasure as must be done for the UV PROM devices. Flash and EEPROM are different in that EEPROM can be erased and programmed a byte at a time, and thus can be used to store program variables that must remain after the power has been cycled off and then on. Flash EEPROM uses similar integrated circuit technology but is organized so that it can be erased and then programmed in large blocks. Flash is used to store our programs because this organization allows for faster erasing and programming.

The RAM is *random access memory*. This terminology is somewhat misleading because ROM can be randomly accessed, too. RAM is memory that can be read from and written to, making it useful for program variable data. Most RAM is volatile, and so the program must initialize it after the power has been turned on.

ROM Operation

ROM is *nonvolatile* and is used for program code and constants.

The ROM contains the opcode, operand, and constant data bytes. We will not worry about how they get there at present, leaving that discussion for a later chapter. The microcontroller CPU needs to *fetch*, or read, the opcode and operand bytes from the ROM to be able to execute the program. To see how this is done, let us first consider how the ROM works, and for that matter, the RAM.



$0000
$03FF
$0800
$0FFF
$8000
$FF00
$FFFF
1 Kbyte Register Space
1 Kbyte EEPROM
2 Kbyte RAM
None
32 Kbyte Flash
Vectors
Control Registers for I/O
EEPROM used for nonvolatile variable storage
RAM used for variable data storage and the stack
Program code and constant data storage
Interrupt Vectors

Figure 2-9 A microcontroller's memory map.

Table 2-9 Memory Contents

Memory Address	**Memory Contents (Binary)**	**Memory Contents (Hex)**
8000:	1100 1111	CF
8001:	0000 1010	0A
8002:	0000 0000	00
8003:	1100 1100	CC
8004:	1000 0000	80
8005:	0000 1011	0B
...		...
8018:	0000 1010	0A
8019:	0000 0000	00

The ROM is a sequence of storage locations, each containing a byte of information as shown in Table 2-9[†]. Figure 2-10 shows a read-only memory. The bus labeled *address bus* consists of *address* bits coming from the CPU. The number of bits depends on the number of memory storage locations. For example, a 64 Kbyte memory (65,536 storage locations) must have 16 address bits to specify each location uniquely. The CPU provides this address. The

[†] This is a *byte-wide* memory. Some applications may have 16 or more bits per location.

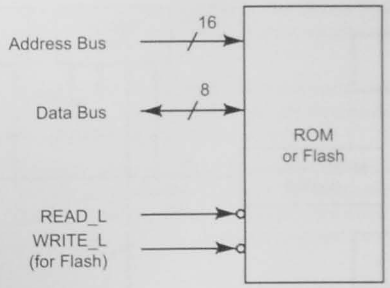


Figure 2-10 Read-only memory.

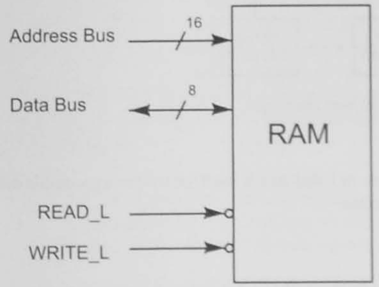


Figure 2-11 Random access memory.

line labeled READ_L is a *read control* signal, and it is also asserted by the CPU. When the memory receives an address and READ_L is asserted, the ROM places the data byte stored in that address onto the *data bus*.

RAM Operation

Figure 2-11 shows the RAM, and it is very similar to the ROM shown in Figure 2-10. In normal operation, the data connection to a RAM is bidirectional. This means that data from the data bus can be written into the selected address in RAM in response to the WRITE_L signal or read from memory when READ_L is asserted. The most notable difference between ROM and RAM is that the data in RAM is *volatile*. If the power to the RAM is turned off at any time, the data will be lost.

There are two uses for RAM in embedded microcontroller systems. The first is for variable data storage. For example, if we have read a value from an analog-to-digital converter and want to save it for later reference, we would write that value into the RAM. The second use is for

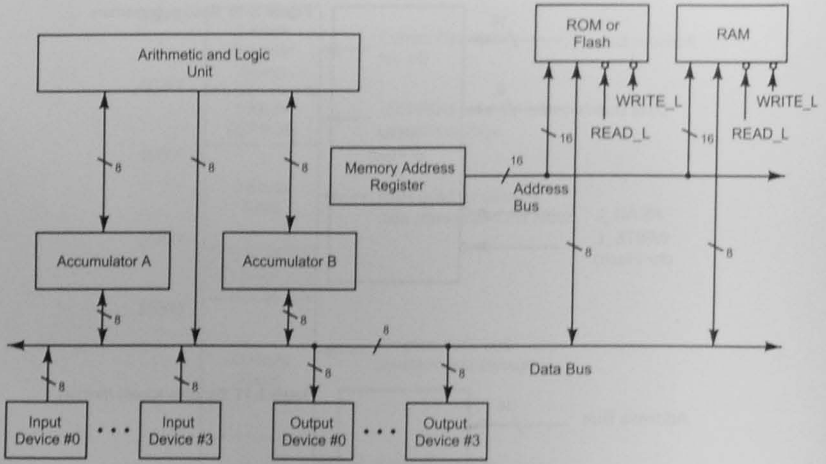


Figure 2-12 The picocontroller with memory.

the stack. The *stack* is an area of RAM that is used for temporary variable data storage and for return addresses for subroutines.

The Picocontroller's Memory

Figure 2-12 shows how to add the ROM and RAM to the picocontroller. A memory address register is added also. It will contain the 16-bit addresses necessary to address the memory locations where the program and data are stored.

2.5 The Central Processor Unit

Figure 2-13 shows a representation of a central processor unit, or CPU. Although the operation of most microcontrollers is somewhat more involved, this general description of CPU operation is sufficient to let us see, in principle, how a typical microcontroller CPU functions.

The CPU Registers

> The *programmer's model* is the set of registers in the CPU that you will use in your programs.

The CPU contains registers that you will use extensively in your assembly language programs. In general, there are accumulator registers and registers used to access memory.

Accumulators A and B: The two 8-bit accumulators, A and B, may be a source or a destination operand for instructions that manipulate 8-bit

data. For example, Accumulator A or B may be used to retrieve a byte of data from memory. The registers are called accumulators because the results of an arithmetic or logic operation may accumulate there.

Index register: The program may use an *index* register to access memory. As you will see when you find out more about your own microcontroller, there are a variety of instructions that use this type of register to access memory.

Program counter: Although the program counter is usually shown in the programmer's model, the programmer does not have direct control over it, as is the case with the other registers. The number of bits in the program counter shows how much memory can be directly addressed. In this example, a 16-bit address bus is needed for a 64 Kbyte memory.

The Instruction Execution Cycle

> Each computer instruction is completed during the *instruction execution cycle*, which consists of one or more steps.

The process by which the microcontroller executes each instruction in a program is called the *instruction execution cycle*. When an instruction opcode is to be fetched from ROM, the memory must be supplied with the address of the opcode, and the read control signal must be asserted. This is how the *instruction cycle* starts, and it continues by fetching the rest of the instruction bytes, doing whatever is required by the instruction, incrementing the program counter to point to the next opcode, and then repeating. We can describe the full instruction execution cycle in the following way (refer to Figure 2-13):

- The CPU's *program counter* contains the address of the first byte of the instruction to be executed. We say the program counter *points* to the opcode. The CPU places that address into the memory address register and then onto the address bus.
- The *sequence controller* asserts the *READ_L* control signal on the *control bus*.
- After a small delay, called the *memory access time*, the ROM places the contents of the addressed memory location on the *data bus*.
- The sequence controller writes this byte into the *instruction decoder*.
- The *instruction decoder* holds the opcode byte and decodes it for the sequence controller.
- The decoded instruction causes the sequence controller to go through a sequence of actions that complete the execution of the instruction. These include fetching operands from memory, loading registers, performing an arithmetic or logical operation on a pair of operands, and incrementing the program counter.
- When the instruction execution is complete, the program counter is pointing to the next opode to be fetched and executed. The instruction execution cycle then repeats.

> A microcontroller is *always* fetching and executing instructions.

The instruction execution cycle continues forever, or at least until the power is turned off or a special instruction that stops the cycle is encountered. Remember, while power is turned on, the microcontroller is *always* fetching and executing instructions.

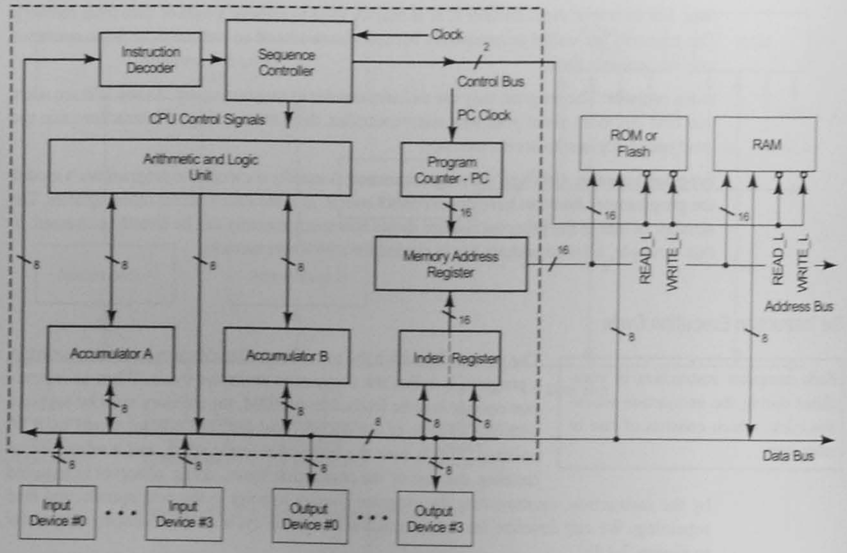


Figure 2-13 The central processor unit (CPU).

The Sequence Controller

The sequence controller is the brain of the microcontroller.

The sequence controller and instruction decoder shown in Figure 2-13 are, in combination, a sequential state machine, much like one you may have designed in an earlier digital logic course. It is the "brains" of the microcontroller. All computers have some kind of sequence controller.

Figure 2-13 shows that the program counter has a clock input. Where do these pulses come from and, even more important, how fast do they come? The name of the game in computer design is speed, and the computer should be designed to run as fast as possible. The time it takes for a change in the input to a logic circuit to appear at the output is called the propagation time. The total time depends on the integrated circuit technology and the number of gates contributing to the delay. You might think that the best way to specify the clock frequency would be to analyze the instructions and see which takes the longest. For this instruction set, add and subtract instructions take the longest time. If we could find out how much time this is, we would make sure that the program counter's clock runs no faster than this. Is there a better way?

The instruction execution cycle described in the preceding section can be broken into smaller elements of time. To see the partition, consider the sequential state machine diagram shown in Figure 2-14 and the timing diagram shown in Figure 2-15.

The instruction execution cycle is partitioned into five different states. Each state is an element of time long enough to allow an event to occur. In State 1, the instruction is fetched from memory; in State 2 it is decoded by the instruction decoder. Then, either State 3 or State 4 is entered depending on the instruction. State 3 is chosen if a register transfer operation, such as `IN`, `OUT` or `MOV`, has been decoded. State 4 is chosen if an `ABA` or `SBA` ALU operation is the current instruction. After State 4, State 3 is entered to transfer the ALU result back into the destination register. In State 5, the program counter is incremented and the memory address register updated. The state machine should be designed to ensure that the time required to do a task is the same in each state.

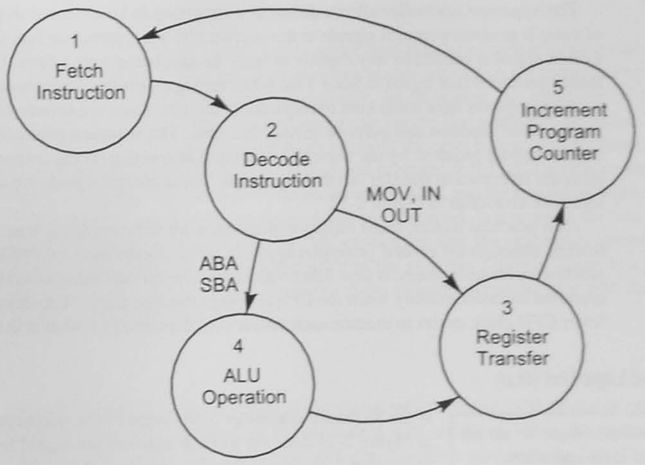


Figure 2-14 A sequential state transition diagram for the operation of the picocontroller.

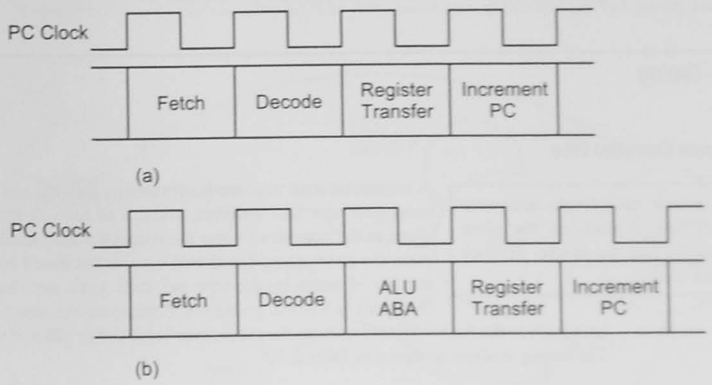


Figure 2-15 Instruction timing diagrams: (a) IN instruction; (b) ABA instruction.

The *sequence controller* allows different instructions to be executed in different amounts of time. It generates control signals at the *correct time* for a particular function. For example, a clock signal is needed by any register to latch the data being stored there. The sequence controller generates that signal in State 3 for whatever register is the destination.

All computers have some kind of sequence controller. There are several ways to design one, but the basic function and purpose remain the same. The sequence controller generates CPU control signals required by the currently executing instruction, at the correct time, to accomplish the information transfer or other operation. These control signals are shown leaving the sequence controller in Figure 2-13.

As a practical matter, these timing explanations are not completely true for all microcontrollers, although the general principles apply. In many modern microcontrollers, an instruction pipeline, or cache memory, is kept filled with instruction opcode and operand bytes by a mechanism that accesses memory while the CPU is doing other operations. This design seems to allow fewer CPU clock cycles to execute each instruction, depending on what is in the pipeline.

Arithmetic and Logic Unit (ALU)

The ALU contains logic to do all arithmetic and logic operations.

If the sequence controller is the brain of the microcontroller, the ALU in Figure 2-13 does the work. It contains the digital logic to operate on the operands as specified by the opcode. It does arithmetic (ADD, SUB, etc.), logic (AND, OR, etc.) and other operations such as shifts, rotates, increments, and decrements. As Figure 2-13 indicated, the ALU receives its inputs from, and places its outputs to, accumulator registers and the data bus.

2.6 Timing

Program Execution Time

The time it takes for an instruction to execute depends on the clock frequency and the number of clock cycles needed.

A sequential state machine is operated by a clock, and different instructions may now take different amounts of time. Normally, this time is given as the number of states the sequential state machine uses to complete the instruction. The actual time is calculated by multiplying the number of states by the time per state. Let's say that the basic clock frequency is 8 MHz, giving 0.125 microsecond (ms) for each state, and let's analyze the time it takes to execute the program in Table 2-6 to add two numbers together. The timing analysis is shown in Table 2-10.

Table 2-10 Program Execution Time

Instruction	Number of States
`IN 1,A`	4
`IN 1,B`	4
`ABA`	5
`OUT A,2`	4
	17 states = 2.125 μs

I/O Synchronization

The speedy CPU must be synchronized to the speed of any slow I/O operations.

It is important to understand that the computer's sequential operation and the time it takes to execute an instruction are controlled by the sequence controller. Accordingly, let us review the operation of the computer. First, the switches are set for the first number to be added. Then the program is started by pressing the reset button. The program counter starts at zero, and the first instruction is fetched and executed in 0.5 μs. This places the first number into the A register. The program counter is incremented, and the second instruction (the second `IN`) is fetched and executed in the next half-microsecond. Is it possible for you to change the data from the first number to the second between the time the program is started and the time the computer takes the second number from the switches? Hardly. There is a problem with the design. We must somehow synchronize the speedy microprocessor with the slow human operator of the switches.

Wait States

Here is a solution to the problem of synchronization. Figure 2-16 shows an additional state, called a *wait state*, is added to the sequential state machine. When the instruction decoder detects an `IN` instruction, the sequential state machine goes into the WAIT state, where it stays until an external control signal, called READY, is asserted by the user of the computer. This signal is like the Enter button on your calculator: after entering the first number into the switches, you must assert the READY signal and allow the computer to progress to State 3, where it does the register transfer. The `MOV` instruction is executed at full speed, and the

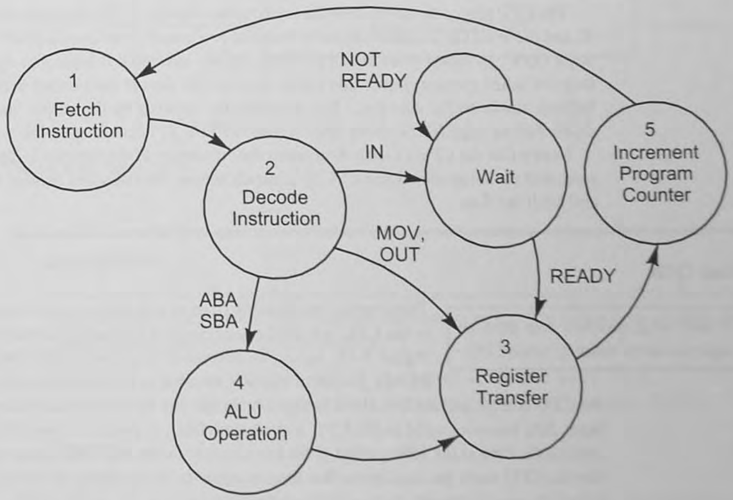


Figure 2-16 A WAIT state is added to allow synchronization with slow input devices.

program counter is incremented. When the second `IN` instruction is fetched and decoded, the sequence controller enters the WAIT state again to wait for you to assert the READY line after the second number has been entered. Although we haven't shown it in this diagram, the output instruction can be designed to enter the wait state also. Then the output device must assert the READY line when it has received the data. This I/O synchronization method, called *handshaking*, is covered in Chapter 9.

Bus Cycle Timing

Data must be taken from the bus or placed onto the bus at the correct time. The CPU controls this timing.

We mentioned in our discussion on the operation of the ROM that there is a short delay while the data is being read out of the ROM. Also, Figures 2-14 and 2-15 show the sequence of steps that occur in the fetching and executing of an instruction. The clock shown on Figure 2-13 controls the timing of all this.

Two fundamental processes of the microcontroller during program execution are writing to and reading from the data bus. These operations are called the *write cycle* and the *read cycle*, and they are used for both memory and I/O access.

Write Cycle

A write cycle transfers data from the CPU to an output register or to memory.

The CPU is the bus master and controls all information transfer timing. Consider transferring data from a CPU register to an output data latch. The CPU's timing is controlled by its clock, and this output operation is called a *write cycle*. Figure 2-17 shows a typical CPU write cycle.

The CPU places the address on the address bus at point A. The data bits are supplied at point B, and the WRITE_L control signal is asserted low a short time later at point C, when the read/write (R/W_L) signal from the CPU is low and the bus clock is high. The output device interface (or RAM memory) uses this signal to latch the data at the correct time (after data have become stable on the data bus.) The data may be captured by the output latch or the memory on the falling edge (C) or rising edge (D) of WRITE_L, depending on the type of latch.

Notice that the CPU clock is four times the frequency of the bus clock. This is normally the case, and the setup allows the CPU to generate timing intervals and control signals to transfer and latch the data.

Read Cycle

A read cycle transfers data from an input device or memory to the CPU.

Transferring information from an external source or from ROM or RAM to the CPU is called a *read cycle*. A typical CPU read cycle is shown in Figure 2-18. Again, an address is supplied by the CPU at point A. The READ_L control signal is asserted at B (when the read/write signal from the CPU is high and the bus clock is high) to enable the input interface three-state gates. The input data becomes valid to the CPU a short time later, at point C. The CPU actually latches (reads) this data at the falling edge of the bus clock at D. An important point to mention here is that the CPU reads the data bus at this time whether or not the input device has it ready. If it is not ready, we need some form of I/O synchronization or a way to extend the CPU read cycle. See Example 2-11.

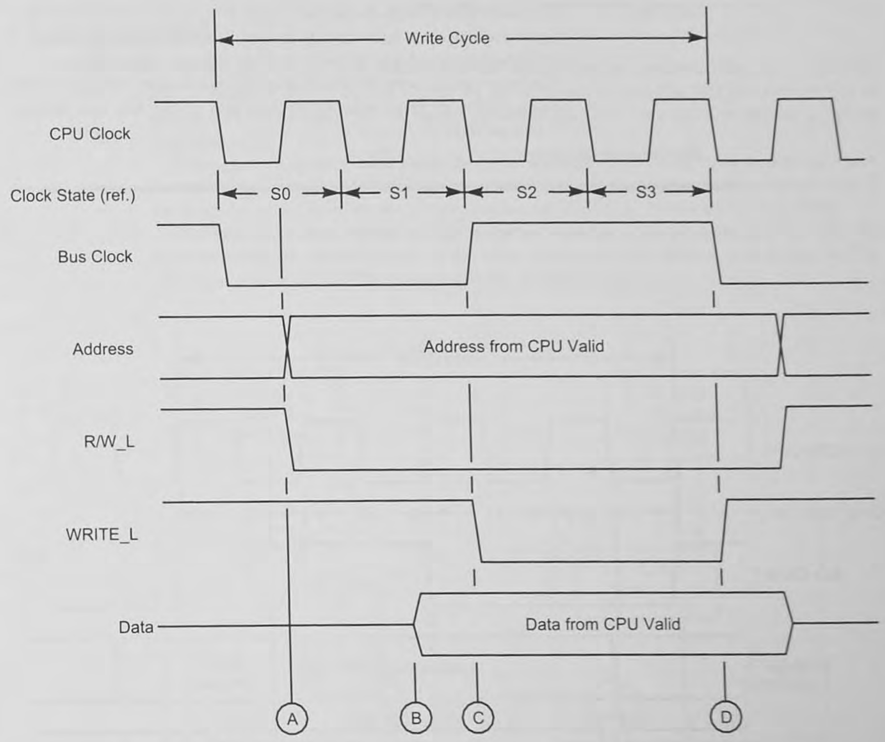


Figure 2-17 Write cycle.

Example 2-11

For each of the following instructions, describe the instruction execution cycle in terms of the read and write cycles needed, assuming that each read or write cycle operates on one byte and that there is no pipeline for instruction bytes.

```
LDAA  #0x12  ; Load the 8-bit A register with the data 0x12
STD   0x1234 ; Store the 16-bit D register in memory location 0x1234
```

Solution

`LDAA 0x12:` First read cycle to fetch the opcode
Second read cycle to read the data (0x12) from the next memory location
Two memory cycles total.

`STD 0x1234`	First read cycle to fetch the opcode
	Second read cycle to fetch the high byte of the memory address (0x12)
	Third read cycle to fetch the low byte of the memory address (0x34)
	First write cycle to write the first byte of data into memory address 0x1234
	Second write cycle to write the second byte of the data into memory address 0x1235

Five memory cycles total.

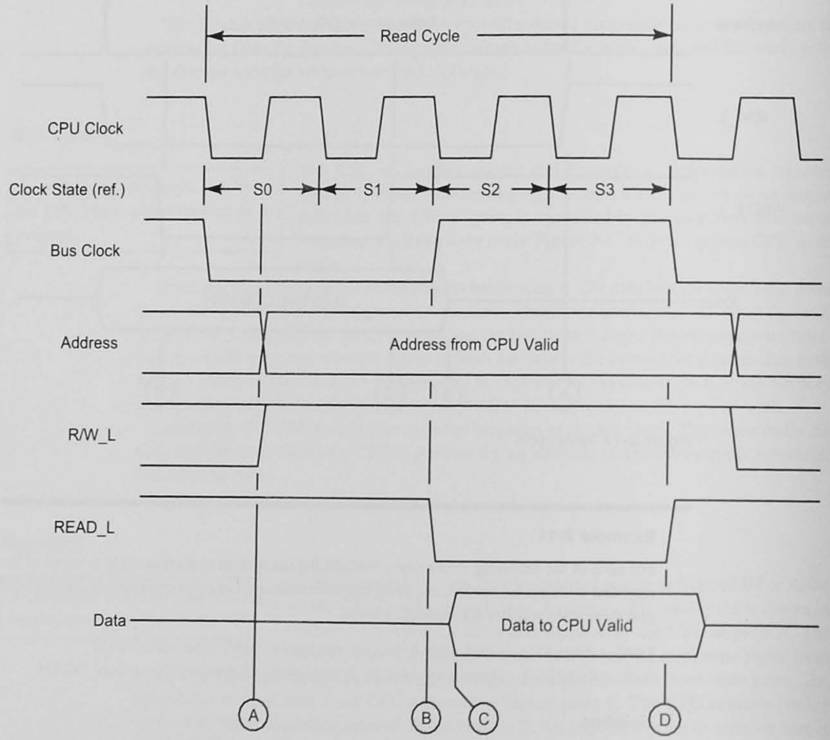


Figure 2-18 Read cycle.

2.7 The I/O Interface

For simplicity, Figure 2-13 shows input and output devices connected directly to the data bus; in practice, an I/O interface must be added to the design. The I/O interface shown in Figure 2-19 has two components, one to *input* data into the microcontroller and one to *output* data from it.

Through a set of three-state gates, the input interface connects an input device, such as a bank of switches, to the data bus. The input three-state gates are activated when the *address* of the input device is placed on the address bus and the READ_L control signal is asserted.

The output interface consists of a set of latches to capture data from the data bus. Like the input interface, the correct address on the address bus asserts the address decoder output. The CPU then asserts the WRITE_L control signal to latch the data.

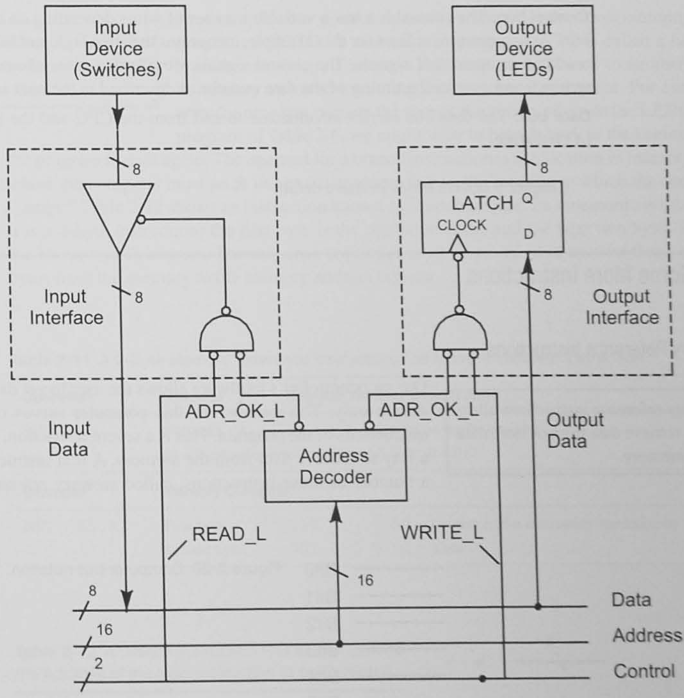


Figure 2-19 Input/output interface.

2.8 The Address, Data, and Control Buses

Three structures, called buses, connect the CPU, ROM, RAM, and I/O interface together (Figure 2-19). A bus can be defined as follows:

A bus is a multiple wire, information pathway with multiple sources and destinations for the information.

A source places information onto a bus and a destination takes information from it. Although the bus has many wires, it is normally drawn on schematic diagrams as one wire with an indicator showing how many wires are used (Figure 2-20).

Address bus: The address bus carries the address from the CPU to the ROM, RAM, or I/O interface to select one particular byte location in the 2^{16} locations in the memory map.

Control bus: The control bus has a variable number of wires depending on the particular system. At a minimum, at least for this example, it contains the READ_L and WRITE_L memory read and write control signals. The control signals provide direction information (reading or writing) and control the timing of the data transfer as described in the next section.

Data bus: The data bus carries information to and from the CPU and the ROM, RAM, and I/O interface.

Chapter 9 covers buses in more detail.

2.9 Some More Instructions

Memory Reference Instructions

Memory reference instructions allow you to retrieve data from or store data in the memory.

Our picocontroller's hardware allows the user to get data from the input device only. The memory of this computer serves only to store the instructions of the program. This is a severe restriction, and we must add a way to retrieve data from the memory. A real instruction set contains a number of these instructions, called *memory reference instructions*, which *read* data from or *write* data to memory. A particular location in memory is accessed by providing the memory with an address, and the various ways of generating this address are called *addressing modes*. One of these is immediate addressing. The move-immediate instruction shown in Table 2-11 is a 2-byte instruction, where the first byte contains the operation and operand codes. The second byte *immediately* follows the instruction byte and contains the data. Of course we must modify the instruction decoder and sequence controller to be able to decode the instruction and generate the control signals.

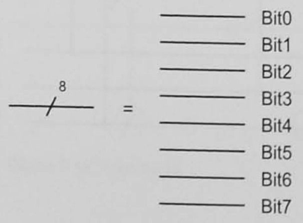


Figure 2-20 Computer bus notation.

Memory reference instructions of other types can read data from or write data into any memory location. You will learn more about these instructions and other addressing modes in Chapter 5 and when you study a real processor.

Control Instructions

The Branch Instruction

Jump or *branch* instructions transfer the program counter from one part of the program to another.

An example of a control instruction seen in other programming languages is a GOTO. In assembly language, a GOTO is called a *jump* or *branch instruction*; it instructs the computer to branch to another place in memory and start executing the program at that point. For example, after the program outputs the sum of the two numbers to the LEDs in the program of Table 2-6, we might want to branch back to the beginning of the program to do it again. The operand for a branch instruction is the location in memory from which the computer must fetch its next instruction, that is, the location to which the computer "jumps." Table 2-12 shows an instruction known as *branch always*. Its mnemonic is `BRA`, and it is a 3-byte instruction; the first byte is the operation code, and the next two bytes specify the 16-bit branch address. The sequence controller must be modified to transfer these address bytes from the memory to the memory address register.

Table 2-11 A 2-Byte Memory Reference Instruction: The Move-Immediate Instruction

Operation	Operand	Register Transfer Description	
`MVI`	8-bit data, dd	(memory location following the opcode) → dd dd = Destination register address: A = 00, B = 01	
Example	**Memory Contents**		
`MVI 65,A`	First byte:	`110 _ _ _ 00`	Operation plus destination operand code
	Second byte:	`001 0 0 0 01`	Data

Table 2-12 A Branch Instruction Has as Its Operand the Address of the Next Instruction to be Executed

Operation	Operand
`BRA`	Memory branch address

Conditional Branch Instructions

The *status* or *condition code register* contains bits that are set or reset when an ALU operation is performed.

The branch-always instruction is an *unconditional branch* because the processor always does the branch. In another type of branch instruction, the *conditional branch*, the computer takes the branch if some condition, or set of conditions, is true. If the condition is false, the next instruction in the memory is fetched and executed. For example, in the addition program we might want to show some error if the addition of the two 8-bit numbers results in a number too large for the 8-bit accumulator. This can be done by attaching to the arithmetic and logic unit a flip-flop called the *carry flag*, which is set when the adder circuit generates a carry and reset when it does not. Other flip-flops can store other information such as a zero result, negative result, two's-complement overflow, and odd or even parity. These flip-flops are contained in a register called the *status*, or *condition code*, register. The status register bits are connected to the sequence controller. We may then design branch-if-carry, branch-if-no-carry, and other conditional branch instructions.

2.10 The Final Picocontroller Design

Figure 2-21 shows the final design. There has been some reorganization of the information flow and a new address register added, so let's briefly discuss these changes.

Figure 2-21 shows two additional accumulator registers (C and D), and the registers and the ALU are connected by an internal 8-bit data bus. Data can now flow between any of the registers and ALU. Input and output interfaces as shown in Figure 2-19 are used for the I/O devices. The data path to the external memory and I/O devices is over an external 8-bit data bus. Information from memory can be transferred into the instruction register for instructions, or any of the registers or ALU for data.

The status bits, which are set or reset by ALU operations, are connected to the sequence controller. They are used to determine whether the branch in a conditional branch instruction is to be taken.

A temporary address register has been added. This is used for branch addresses that are retrieved from memory. For example, when a `BRA` address is fetched from memory, it must be done one byte at a time. The temporary address register holds the address as it is being fetched before it is placed into the program counter to complete the branch instruction.

2.11 Software/Firmware Development

Embedded system software is called *firmware* because it is in ROM and is not so easily changed as programs in RAM.

The software developed for embedded systems is often called *firmware* because, unlike programs you might have written for your computer science classes that are loaded into RAM on a PC or other desktop system, an embedded system requires its program to be in *read-only memory*. Thus, the "software" is more "firm" because it is retained in the computer memory even while the power is removed from the system.[8] "Software" developers must know something about the hardware upon which their

[8] An unknown author, critical of many computer programs being written, once referred to these programs as *mushware*!

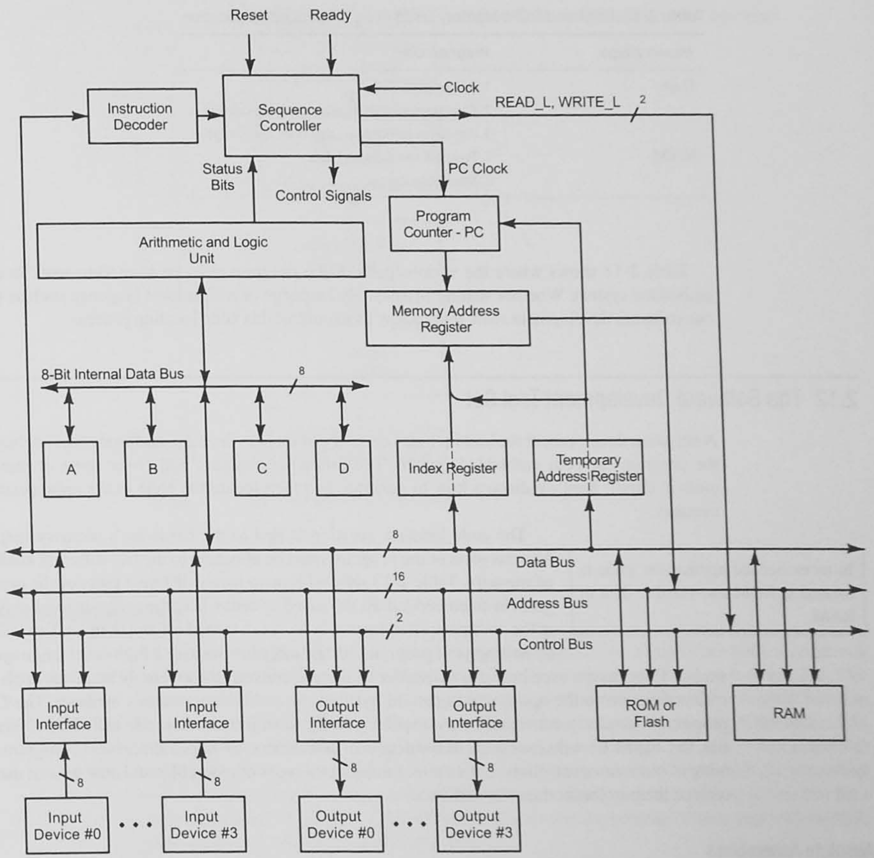


Figure 2-21 The final picocontroller.

software is installed. This hardware is called the *target system*, and it is vital to know the addresses used for the various kinds of memory in the system. Recall the typical *memory map* with both random access memory (RAM) and read-only memory (ROM) shown in Figure 2-9. The ROM may be of several types including *programmable ROM (PROM)*, such as *Flash electrically erasable PROM*. Flash, used in many microcontrollers, allows us to create our software using a development tool set as described next and then to convert it to firmware by programming the EEPROM. The program development process must take into account the physical address of each type of memory and must *locate* the various parts of the program correctly.

Table 2-13 RAM and ROM Memory Used in an Embedded Application

Memory Type	Program Use
Flash	1. All program code
	2. Constants such as messages and lookup tables
	3. Any other information that does not change
RAM	1. Program variables and data
	2. Stack data storage

Table 2-13 shows where the various parts of the program must be located to work in an embedded system. Whether writing in assembly language or a high-level language such as C, our software development tools must allow us to control this code location process.

2.12 The Software Development Tool Set

A software development tool set includes a variety of tools to develop, and sometimes debug, the program for your embedded system. The following chapters will cover some of these tools in detail; here we discuss how to generate and then locate the code in the appropriate memory.

In an embedded application, code is located in ROM and variable data in RAM.

The code location question is tied to the hardware's memory map. Various parts of the program must be allocated to the two different kinds of memory. Table 2-13 showed how to locate different parts of the program in an embedded, ROM-based system where the program must exist in the computer after the power has been turned off and then on. You will be writing your programs in assembly language or a high-level language such as C, or maybe even both. An *assembler* (program) converts the assembly language application program to the opcode and operand bytes in the embedded system's memory. The C program is similarly converted by a *compiler,* usually to an intermediate file called an *object* file. In Chapter 6 we discuss some of the details of assemblers. Chapter 7 discusses C programming of our microcontrollers. Let us now consider two types of assembler and how we can use each of them to locate the code and data.

Absolute Assemblers

When programs are written, the hardware design specifies where the code is to be located in memory. A special directive called ORG provides this information to the assembler. All code is located, *absolutely* at a *specific* memory address, from this information. This is the simplest form of assembler. It takes the source code file and produces an executable file that is transferred (*downloaded*) to the target system. Figure 2-22 shows an absolute assembler in use.

Downloading transfers an executable file from the computer that created it to the computer that executes it.

A major disadvantage of the absolute assembler is that the source file must contain *all* of the source code intended to be in the program. This means that when large programs are being written, all code must be assembled whenever any change is made. Further, the project cannot be split easily into elements that can be written and debugged by different project engineers.

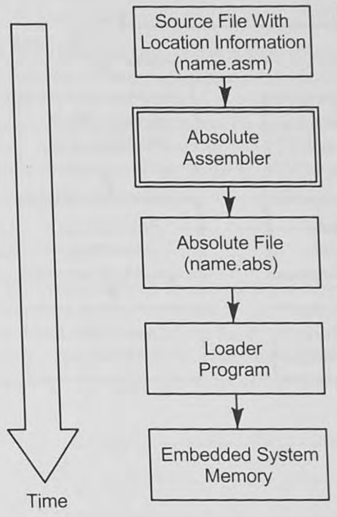


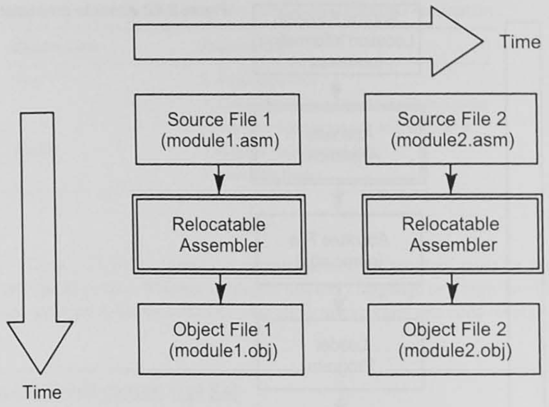
Figure 2-22 Absolute assembler operation.

Relocatable Assemblers

A *relocatable* assembler can overcome the disadvantages of the absolute assembler. As shown in Figure 2-23, the assembler accepts a program, or a program segment, as a source file. The source file does not need to be the complete program, nor does it need to contain location information or ORG directives. The assembler produces an output file, called the *object* file, which contains the binary codes for the operations and as many operands as the assembler can evaluate. When an operand, such as a branch address, cannot be evaluated, the assembler adds this fact (that an address needs to be resolved) to the object file, making it possible for a *linker* program to provide the final addresses. Notice that the program can be split into multiple source files and assembled at different times.

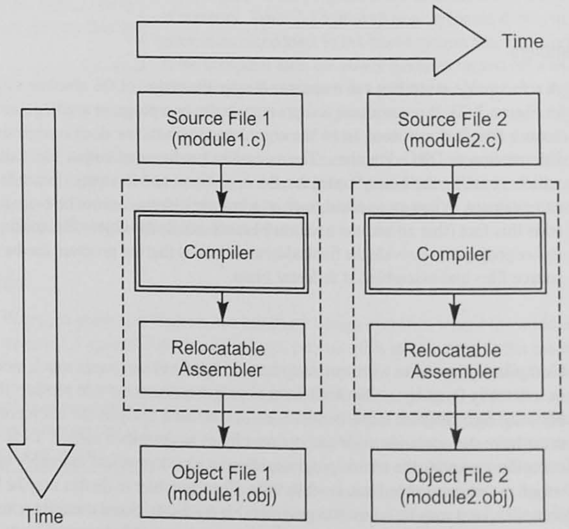
Compilers

Compilers allow us to write our programs in high-level languages much more efficiently than in assembly languages. One high-level C program statement can replace 10 or more assembly language program steps. Nevertheless, in the final analysis the microcontroller's memory must have the operation code and operand bytes as described earlier. To accomplish this, the compiler *compiles* the source program, often to an intermediate assembly language program, which is then assembled into an object file. The assembler to do this may be hidden within the compiler, or it may be a separate program (Figure 2-24). Some compilers, such as Freescale's CodeWarrior compiler, can supply a listing of the assembly language code it produces. Thi can be very useful during the program debugging stages.



Time
Source File 1 (module1.asm)
Source File 2 (module2.asm)
Relocatable Assembler
Relocatable Assembler
Object File 1 (module1.obj)
Object File 2 (module2.obj)
Time

Figure 2-23 Relocatable assembler.



Time
Source File 1 (module1.c)
Source File 2 (module2.c)
Compiler
Compiler
Relocatable Assembler
Relocatable Assembler
Object File 1 (module1.obj)
Object File 2 (module2.obj)
Time

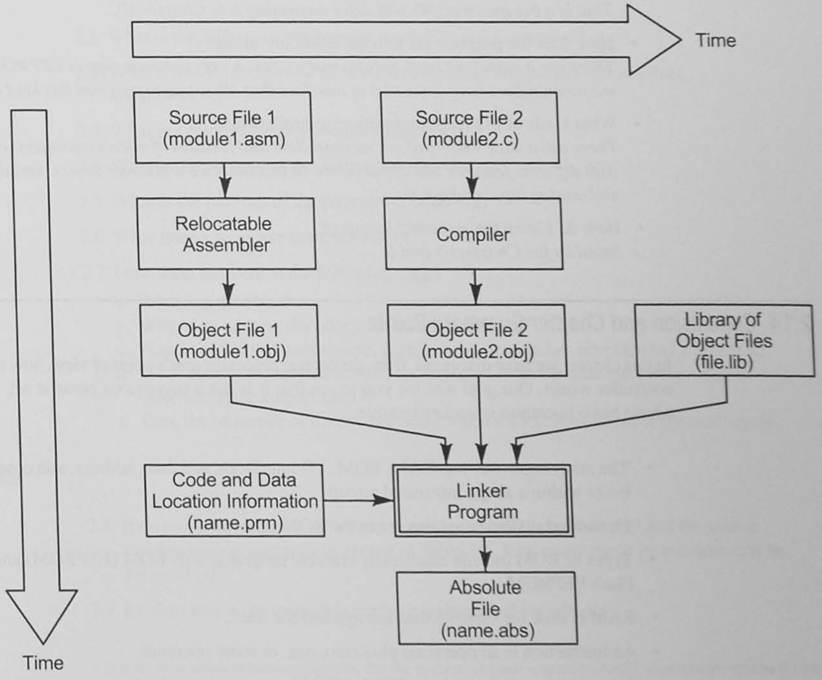
Figure 2-24 Compiler.

The Linker

A *linker* program takes object modules that have been assembled by a relocatable assembler or a compiler, *links* them together, and *locates* all addresses. Figure 2-25 shows two source files, Module1.asm and Module2.c, which are separately assembled and compiled by a relocatable assembler and a C compiler. The linker combines the object files to produce the executable file. You can see in Figure 2-25 that the location information for the code and data parts of the program is given to the linker by a *linker parameter* file (*.prm*). Figure 2-25 also shows that object files can be linked from a library.

Creating a Relocatable Program

The beauty of using the relocatable method to create firmware is that the project can be partitioned by using top-down design techniques and allocated to separate programmers. Each programmer is responsible for developing *modules* that ultimately fit into the whole program. The modules are separately assembled by the relocatable assembler to produce object files. In addition, C program modules may be compiled. These object modules are put together by the



Time
Source File 1 (module1.asm)
Source File 2 (module2.c)
Relocatable Assembler
Compiler
Object File 1 (module1.obj)
Object File 2 (module2.obj)
Library of Object Files (file.lib)
Code and Data Location Information (name.prm)
Linker Program
Absolute File (name.abs)
Time

Figure 2-25 Linker program.

linker, and any addresses or operands that the assembler or compiler was not able to create are generated at this time. Notice in Figure 2-25 that previously assembled and compiled object modules may be maintained in a library. A *librarian* program is included in the tool set to manage the libraries used by various projects.

2.13 Remaining Questions

This chapter has covered a lot of ground to explain some of the basic principles of the operation of a microcontroller. Among the many questions and topics that remain for further discussion are the following.

- How does the microcontroller start executing a program when it is first turned on?
  When the starting address of your program is known, that address is placed into a special place in ROM. When the microcontroller is powered up, or the reset signal is asserted, it goes to that special address to find the starting address of your program.
- How do interrupts work?
  That is a big question! We will defer answering it to Chapter 10.
- How does the program get into the read-only memory?
  There are a variety of read-only memory types. A very common type is EEPROM. Most microcontrollers have some sort of interface that allow you to program this kind of ROM.
- What kinds of I/O features do microcontrollers have?
  There are a wide variety of microcontrollers and families of microcontrollers, each with different features and capabilities. In this book we will cover timers, serial I/O, and analog input and output.
- How do I learn the assembly language?
  Stand by for Chapters 5 and 6.

2.14 Conclusion and Chapter Summary Points

In this chapter we have discussed, from the central processor unit's point of view, how a microcontroller works. Our goal was for you to see that it is not a mysterious beast at all, but one whose basic operation is understandable.

- The microcontroller has RAM, ROM, I/O interfaces, and data, address, and control buses within a single integrated circuit.
- Embedded system programs are in the ROM.
- Types of ROM include electrically erasable programmable ROM (EEPROM) and Flash EEPROM.
- RAM is used for variable data storage and the stack.
- An instruction is an operation plus zero, one, or more operands.
- To retrieve or read data from ROM or RAM, you must supply the memory with the address and a READ control signal.

- To write data into RAM you must supply the address, the data, and a WRITE control signal.
- The instruction execution cycle repeats continuously.
- An input interface is a set of three-state gates connecting an input device to the data bus.
- An output interface is a set of latches into which data from the data bus is latched during an output operation.
- The CPU controls the timing of all read and write operations.
- Embedded system software is often called firmware.
- A relocatable assembler allows you to develop application software in modules that are linked together to create the program to go into the ROM.

2.15 Problems[9]

Explore

2.1 What is the difference between an assembler and a compiler? [a, c]

2.2 What is the advantage of a relocatable assembler compared to an absolute assembler? [a]

2.3 What is a microcontroller memory map? [a]

2.4 What is the purpose of the instruction decoder? [a]

2.5 What is the purpose of the program counter? [a]

2.6 What does a sequence controller do? [a]

2.7 Give short answers to the following: [a,g]

a. What is a data bus?
b. Why is an address decoder used in I/O interfaces?
c. How is an information source, such as a set of switches interfaced to a data bus?
d. What control signals are needed to latch data from the data bus into an output interface at the correct time?
e. Give the sequence of events that occur when a CPU does an input (or read) cycle.

Stimulate

2.8 If all move instructions are coded in one byte with the opcode 010 and the source and destination operands as shown in Table 2-5, how many move instructions can be defined? [a]

2.9 Explain why a computer has ready or wait control signals. [c]

[9] One or more letters in brackets signifies that the problem in some way meets ABET accreditation criteria for outcomes a–k. The criteria reference abilities, understandings, and habits of thought that are necessary in engineering and computing, among other fields.

2.10 Discuss the difference between an absolute and a relocatable assembler. [a, k]

2.11 How do most microcomputer systems solve the problem of multiple sources of information present on a data bus? [g]

2.12 Why must a three-state gate be used to interface an input device to the data bus? [a, c]

2.13 Why must a latch be used to interface an output device to the data bus? [a, c]

2.14 For a CPU performing a write cycle, why does the CPU place the data on the data bus before asserting the WRITE_L control signal? [a]

2.15 A microcontroller memory map shows 16 Kbyte of Flash EEPROM (ROM) in memory space 0xC000–0xFFFF and 1 Kbyte of RAM in memory space 0x1000–0x13FF. [c, k]

a. Give a range of addresses (in hex) suitable for locating code.
b. Give a range of addresses (in hex) suitable for allocating variable data storage.

Challenge

2.16 Discuss the changes that must be made to the sequence controller to add the move-immediate instruction discussed in Section 2.9. [c, e]

2.17 Design an instruction decoder as shown in Figure 2-13 using AND, OR, and inverter gates to decode the 3-bit opcodes and produce a control signal asserted by each of the operations given in Table 2-5.

2.18 Design the hardware required to implement a `HALT` instruction, which stops the CPU from progressing further in the program. [c]

2.19 Describe the instruction execution cycle of a move-immediate instruction shown in Table 2-11. [c, e]

2.20 Draw a timing diagram relative to the CPU clock shown in Figure P-2-20, which includes the address and data buses, R/W_L and the write control signal (WRITE_L = active low) and shows a write cycle. [a]

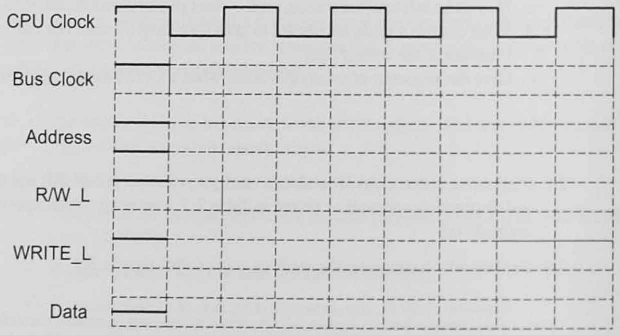


Figure P-2-20

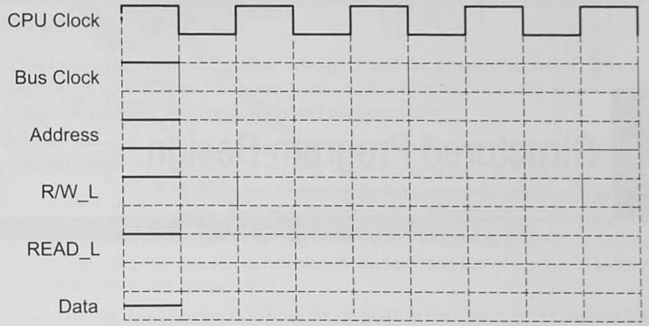


Figure P-2-21

2.21 Draw a timing diagram relative to the system CPU clock shown in Figure P-2-21, which includes the address and data buses, R/W_L, and the read control signal (READ_L = active low) and shows a read cycle. [a]

2.22 A CPU generates a bus clock and R/W_L signal during a write cycle as shown in Figure 2-17. Give a logic equation or show a logic diagram expressing the logic required for the WRITE_L control signal.

2.23 A CPU generates a bus clock and R/W_L signal during a read cycle as shown in Figure 2-18. Give a logic equation or show a logic diagram expressing the logic required for the READ_L control signal.

Reflect on Learning

2.24 Create a five list of questions you would like to have answered to be able to understand how the microprocessor or microcontroller you are studying works.

2.25 How does what you learned in this chapter compare to what you previously knew about the operation of a computer?

2.26 What have you learned in this chapter that you think will make it easier for you to write microcontroller programs?

3 Structured Program Design

Objectives

This chapter presents a design procedure, called top-down design, suitable for both hardware and software projects. You will learn to use tools to design programs following the top-down design procedure and the principles of structured programming. Designing before writing is vital to producing good software.

3.1 The Need for Software Design

In the design and development of many systems, the cost of producing software is higher, often much higher, than the cost of the hardware. Frederick Brooks, in *The Mythical Man-Month*,[1] compares large-system programming that does not use good design techniques with the tar pits that swallowed saber-toothed tigers, dinosaurs, and mammoths. Few of these systems meet their goals in terms of schedules and costs. Designing the software before writing the code is vital both to controlling costs and to meeting requirements and schedules.

Software design means *designing* the software *before* writing the code. When you are beginning your studies of any processor, or any programming language, designing before writing is difficult. You are wrapped up in just learning the details of the processor and its instruction set or the syntax of the programming language. Soon, however, the problems get more complicated and, with your newfound mastery of the language, you should be able to design the solution to the problem instead of just programming the solution.

In this chapter we assume that you are about to learn the instruction set of a microcontroller and the operation of the assembler or high-level language compiler. To prepare for this task, we would like you to learn how to design software properly instead of just writing it. We will look at various design philosophies and at tools used to design software.

[1] Frederick P. Brooks Jr. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, 1982.

3.2 The Software Development Process

Developing fully designed, coded, debugged, and documented software for any real system proceeds in several steps. These are design, coding of modules, testing and debugging of modules, system testing and verification, and documentation.

All software development starts with a *design* phase.

1. **Design:** The design for any complex system might well take 50% or more of the total effort required for a project. In the sections that follow we will distinguish between design methodologies and design tools. A design methodology is a philosophy to do design, and design tools are the mechanics used. The goal of the design phase is to understand completely the problem and to propose a solution broken down into modules or functional elements that can be coded, tested, and documented.

2. **Coding:** Coding means writing the program in the chosen programming language. We would hope to use a high-level language for most of the code; but often, especially in time-critical applications, assembly language programs are needed.

3. **Module testing:** A properly done design will have coded modules that can be tested and proven to work correctly. The testing and debugging tools used depend on how we have done the coding. Fortunately, many high-level languages have very powerful debuggers that allow us to test and debug our software.

4. **System testing:** This step follows subsystem or module testing and is necessary to prove that the software and hardware work as a whole.

Documentation is so important that it accompanies each step in the process.

5. **Documentation:** Although mentioned last in the list of steps, *documentation in fact accompanies each step of software development.* The design documentation specifies what the system is to do and how the function is implemented; typically, this work will form the basis of user manuals. Documentation effort is never wasted. Documentation begins in the design step, and various types of design documentation are discussed in later sections of this chapter. The documentation produced in the coding phase is the code itself, which includes features of the design as comments. Code testing phases are documented with test plans and results. These not only become templates to show that the system meets the specifications, they also allow future modifications of the software to be tested to the same standard. Documentation efforts include the installation and user manuals.

3.3 Top-Down Design

A design methodology is a stepwise procedure for doing the design. This can be contrasted with design tools, which are the mechanical things (e.g., pseudocode or flowcharts) used to produce the design. The top-down design (TDD) method is the design procedure of choice. By following the steps presented next, we can almost assure ourselves that in the end there will be a good design, meeting the system requirements.

Understand the Problem Completely

Understand what is required of the system before starting to program.

Unfortunately, many programmers violate this first principle of TDD right away because it is so much fun to program that they start before they fully understand the problem. For example, consider designing the hardware and software for a digital voltmeter. Questions that should be asked (and answered) before proceeding with the design might include the following:

"What is the range of input voltages?"

"What is the resolution needed for the display?"

"How are the analog voltages coded?"

"How does the analog-to-digital converter work?"

Understanding the problem means we must specify exactly what the software is required to do. It is not necessary to understand (at least in the initial stages of the design) how elements of the proposed solution work in detail. For example, when designing a digital voltmeter, we do not need to know how the output display works. We just have to know what we need for the output.

A student recently suggested that this part of the design process be called "outside-in design" to emphasize that the specifications for the software often come from an outside customer. The specifications must be written so that both the end user and the engineer of the system know exactly what the system is to be.

The *requirements specifications* tell exactly what the program is supposed to do.

A document that is produced during this phase of the design is called a *requirements specification*. This bit of jargon simply means that you specify (write down) what the system is required to do. We are not specifying *how* something is to be done, just *what* is to be done.

The design process should consider potential error conditions and should allow for them in the rest of the design. Often when customers supply specifications, they fail to consider all error conditions. You should make it your responsibility to think about errors and error handling requirements.

A statement that summarizes this first principle of top-down design is "Think first, program later."

Design in Levels

Upper levels of the design are more general; lower levels are more detailed.

Once the requirements have been specified, it is time to start designing a system to meet them. This is the "how" part of the design process. It is natural to feel overwhelmed by the complexity of the problem. Often one cannot see a way to the end. Do not worry. The design procedures will help us through to the end.

Designing in levels means that we recognize that the whole solution to the problem cannot be seen at once. Just start at an upper level and propose a solution to the problem. As you learn more about the problem and how to solve it, levels that are more detailed can be added to the design. A tree structure, as shown in Figure 3-1 is developed to represent a design that is being done in levels. The upper levels of the tree are more general statements of the problem solution; as one progresses down the tree, more detailed information is shown.

Let us look at an example. Consider designing the software for a digital voltmeter. The requirements are the following:

The input voltage ranges from 0 to 5 V.

An analog-to-digital converter is to be used to produce an 8-bit unsigned binary code.

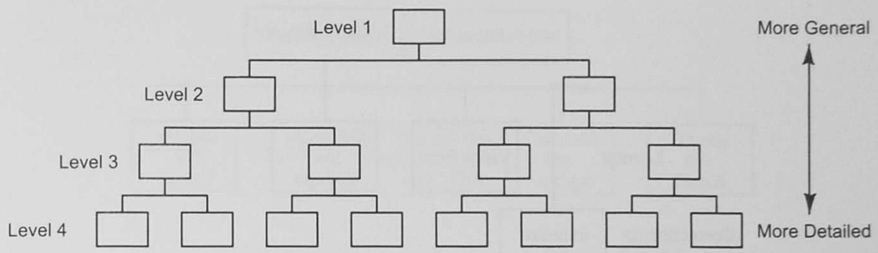


Figure 3-1 Tree structure that results from designing in levels.

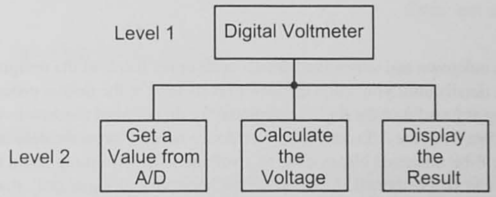


Figure 3-2 Two-level design for a digital voltmeter.

The voltage is to be displayed on a two-digit, seven-segment LED display to a resolution of 0.1 V.

We will not complete this design to the final level of details needed in a real-world project. Our goal is to show how to start a top-down design. The first two levels of the design are shown in Figure 3-2.

The top level is a simple statement of the problem, with the next level providing some details of how that top block is to be done. This level starts to focus our thoughts as we consider what should be done to program the digital voltmeter. The design may not be correct or complete at this stage, but it is at least a start, and starting is often the hardest part of any project. Notice that the blocks in Level 2 are algorithmic. That is, by reading them from left to right, we have a description of a sequence of things done to input the voltage and display the result.

Ensure Correctness at Each Level

The design started in Figure 3-2 is not necessarily correct or complete after the first pass. Before going on to lower levels, make sure the algorithm is correct at this level. In going back over the design, try to think of anything else that perhaps should be done. For example, we might remember that we need to initialize some of the I/O devices in the system. It is easy at this stage to add another block to the design, as shown in Figure 3-3.

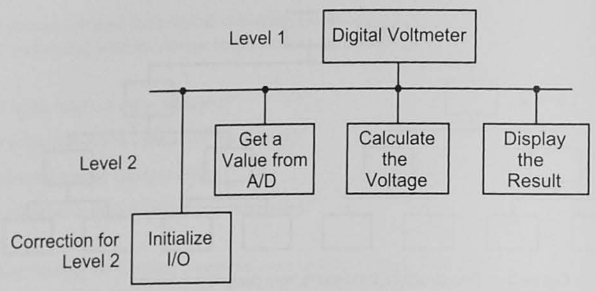


Figure 3-3 A more correct design for Level 2.

Postpone Details

There will be unknown and unresolved details at all upper levels of the design. Postpone thinking about the details until you reach the lower levels later in the design process. For example, when working at Level 2 of the digital voltmeter, we do not need to know in detail how we are going to get data from the A/D converter. Nor do we need to know the details of the algorithm to convert the 8-bit unsigned binary code to a voltage value. Thinking about and designing for these details can be postponed. At Level 2 it is necessary to know only that this conversion needs to be made, not the details of how to do it.

Successively Refine Your Design

As progress is made through the lower levels, more details of what is required become apparent. Inevitably, as this occurs, we think of something that could be done at an upper level to make the design easier at lower levels. That is OK. Since no time has been invested in programming, it is easy to change the design. Go back to the upper level, change it, make sure it is now correct at that level, and continue to work at the lower levels.

Design Without Using a Programming Language

The initial design should propose solutions to the problems that are independent of any programming language. It should make no difference to the design how the machine code in the memory of the computer is generated. We are now beginning to talk about design tools—the tools and techniques used to write down the design. One widely used design tool is pseudocode. This is a programming-like language used for design. For example, a pseudocode design for the digital voltmeter at Level 2 is shown in Table 3-1.

Table 3-1 Pseudocode Design for a Digital Voltmeter

```
Initialize I/O devices
Get a value from the A/D
Calculate the voltage
Display the results
```

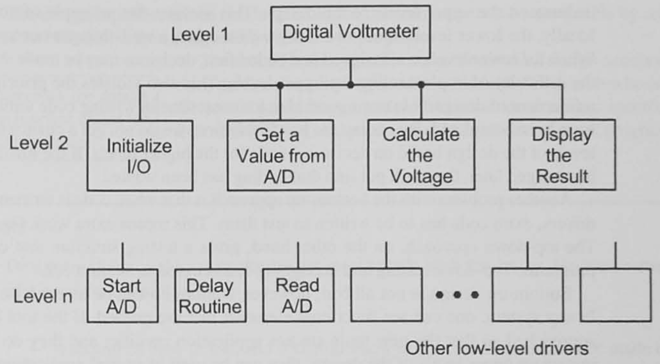


Figure 3-4 Bottom-up design.

3.4 Design Partitioning

Most programming problems can be partitioned into elements that are divided among the programmers working on the job.

The top-down design method allows us to partition the design into easily handled pieces. At the upper levels, we can concentrate on more general ideas, leaving the detailed design until later. Also, it is usually easy to see where work at the upper levels can be divided among different people working on the project. In the digital voltmeter design, it would be easy to split the design at Level 2 into two parts. One engineer could work on the I/O initialization and on getting data from the analog-to-digital converter, and another could be assigned to convert the unsigned binary data to the voltage display. Partitioning the design and allocating work to different people is part of managing a software development project.

3.5 Bottom-Up Design

In *bottom-up design,* low-level functions are designed, coded, and tested before the upper levels of the design are completed.

"Bottom up" is design philosophy that some people use. They think they are doing top-down design, but they really are not. Here is how designers may fall into bottom-up designs. They begin with a top-down design for the first levels: for example, the digital voltmeter design could be started just as before. So far, so good. But soon they start looking ahead to doing some coding. After all, they are programmers, aren't they? There will be some low-level drivers required, such as a routine that reads the A/D. Why not, they argue, take a break from this design stuff and do some programming for a change? The design starts to look like Figure 3-4.

What is wrong with this procedure? First, by writing programs before the design has been completed, we cast in code[2] how things are being done at lower levels before we have

[2] Sometimes very much like concrete!

understood the upper levels of the design. This violates the principle of postponing details. Ideally, the lower levels should be *designed* based on a well-thought-out upper level design. When the lower levels are designed and coded first, decisions may be made that could increase the difficulty of implementing the upper levels. This also violates the principle of successive refinement of design. It is not a good idea to invest time in writing code until all design levels have been completed. By coding the low levels first, we do not get a chance to optimize lower levels of the design based on decisions made for the higher levels. If the low-level design must be changed later, the work put into that coding has been wasted.

Another problem with the bottom-up approach is that when code is written for the low-level drivers, extra code has to be written to test them. This means extra work for the programmer. The top-down approach, on the other hand, gives a testing structure that can test low-level programs. Top-down testing and debugging is discussed in Section 3.9.

Bottom-up design is not all bad, however. Bottom up can be an exercise in tool building. In any system, one can see functional elements that are needed. If the tool building phase is approached so that the new tools are not application specific, and they do not have a great impact on upper levels of the design, they can be used in several applications. This may save work in the end.

3.6 The Real-World Approach

Top-down design combined with judicious use of functions already programmed works best in the real world.

Rarely in the real world do we have the opportunity to follow the ideals of the top-down philosophy and complete the design to all levels of detail before doing any coding. Often low-level functions are available that have been coded and tested; you can use them in your design. It makes sense to use these working functions and not have to redesign, code, and test them again. Most high-level languages come with libraries of functions, and your company or coworkers may have useful libraries, too. Using these functions violates the principles of top-down design but is acceptable, provided you understand why the principles are being compromised and what the consequences may be. Using previously written low-level functions may impose constraints on the higher levels of the design. However, the time saved by using already working functions can offset the disadvantages of these constraints. Note that this is different from the bottom-up philosophy. In bottom up, one sets out to write the low-level drivers, putting effort into their design, coding, and testing.

In summary, the real-world approach is one in which we recognize the power of the top-down method and attempt to do as much design as possible before coding, but we use previously developed and debugged functions where possible.

3.7 Types of Design Activity

Functional design is the more general activity in which the required functional elements are defined.

The top down design philosophy supports two types of design activity found in any software development project. The first is oriented toward defining the *functionality* required in the software. We do not care *how* the software does its thing as much as *what* function is to be provided.

The design must be refined to a level at which the components are manageable by one person. That function is then assigned to someone to program.

Detailed design specifies the details necessary for each function called for in the functional design stage.

The second activity is arriving at the *detailed design* necessary to produce the functionality required in a module. The person who is assigned the job of producing a module takes the requirements specified by the first activity level and produces a detailed design to be programmed.

3.8 Design Tools

Design tools are used by the software engineer to help with the design. Here are some qualities of a good design tool.

- It should be easy to use, and it should allow design modifications to be made easily.
- It should support structured programming.
- It should allow us to see easily the design at many levels.
- It should have good documentation facilities.

Structured Programming

Any program can be written with just *sequence*, *decision*, and *repetition* structures.

In the mid-1960s, people writing software for large systems were appalled at the cost of these systems and at the amount of time needed to develop them. A landmark paper by Bohm and Jacopini[3], said in 1966 that any proper program was equivalent to a program that contains only three structures. That is, we can construct *any program* with only three basic structures, none of which is a GOTO statement. No one paid much attention until two years later, when Dijkstra[4] wrote a provocative paper stating that the GOTO statement in a program is harmful. The three structures that Bohm and Jacopini suggested (and a few more that software designers could not resist adding) form the basis of structured programming and the structured languages we know today.

The three basic structures are a *sequence*, a *decision*, and a *repetition*. Beyond these, several general principles of structured programming can be enumerated:

1. Use these simple structures to aid in minimizing the number of interactions and interconnections between elements of the program.
2. Keep program segments small to sustain manageability.
3. Organize the problem solution hierarchically.

[3] Corrado Bohm and Giuseppi Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Communications of the ACM* vol. 9, nos 5, pp. 266–371, 1966.
[4] Edsger W. Dijkstra, "GOTO statement considered harmful," *Communications of the ACM* vol. 11, no. 3, pp. 147–148, 1968. Some programmers suggest that Dijkstra claimed that GOTO is a four-letter word.

4. Organize each program segment to have one input and one output (in terms of the program flow). This is not a data restriction but a restriction on the program flow. We would like to draw a box around a program segment, enter that segment in only one place, and leave it at only one place.

Structured programming really is not a design tool. It is a way of writing programs. However, because the principles of structured programming fit so well with the top-down design procedure, the elements of structured programming have been adapted for use as a design tool. With that in mind, let us look at the sequence, decision, and repetition structures used in structured programming, along with the pseudocode design tool. We will then see how pseudocode can be used in a top-down design exercise.

Pseudocode

The popular pseudocode technique is frequently used design tool because it is easy to modify, does not require special graphical tools, and fits well with the documentation required for all design. Further, the design text can be included in the software code as comments.

Many texts give a complete treatment of the pseudocode design tool. Here is an abbreviated approach that shows how to pseudocode the three simple design structures: sequence, decision and repetition.

1. **Sequence:** A sequence structure is a sequence of functions or operations that the program is to perform. A sequence usually does not show any logic. It should show the function provided by a process block and must have a beginning and an end. These are explicitly stated to show the single-input, single-output form we would like to achieve in the design. Thus, a sequence of A, B, C would be as shown in Table 3-2.

The ellipsis (...) represents the elements of the design provided in the A, B, and C blocks.

2. **Decision:** The decision structure is called an *IF-THEN-ELSE*, and Table 3-3 shows how to write the structure in pseudocode. The decision structure code executes one of the two elements in the program. *IF* X is true, *THEN* the process A is executed, otherwise (*ELSE*) B is executed.

Another view of the IF-THEN-ELSE structure is shown in Figure 3-5. This is a structured flowchart symbol, and while flowcharts are not generally used as design tools these days, they are useful to help visualize the proper structure of a good design. It is easier to see that only one process block is executed in Figure 3-5 than in the corresponding pseudocode of Table 3-3.

Table 3-2 SEQUENCE Pseudocode

```
BEGIN A
  ...
END A
BEGIN B
  ...
END B
BEGIN C
  ...
END C
```

Table 3-3 DECISION Pseudocode

```
IF X
  THEN
    BEGIN A
      ...
    END A
  ELSE
    BEGIN B
      ...
    END B
ENDIF X
```

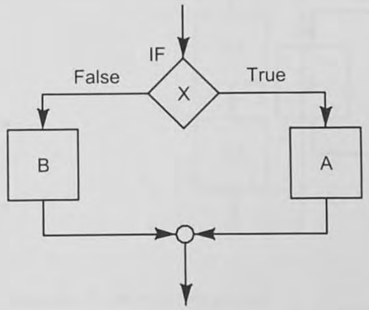


Figure 3-5 IF-THEN-ELSE decision element.

Inspection of Table 3-3 and Figure 3-5 shows that a Boolean or logic decision is made and must be either true or false. It doesn't have to be a simple decision; you may use any of the Boolean logic learned in your logic design course. For example, the following is a Boolean function:

F is TRUE if (A is TRUE AND B is FALSE) OR C is TRUE OR D is FALSE.

The decision structure may be single sided, however. That is, there might not be an ELSE part to the decision. Table 3-4 and Figure 3-6 show the pseudocode and the corresponding structure.

Note that the IF-THEN-ELSE pseudocode block ends with an ENDIF X statement. The ENDIF, of course, signifies the end of the block. Repeating the conditional at this point is a good technique to help you remember what the decision was all about. This feature is especially useful later, when you are looking at the design.

3. **Repetition:** The pseudocode for a repetition structure is shown in Table 3-5. This structure is called a *WHILE-DO*, and as you can see in Figure 3-7, *WHILE* the Boolean X is true, the process elements S1, S2, and S3 are *DONE*.

Table 3-4 Single-Sided DECISION Pseudocode

```
IF X
  THEN
    BEGIN A
      ...
  END A
ENDIF X
```

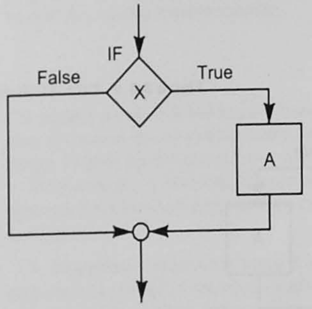


Figure 3-6 Single-sided DECISION structure.

Table 3-5 WHILE-DO Pseudocode

```
WHILE X
DO
   BEGIN S1
      ...
   END S1
   BEGIN S2
      ...
   END S2
   BEGIN S2
      ...
   END S2
ENDO
END WHILE X DO
```

There are some other variations of the repetition structure. One particularly useful in assembly language programming is the *DO-WHILE*, shown in Table 3-6 and Figure 3-8. Here the processing blocks, S1, S2, and S3, are done before the Boolean decision block. Thus, the code in the DO block is executed at least once.

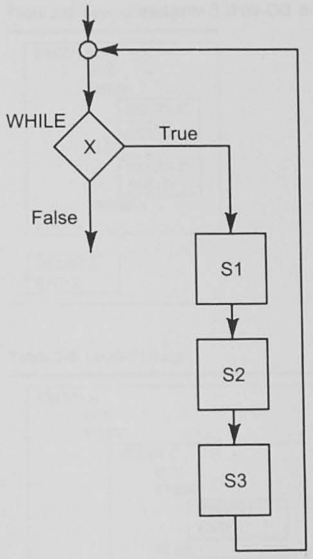


Figure 3-7 WHILE-DO structure.

Table 3-6 DO-WHILE Pseudocode

```
DO
   BEGIN S1
      ...
   END S1
   BEGIN S2
      ...
   END S2
   BEGIN S2
      ...
   END S2
   ENDO
WHILE X
ENDO WHILE X
```

Two additional definitions complete our introduction to pseudocode.

Indentation: Indentation is often used in pseudocode. The code statements (or design requirements) for each block (bracketed by BEGIN and END) are indented to help show the structure of the design.

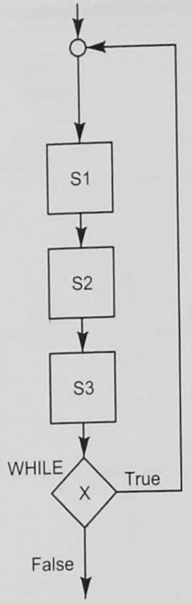


Figure 3-8 DO-WHILE structure.

Table 3-7 Level-1 Design

```
BEGIN A
END A

BEGIN B
END B
```

Single-input, single-output: A principle of structured programming is to keep things simple without many interconnections between different parts of the program. A way to do this is to write the program so that elements of it (sequences, if-then-elses, and repetitions) have single entry and exit points.

Using Pseudocode Structured Elements as a Design Tool

A top-down design can be done in several levels of pseudocode. For example, when you first start the design, you might know only that A and B have to be done. The Level-1 design is shown in Table 3-7.

Table 3-8 Level-2 Design

```
BEGIN A
        IF X
        THEN
                BEGIN C
                END C
        ELSE
                BEGIN D
                END D
        ENDIF X
END A

BEGIN B
END B
```

Table 3-9 Level-3 Design

```
BEGIN A
        IF X
        THEN
            BEGIN C
                IF Y
                THEN
                        BEGIN E
                        END E
                ELSE
                        BEGIN F
                        END F
                ENDIF Y
            END C
        ELSE
            BEGIN D
                IF Z
                THEN
                        BEGIN G
                        END G
                ELSE
                        BEGIN F
                        END F
                ENDIF Z
            END D
        ENDIF X
END A

BEGIN B
END B
```

As we start to know more about the sequence block A, we can begin to fill in its details. The Level-2 design becomes that shown in Table 3-8.

The design goes on to Level 3 (Table 3-9), where the C and D sequence blocks can be expanded.

In each of the design levels shown here, elements have been enclosed in boxes. This is to emphasize the single-input, single-output nature of the program flow. In Table 3-7 we can see that the A and B blocks are quite separate. By the time we get to Table 3-9, the separate blocks for A and B are still apparent even though A has been expanded.

3.9 Top-Down Debugging and Testing

Stubs are dummy programs for functions or subroutines that have not been written yet.

The discussion of the bottom-up design technique in Section 3.5 showed that extra code is required to test the modules. However, if the code is developed in a top-down fashion by writing the higher level modules first and postponing the details of lower level modules, the program structure can test itself. The program development might progress as illustrated in Tables 3-7 through 3-9. The upper level of the program is coded as calls to the modules that provide the functions A and B. As we start to work on function A, delaying our work on the others, we must have something to substitute for function B so the top-level program will run. Thus, any lower level module that has not yet been coded is temporarily coded as a *stub*. A stub is just a return with no processing done. Table 3-10 shows how the program might look for the Level-2 design. Here, functions B, C, and D are coded as stubs while we work on A. If a function must process some data and return a value, a dummy or test value can be returned by the stub to be used by the calling program. In this way, you can delay the actual programming of the lower level functions but still develop and test the upper levels. The whole design becomes the test jig for itself. It allows you to design and test program logic and to see how data are passed back and forth at higher levels before coding the lower levels.

Table 3-10 Using Stubs for Unfinished Modules in Level-2 Design

```
BEGIN PROGRAM
        Call Sub_A
        Call Sub_B
END PROGRAM
```

The main program consists of calls to subroutines only.

```
BEGIN Sub_A
        IF X
        THEN
                Call Sub_C
        ELSE
                Call Sub_D
        ENDIF X
END Sub_A
```

This is the module we are working on, and it calls two more subroutines.

```
BEGIN Sub_B
        (Return)
END Sub_B
```

We have not started working on this module yet, so it is just a stub. It returns without doing any processing.

```
BEGIN Sub_C
        (Return)
END Sub_C
```

This is a stub too, but we will probably start to work on it next.

```
BEGIN Sub_D
        (Return)
END Sub_D
```

Another stub awaiting attention.

3.10 Structured Programming in Assembly Language

Structured programming and structured languages were invented to increase our efficiency as programmers and to make it easier to produce software without bugs and problems. What if the program has to be written in assembly language? Although assembly languages are vastly different from high-level languages and do not have built-in structured language elements, it is still possible to write structured code in assembly language. You must remember the principles of structured programming, particularly the idea of a single input and single output for a process block. Make your process blocks small, with jumps that do not span over great chunks of code, and ensure that there are no jumps into the middle of a block of code. This is what a compiler does for programs written in a high-level language. We will see some examples of structured assembly programming in Chapter 6.

3.11 Program Comments

There should be comments in the program that describe the *design* of the code needed to meet the system requirements.

The preceding sections describe the top-down design process and how to use structured programming techniques to implement a design. Developers of high-level languages often claim that their language produces "self-documenting" code. Nothing could be further than the truth. A language cannot document a design; it documents only the implementation of the design, and poorly at that.

Many beginning programmers, and it is sad to say experienced ones as well, wait until the program is completed before including any comments. This at best produces sloppily done and incomplete comments. We have emphasized the importance of completing the *design* of a program before any code is written. We have shown, however, that in the real world coding may start before all detailed design is complete. This is possible because there exist previously developed modules suitable for use in a given application and because the top-down debugging and testing scenario presented in Section 3.9 supports their use. Nevertheless, we should always write down some of the design before any coding is done, to allow us to test the design first. These design comments form the basis of a useful commenting strategy.

There should be two types of comments in our programs, design comments and code comments.

Design Comments

The design comments convey what the program is to do, not how the program will do it.

Program Header

After reading the header, you should know what the program does, not in any great detail, but at least in general. The author's name should be here, so praise (or blame) can be apportioned correctly. The date of original code release and modification record is good information too. The modification record should give what has been done to the original code, when it was done, and by whom. See Example 3-1.

Example 3-1 Program Header

```
/*****************************************************************
 * Hex keypad scanning module
 *   unsigned char hex_key_scan( void );
 * This module scans a 16-key keypad
 * attached to Port AD. It returns an unsigned char ASCII code
 * for the key pressed. It returns the first key pressed
 * when scanning. It does not check for multiple keys pressed
 * at the same time and it does not debounce key strokes.
 * Author: F. M. Cady
 * Source File: hex_keypad.c
 * Revision: 1
 * Revision date: 1 February 2009
 *****************************************************************/
```

Hardware Definitions

As you start to learn more of the details needed to solve your programming problems, you will learn more about the hardware. Including these details as comments will help you remember later how your program works. See Example 3-2.

Example 3-2 Hardware Definitions

```
/* Hardware Definitions */
/*****************************************************************
 * Port AD bits:
 *   PAD-3 - PAD-0: Output: Scan row scan codes
 *   PAD-7 - PAD-4: Input: Column code
 *             |      Col3 Col2 Col1 Col0
 *     Row     | Col Code
 * Row Code    |1111 0111 1011 1101 1110
 * ------------|-------------------------------
 * 3  1110     |None  1    2    3    A  Key
 * 2  1101     |None  4    5    6    B  Pressed
 * 1  1011     |None  7    8    9    C
 * 0  0111     |None  *    0    #    D
 * ------------|-------------------------------
 *****************************************************************/
 /* Define Grayhill Series 96 4x4 keypad */
#define NUM_ROWS 4        /* Number of rows */
#define NUM_KEYS 16       /* Number of keys */
```

```
/* Define where they are connected to the microcontroller
 * PTAD Bit   Grayhill Keypad Pin
 *    0          1
 *    1          2
 *    2          3
 *    3          4
 *    5          6
 *    6          7
 *    7          8
 *****************************************************************/
```

Constant Definitions

Example 3-3 shows constants defined for this module. Note that it is not sufficient to merely define the constants. You must add comments stating how the constants are used in your program.

Example 3-3 Constant Definitions

```
/*****************************************************************/
/* Define constants */
#define ROW3 0x0e     /* Row 3 scan code */
#define ROW2 0x0d     /* Row 2 scan code */
#define ROW1 0x0b     /* Row 1 */
#define ROW0 0x07     /* Row 0 */
#define OUTPUTS 0x0f  /* Row outputs */
#define INPUTS 0xf0   /* Col inputs */
#define COL3 0x70     /* Col 3 scan code */
#define COL2 0xb0     /* Col 2 */
#define COL1 0xd0     /* Col 1 */
#define COL0 0xe0     /* Col 0 */
#define KEY_MASK 0xf0
#define NO_KEYS 0xf0  /* Code for no keys pressed */
#define END_MARK 0xff /* End of Good_Codes array */
/*****************************************************************/
```

Data Structures and Definitions

If you are using data structures in some way, document them with comments explaining their use. See Example 3-4.

Example 3-4 Constant Definitions

```
/*************************************************************/
/* Define arrays to store the scan codes, key codes, and a
 * lookup table for the return value */
unsigned char Row_Codes[] = {
  ROW3,     /* Row 3 scan code */
  ROW2,     /* Row 2 scan code */
  ROW1,     /* Row 1 */
  ROW0      /* Row 0 */
};
/*************************************************************
 * This lookup table contains the 8-bit scan codes for all
 * keys on the keypad
 ************************************************************/
unsigned char Good_Codes[ ] = {
  COL3 | ROW3,  /* "1" = 0x7e */
  COL2 | ROW3,  /* "2" = 0xbe */
  COL1 | ROW3,  /* "3" = 0xde */
  COL0 | ROW3,  /* "A" = 0xee */
  COL3 | ROW2,  /* "4" = 0x7d */
  COL2 | ROW2,  /* "5" = 0xbd */
  COL1 | ROW2,  /* "6" = 0xdd */
  COL0 | ROW2,  /* "B" = 0xed */
  COL3 | ROW1,  /* "7" = 0x7b */
  COL2 | ROW1,  /* "8" = 0xbb */
  COL1 | ROW1,  /* "9" = 0xdb */
  COL0 | ROW1,  /* "C" = 0xeb */
  COL3 | ROW0,  /* "*" = 0x77 */
  COL2 | ROW0,  /* "0" = 0xb7 */
  COL1 | ROW0,  /* "#" = 0xd7 */
  COL0 | ROW0,  /* "D" = 0xe7 */
  END_MARK      /* End marker */
};
/************************************************************
 * This lookup table returns the ASCII code for the key pressed.
 ***********************************************************/
/* User-defined key codes. These are ASCII. */
unsigned char Key_Codes[ ] = {
  "123A456B789C*0#D"
};
```

Variables

Every variable used in your program should include a comment defining its use. See Example 3-5.

Example 3-5 Variable Definitions

```
/***********************************************************/
unsigned char key_hit;   /* 8-bit ASCII code for the key or 0 */
                         /* if no key is pressed */
unsigned char col_code;  /* 4-bit col code from keypad scan */
unsigned char scan_code; /* 8-bit col_code, row_code */
unsigned char key_code;  /* 8-bit scan_code for key pressed */
unsigned char i;         /* An indexing variable */
/***********************************************************/
```

Code Comments

As you proceed to write the program code, structure it with blocks of design comments, straight from your design documentation, with code that follows implementing the design. You may find it useful to include code comments that explain how the code is working to implement the design. See Example 3-6.

Example 3-6 Comments in the Code

```
/************************************************************
 * Initialize your microcontroller's I/O port connected to the
 * keypad.
 ***********************************************************/
  /* Initialize the Port AD bits 3-0 for output */
  /* Check to see if the port has been set up */
  /*   IF the data direction register is not set
   *   to output on the ROW_OUT bits */
     if (( DDRAD & OUTPUTS ) != OUTPUTS ){
      /* Then initialize the data direction register,
       * enable the input pull-ups and enable the ATD
       * input bits */
       DDRAD |= OUTPUTS;  /* Set the Data Direction Register */
       PERAD |= INPUTS;   /* Pull ups enabled */
       ATDDIEN |= INPUTS; /* ATD inputs enabled */
     }
  /*********************************************************/
```

The comments in a program should document the design needed to implement the system as well as details that explain how the code implements the design. An overall program structure of a well-designed and well-documented program is shown in Example 3-7.

Example 3-7 Commented Code Outline

```
/************************************************************/
/*** Program Header ***/
/************************************************************/
/*** Hardware Definitions ***/
/************************************************************/
/*** Constant Definitions ***/
/************************************************************/
/*** Data Structures ***/
/************************************************************/
/*** Variables ***/
/************************************************************/
/*** Design Comment Block ***/
/************************************************************/
/****** Code to implement this design block ******/
/************************************************************/
/*** Design Comment Block ***/
/************************************************************/
/****** Code to implement this design block ******/
/************************************************************/
```

Comments in Assembly Language Programs

Many assembly language programmers tend to place comments at the end of each line of the assembly code, or on many lines. While better than omitting comments, this is not a good practice or commenting style. It leads to comments that are related to the line of code but not necessarily to the design or function the code is producing. Further, these comments are often added after the code has been written rather than before. A better way to comment an assembly language program is to follow the preceding strategy and use a block of comments to describe what the following block of code is to do. You may place comments on some lines of code to describe how they work. Comments are generally not needed on every line of assembly code.

3.12 Software Documentation

Each of the software development phases—design, coding, and testing—has associated documentation.

Software Requirements Specification (SRS)

The SRS is a document or series of documents defining what is required of the software. At the upper levels of the best-designed systems, the SRS should completely define what the user of the system is to see, that is, the user interface. This document can form the basis of the user

operator's manual. It must be written first and accepted by the customer and the software developer. As you continue with the lower levels of the design, where one starts to think about how things are going to be done, the SRS documentation begins to define the functions required by modules in the system. You should be able to give an SRS document to a colleague and have him or her code the function and return a working module to be included in the system.

Software Design Document (SDD)

The SDD is the document produced for the detailed design of a module. It defines the logic required to produce a particular function. You start with the SRS for the module and use a design tool such as pseudocode, described in Section 3.8.

Software Code

The coding phase has an element of software documentation. This means including comments in the software. We would like the code to be written so clearly that extra comments are not necessary. High-level languages allow us to do some of this, but rarely should we write a program without any extra comments describing what is going on. In assembly language programs, comments are mandatory because the language is not as design oriented as high-level languages. It is particularly effective to use the pseudocode produced for the SDD for the comments in an assembly language program.

Software Verification Plan (SVP)

The SVP is a document that describes how we are going to test and verify that a particular module or system meets its specifications. The SVP should give the details of limiting values to be tested and the expected results. There may be levels of SVPs associated with the various levels of our design.

User Manuals

The four document types just described are often treated as design documents to be used within the company and not delivered to the customer. Beyond these, there must be manuals for the customer's use. These include instructions on how to install the software (if appropriate) and instructions on using the software.

3.13 A Top-Down Design Example

As a final exercise, let us use the top-down design approach to tackle a design problem. As a review, the principal steps of top-down design are as follows:

- Understand the problem completely.
- Design in levels.
- Ensure correctness at each level.
- Postpone details.

Table 3-11 Problem Solution Logic

```
Alarm Sounds
    When the key is in the ignition
        and the motor is not running
            and the door is open
    When the key is in the ignition
        and the motor is running
            and the driver belt is not fastened
    When the key is in the ignition
            and the passenger seat is occupied
                and passenger belt is not fastened
    When the key is not in the ignition
        and the lights are on
```

- Successively refine your design.
- Design without using a programming language.

Seat Belt Alarm: Problem Statement

In many cars the seat belt alarm buzzer is also used to warn against leaving the key in the ignition or leaving the lights on. The following statement describes how such a system might operate:

> The alarm is to sound if the key is in the ignition when the door is open and the motor is not running, or if the lights are on when the key is not in the ignition, or if the driver belt is not fastened when the motor is running, or if the passenger seat is occupied and the passenger belt is not fastened when the motor is running.

The Top-Down Design

Understand the problem

It is often useful to restate the problem to understand it better. Often a tabular form, as shown in Table 3-11, can help clarify the logic needed.

First-Level Design

By reading the problem statement and perhaps restating it, we begin to understand the problem better; but we need a place to start the design. Table 3-11 lists circumstances under which the alarm is to sound if the key is in the ignition and other conditions for sounding the alarm when the key is not in the ignition. Our first cut at the design, for which we have used the pseudocode tool and postponed details, looks like this:

```
IF the key is in the ignition
THEN
 DO the alarms if the key is in the ignition
```

```
 ENDDO the alarms if the key is in the ignition
ELSE (the key is not in the ignition)
 DO the alarms if the key is not in the ignition
 ENDDO the alarms if the key is not in the ignition
ENDIF (the key is in the ignition)
```

Second-Level Design

We have obviously left out all the details that will sound the alarm, but we do have a starting structure to which we can now add details. First, though, we should look back at the design to make sure it is correct. Notice that we have put comments after the `ELSE` and `ENDIF` statements. These will help us keep track of where we are in the logic as we add details. There is not much logic in the design at this stage, so we continue with the second level and start to fill in some of the details. Details that are added in each of the following levels are shown in bold type.

```
IF the key is in the ignition
THEN
  DO the alarms if the key is in the ignition
    IF the motor is not running
    THEN
      DO the alarms if the motor is not running
      ENDDO the alarms if the motor is not running
    ELSE (the motor is running)
      DO the alarms if the motor is running
      ENDDO the alarms if the motor is running
    ENDIF (the motor is not running)
  ENDDO the alarms if the key is in the ignition
ELSE (the key is not in the ignition)
  DO the alarms if the key is not in the ignition
  ENDDO the alarms if the key is not in the ignition
ENDIF (the key is in the ignition)
```

Third-Level Design

Check back to ensure that the second-level design is correct, and continue adding details.

```
IF the key is in the ignition
THEN
  DO the alarms if the key is in the ignition
    IF the motor is not running
    THEN
      Do the alarms if the motor is not running
        IF the door is open
        THEN
          Sound the alarm
        ENDIF (the door is open)
      ENDDO the alarms if the motor is not running
    ELSE (the motor is running)
```

```
      DO the alarms if the motor is running
      ENDDO the alarms if the motor is running
    ENDIF (the motor is not running)
  ENDDO the alarms if the key is in the ignition
ELSE (the key is not in the ignition)
  DO the alarms if the key is not in the ignition
  ENDDO the alarms if the key is not in the ignition
ENDIF (the key is in the ignition)
```

Fourth-Level Design

Check back to ensure that the third-level design is correct, and continue adding details. You do not have to continue with the "key is in the ignition" logic if it makes sense to do something else. Let us add some details in the `ELSE (the key is not in the ignition)` part of the logic.

```
IF the key is in the ignition
THEN
  DO the alarms if the key is in the ignition
    IF the motor is not running
    THEN
      Do the alarms if the motor is not running
        IF the door is open
        THEN
          Sound the alarm
        ENDIF (the door is open)
      ENDDO the alarms if the motor is not running
    ELSE (the motor is running)
      DO the alarms if the motor is running
      ENDDO the alarms if the motor is running
    ENDIF (the motor is not running)
  ENDDO the alarms if the key is in the ignition
ELSE (the key is not in the ignition)
  DO the alarms if the key is not in the ignition
    IF the lights are on
      Sound the alarm
    ENDIF (the lights are on)
  ENDDO the alarms if the key is not in the ignition
ENDIF (the key is in the ignition)
```

Design for Successive Levels

We continue this design process by refining the design and adding the details needed to implement the solution. You may take several more design steps to complete the design.

```
IF the key is in the ignition
THEN
  DO the alarms if the key is in the ignition
    IF the motor is not running
    THEN
```

```
      Do the alarms if the motor is not running
        IF the door is open
        THEN
          Sound the alarm
        ENDIF (the door is open)
      ENDDO the alarms if the motor is not running
    ELSE (the motor is running)
      DO the alarms if the motor is running
        IF the driver's belt is not fastened
        THEN
          Sound the alarm
        ENDIF (the driver's belt is not fastened)
        IF the passenger seat is occupied
        THEN
          IF the passenger belt is not fastened
          THEN
            Sound the alarm
          ENDIF (the passenger belt is not fastened)
        ENDIF (the passenger seat is occupied)
      ENDDO the alarms if the motor is running
    ENDIF (the motor is not running)
  ENDDO the alarms if the key is in the ignition
ELSE (the key is not in the ignition)
  DO the alarms if the key is not in the ignition
    IF the lights are on
      Sound the alarm
    ENDIF (the lights are on)
  ENDDO the alarms if the key is not in the ignition
ENDIF (the key is in the ignition)
```

Final Check

Table 3-11 can be used to help check the final solution for correctness. Trace through your program logic for each of the cases that sound the alarm shown in Table 3-11. See Table 3-12.

3.14 Chapter Summary Points

It is vital that your software solutions be designed first and then written. Many problems (bugs) can be avoided by designing before writing. You must adopt a design practice such as the top-down methodology shown in this chapter.

This chapter has presented the following points.

- The top-down design method is our choice of design approaches.
- The top-down design steps are as follows:

  Understand the problem completely before writing code.

Table 3-12 Problem Solution Logic Final Check

Alarm Sounds	OK?
When the key is in the ignition	
and the motor is not running	
and the door is open	Yes
When the key is in the ignition	
and the motor is running	
and the driver belt is not fastened	Yes
When the key is in the ignition	
and the passenger seat is occupied	
and passenger belt is not fastened	Yes
When the key is not in the ignition	
and the lights are on	Yes

  Design in levels.
  Ensure correctness at each level.
  Postpone details.
  Successively refine your design.
  Design without using a programming language.

- With bottom-up design and coding, decisions at lower levels may adversely affect the upper levels of the design.
- In the real world, we try to follow the principles of top-down design, but we pragmatically use functions that have been already designed, coded, and tested.
- The elements of structured programming can be listed:
  Use three simple structures—sequence, decision, and repetition—to write all programs.
  Keep program segments small enough to be manageable.
  Organize the problem solution hierarchically (use top-down design).
  Use single-input, single-output program flow.
- The pseudocode technique is an effective design tool for all levels of top-down design.
- The top-down design method can lead to a top-down debugging and testing strategy where the structure of the design tests itself.
- Software documentation is a vital part of all stages of software development and consists of the following:
  Software requirements specifications (SRS)
  Software design documentation (SDD)
  Software code with comments
  Software verification plan (SVP)
  Users' manuals

3.15 Bibliography and Further Reading

Ganssle, J. G., *The Firmware Handbook, The Definitive Guide to Embedded Firmware Design and Applications*. Newnes/Elsevier, Burlington, MA, 2004.

Ganssle, J. G., *A Guide to Commenting*. The Ganssle Group, Baltimore, February 2006. http://www.ganssle.com/commenting.htm

3.16 Problems

Explore

3.1. List at least five principles of top-down design. [a, c]

3.2. What are the three basic elements of structured programming? [a]

3.3. Write the pseudocode and draw the flowchart symbol to represent the decision IF A is TRUE THEN B ELSE C. [a, c]

3.4. Write the pseudocode and draw the flowchart symbol to represent the decision IF A is TRUE THEN B. [a, c]

3.5. Write the pseudocode and draw the flowchart symbol to represent the repetition WHILE A is TRUE DO B. [a, c]

3.6. Write the pseudocode and draw the flowchart symbol to represent the repetition DO B WHILE A is TRUE. [a, c]

Stimulate

3.7. Use structured flowcharts or pseudocode to write a design that will implement the following problem description: [c]

  Prompt for and input a character from a user at the keyboard.
  If the character is alphabetic and is uppercase, change it to lowercase and output it to the screen.
  If the character is alphabetic and is lowercase, change it to uppercase and output it to the screen.
  If the character is numeric, output it with no change.
  If it is any other character, beep the bell.
  Repeat this process until an ESC character is typed by the user.

Challenge

3.8. Design a program that initializes an 8-bit data storage accumulator to 0 and then inputs 10_{10} successive 8-bit values from an input device located at address 0x70, adding each of them to the 8-bit data storage accumulator. If during this process an unsigned binary overflow occurs, print an error message and repeat from the beginning. Otherwise, after the 10 values have been input and added, output the result to an output device at location 0x71. Run the process forever. Your design must be a structured design and must show REPETITION, DECISION, and SEQUENCE. [c]

3.9. Use structured pseudocode to give a design that will accomplish the following: [c]

A user is to input a character to select one of three processes. Valid characters are A, B, and C, where A, B, and C select processes A, B, or C, respectively. Process A requires a byte of information to be input from an A/D converter, which it then converts to a integer decimal number in the range of 0 to 5 and displays it on the screen. Processes B and C are not defined at this stage. Prompts and error messages are to be displayed. You do not have to give details of the decimal conversion required in Process A.

3.10. Use structured pseudocode to give a design that will accomplish the following: [c]

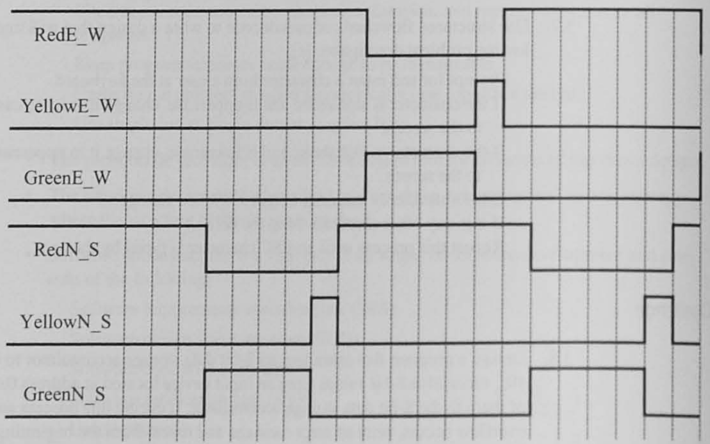
A byte of data is to be input from an analog-to-digital converter, and a critical value is to be input from a set of switches. If the A/D value is greater than the critical value, the microcontroller is to sound an alarm. Otherwise the alarm is to be turned off. This process is to continue forever.

3.11. Design a traffic light controller. [c]

Imagine an intersection with north/south and east/west streets. There are to be six traffic light signals:

RedE_W, YellowE_W, GreenE_W
RedN_S, YellowN_S, GreenN_S

Assume that the time elements in the table below are 10 seconds and that a timer delay is available as a function or subroutine. Give the pseudocode structured design for the light controller.

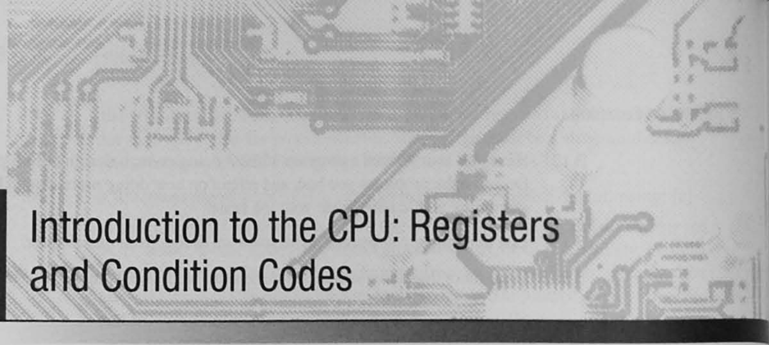


Reflect on Learning

3.12. Have you ever written a program without doing enough design before programming? Describe the problems you had, and reflect on how doing more design before programming would have made your job easier.

3.13. List five things you learned about software design from this chapter.

3.14. In no more than three sentences, summarize what you learned about top-down design.



4 Introduction to the CPU: Registers and Condition Codes

Objectives

This chapter starts your learning about real processors. The steps you take here will be the same ones taken for processors you will meet in your career. We will begin with the registers, that make up what is known as the programmer's model, emphasizing the condition code register.

4.1 Introduction

In learning about a microcontroller or microprocessor, you first evaluate it by looking at the hardware resources. At the basic level, these include the registers in the CPU such as accumulators, memory addressing registers, and the condition code register. For most microcontrollers, the CPU contains other hardware resources such as timers, parallel and serial I/O, and analog-to-digital convertors.

4.2 CPU Registers

The central processor unit, the CPU, contains the registers used in your programs. Depending on the design of the processor, the registers may have 8, 16, 32, or more bits; in any CPU there are registers of several different types.

Accumulators: Accumulators are registers that accumulate answers, such as the A register in the picocontroller of Chapter 2. An accumulator can serve simultaneously as the source register for one operand and the destination for an ALU operation.

Condition code register: The condition code register is also called the flags or status register. It holds condition code bits generated by the processor when instructions are executed.

Doubled registers: The number of bits in a register depends on the general architecture of the CPU. An 8-bit CPU generally has 8-bit data registers. Sometimes two of the data registers are used together to double the number of bits.

General-Purpose registers: These registers hold data; they serve as source and destination operands for data transfer instructions, and as sources for ALU operations.

Index registers: Index registers are used to address memory also. Unlike pointer registers, the memory address is found by adding to the contents of the index register a constant value, often called an offset. The resulting sum, called the *effective address,* is the address generated by the CPU to retrieve or store data. For a pointer register, the effective address is just the contents of the register.

Pointer registers: Pointer registers address memory. The register is said to "point" to a memory location. In most processors, pointer registers can be incremented or decremented, either by a program step or automatically after their use.

Segment registers: In some machines, depending on how memory addressing is organized, the physical address consists of two parts: a segment part, which defines a certain area or page in the memory, and an offset part, which specifies a particular place in the page.

Stack pointer register: This is a pointer register dedicated to addressing memory used for variable data and subroutine return address storage.

4.3 Register Transfers

Many instructions in any computer involve the transfer of information. Sometimes the information is just transferred from one place to another, as in a `MOV A,B` instruction. Here, the A register is the *source* of the information and the B register is the *destination*. Other instructions may modify the information along the way. For example, `ABA` in Chapter 2 will add the contents of the A and B registers and place the result in the A register. A transfer never destroys or changes the source operand in a register transfer instructions, unless a source register is also used as the destination, as in the `ABA` instruction.

A *register transfer language* succinctly describes what the instruction accomplishes.

Manufacturers provide a symbolic notation that precisely and succinctly describes the operation of each instruction. This is sometimes called a *register transfer language*. Typically, a register name in parentheses means that the operation involves the contents of the register. A left arrow (←) denotes a replacement operation. For example, an instruction that replaces the contents of the A register with the contents of the B register has the symbolic notation (A) ← (B). Table 4-1 shows examples of register transfer language statements.

Table 4-1 Examples of Register Transfer Language

Destination Register ← Source Register
(A) ← (B)
A ← B
Source Register → Destination Register
(B) → A
B → A

Exercise 4-1

Turn now to the material that describes what CPU registers are available in your laboratory processor. Be sure to note which registers are accumulators, which are general-purpose registers, and which address memory; then return here to learn more about the condition code register.

4.4 The Condition Code Register

The condition code register has bits that are used to make decisions in the program with condition branch instructions.

The condition code register, also called the flags or status register, has bits that are modified (set or reset) when the computer executes an instruction involving data. Usually arithmetic and logic (ALU) operations modify one or more of the flags. Sometimes a data transfer, like a load or move operation, modifies the flags too; most processors have instructions that can directly set or reset the bits. Let us look at the bits that may be found in a condition code register and understand how to interpret them.

The Carry Bit

The carry bit is set if there is a carry, or borrow, out of the most significant bit during an addition or subtraction. Consequently, this bit is sometimes called the carry/borrow bit. For example, if we add or subtract the numbers shown in Example 4-1, the carry/borrow bit is set (= 1) by each operation.

Example 4-1 Addition or Subtraction of 8-Bit Unsigned Binary Numbers

```
Addition                              Subtraction
  147      1 0 0 1 0 0 1 1              147      1 0 0 1 0 0 1 1
+ 179    + 1 0 1 1 0 0 1 1            - 179      1 0 1 1 0 0 1 1
  ---      ---------------              ---      ---------------
  326    1 0 1 0 0 0 1 1 0            -  32    1 1 1 1 0 0 0 0 0
         ↑                                     ↑
         Carry                                 Borrow
```

What does the carry bit being set (or reset) mean? How do we use this information? *It depends on the code.* The meaning of *any* information *always* depends on the code. If the code is unsigned binary, as in Example 4-1, the presence of the carry bit = 1 means that an *overflow* has occurred. Let us define overflow.

An overflow *occurs when the result of an arithmetic operation cannot be represented by the number of bits available. This could mean the result is too large or too small, although the latter is sometimes called* underflow.

An *overflow* indication means there is an error in the result.

In Example 4-1 the addition of 179 + 147 = 326, a result that is too large for an 8-bit, unsigned binary number (whose maximum is 255). Further, 147 − 179 = −32, and −32 *cannot* be represented with an unsigned binary code. Thus, if we are using the unsigned binary code to represent numbers, the carry bit is set for errors such as overflow or a negative result.

The carry bit can be used for multiple byte addition or subtraction. Consider adding two 16-bit, unsigned binary (or two's-complement) numbers, but add them one byte at a time. In Example 4-2 the carry out from the addition of the least significant bytes is added into the addition of the most significant bytes. Microcontrollers have special instructions for this.

Example 4-2 Use of the Carry Bit in Multiple-Byte Arithmetic

Two 16-bit numbers to be added. For example,

```
0 0 1 1 0 0 1   0 1 1 0 0 1 0 0 1
0 0 0 1 1 0 1   1 1 0 1 1 0 1 1 0
---------------------------------
```

can be added with two 8-bit byte additions:

```
    Most significant byte           Least significant byte (this addition is done first)

        Carry in from least significant byte
                        ↓
                        1  ←————————  1 ← Carry out of least significant byte

    0 0 1 1 0 0 1 0                 1 1 0 0 1 0 0 1
    0 0 0 1 1 0 1 1                 1 0 1 1 0 1 1 0
    ---------------                 ---------------
0   0 1 0 0 1 1 1 0                 1 0 1 1 1 1 1 1
↑
Carry out of most significant byte
```

To use the carry bit to detect arithmetic errors, say when an overflow has occurred, as shown in Example 4-1, your processor will have instructions that test the carry bit as part of the instruction execution. These instructions will be similar to *branch-if-carry-set* and its complement *branch-if-carry-clear*. The program flow for a conditional branch instruction is like the single-sided decision structure shown in Figure 3-6. If the condition is true, the branch is taken; otherwise, the program continues. See Example 4-3.

Example 4-3 Conditional Branching

Show how to branch to an error handling routine if an overflow occurs when adding two 8-bit unsigned numbers.

Solution

```
Load the first number
Add the second number to the first
Branch-if-carry-set to the overflow handler
Otherwise, continue with the program
```

Example 4-2 shows arithmetic operations that use the carry bit to flow from one addition to another. Instructions that do this are similar to *add-with-carry* and *subtract-with-carry*. When you are subtracting, the carry bit indicates a borrow is needed; some instruction sets call these instructions *subtract-with-borrow*.

Exercise 4-2

Investigate your microcontroller's instruction set and list the instructions that test or use the carry bit.

Two's-Complement Overflow Bit

What does the carry bit mean if the numbers to be added or subtracted are encoded with an 8-bit, two's-complement code? Look at Example 4-4.

Example 4-4 Addition or Subtraction of Two's-Complement Numbers

```
Addition                                          Subtraction
   - 109      1 0 0 1 0 0 1 1          - 109       1 0 0 1 0 0 1 1
+  - 179    + 1 0 1 1 0 0 1 1          -(-77)      1 0 1 1 0 0 1 1
   - 288    0 0 1 0 0 0 1 1 0          -  32     1 1 1 1 0 0 0 0 0
            ↑                                    ↑
            Carry                                Borrow
```

Overflow that occurs when doing signed arithmetic with two's-complement numbers is detected by a *two's-complement overflow* bit.

The binary operands and the binary result in Example 4-4 are the same as Example 4-1. In Example 4-1 the sum of 147 and 179 is 326, which is larger than the largest 8-bit unsigned number (255). Thus, the carry bit indicates an overflow. In Example 4-4 the sum of –109 and –77 is more negative (larger) than the most negative number we can represent with a two's-complement binary code (the most negative is –128).

Therefore, it appears that the carry bit will allow us to detect an overflow. However, when –77 is subtracted from –109, the answer, –32, is encoded correctly (1 1 1 0 0 0 0 0 is the two's-complement binary code for –32). No overflow has occurred even though the carry bit is set. In one case, the carry bit indicates an overflow, and in the other it does not! We conclude that when two's-complement codes are used for addition or subtraction of signed numbers, the carry bit *cannot* be used to indicate an overflow. Fortunately, microcomputer manufacturers include a bit to be set if a *two's-complement overflow* has occurred (or reset if it has not). There are several algorithms for overflow; probably the easiest to understand (but not the easiest to do in hardware) is the following, which we use when two's-complement numbers are being added.

Two's-complement overflow occurs if the two operands have the same sign AND the sign of the result is different.
Two's-complement overflow cannot occur if the two operands have opposite signs.

When two's-complement numbers are being subtracted, take the two's complement of the subtrahend and proceed as in addition.

Example 4-1 and 4-4 show the binary numbers, and the results are identical for each addition and subtraction. The hardware to do addition and subtraction is the same in each case. This is the beauty of the two's-complement code. The binary result and the potential for overflow can be interpreted correctly by the program because the hardware provides the carry bit for unsigned numbers and the two's-complement overflow bit to be tested when two's-complement codes are being used.

Your microcontroller has instructions that test the two's-complement overflow bit similar to those that test the unsigned overflow carry bit. They will likely be called *branch-if-overflow-set* and *branch-if-overflow clear*. You would use the *branch-if-overflow-set* instruction to branch to the error handler if the data in Example 4-3 were adding two's-complement data.

Exercise 4-3

Investigate your microcontroller's instruction set and list the conditional branch instructions that test the two's-complement overflow bit.

Sign Bit

The *sign bit* is equal to the most significant bit of the result; it gives the sign *only* if signed number codes are being used.

The sign bit shows that an ALU (or other) operation gave a result in which the most significant bit is a 1 (or a 0). Notice we did not say "resulted in a negative (1) or positive (0) number," since the meaning of the bit depends on the code. In an unsigned binary number computation, there can be no negative result because there are no codes for negative numbers. The sign bit *means* negative *only* if ones'-complement or two's-complement codes are being used to represent signed information. In some microcontrollers the sign bit is called the *negative* bit. Instructions in your microcontroller that test the sign bit will be similar to *branch-minus* and *branch-plus*.

Exercise 4-4

Investigate your microcontroller's instruction set and list the conditional branch instructions that test the sign bit.

Zero Bit

The zero bit is true, or set, if the result of an operation is equal to zero. Otherwise, it is false, or reset. The zero bit conditional branch instructions are those similar to *branch-equal(-to-zero)* and *branch-not-equal(to-zero)*.

Exercise 4-5

Investigate your microcontroller's instruction set and list the conditional branch instructions that test the zero bit.

Parity Bit

Some processors have a bit that shows if a result has even or odd parity. An even parity bit is set if the result has even parity, that is, an even number of 1s. An odd parity bit is set for a result with an odd number of 1s. The parity bit, along with conditional branching instructions for parity-even and parity-odd, is useful for checking to see if errors have occurred in data transmitted over long distances. We will learn more about parity when we discuss serial I/O in Chapter 12. Not all microcontrollers have a parity bit in the condition code register. If yours does, you may see instructions such as *branch-parity-odd* and *branch-parity-even*.

Other Condition Code Register Bits

There may be other bits in the condition code register that are not directly related to conditional branching. These typically include bits to control the interrupt capabilities. We will study these later when we discuss interrupts.

How Do the Bits Get Set or Reset?

The condition code register bits are modified by hardware during the execution of some instructions, usually ALU instructions that modify the data in some way. The bits are set or

reset according to the hardware regardless of the code you are using in the computation. See Examples 4-5 through 4-8.

Example 4-5

Give an example showing the addition of two binary numbers that result in:

(a) Overflow if the numbers are unsigned binary but no overflow if they are two's-complement binary.

(b) No overflow if the numbers are unsigned binary but overflow if they are two's-complement binary.

Solution

(a)

	Unsigned Value	Signed Value
1 1 1 1 1 1 1 1	255	-1
0 0 0 0 0 0 0 1	+1	+1
0 0 0 0 0 0 0 0	0	0
	Overflow	No overflow
	Carry bit = 1	Two's-complement overflow bit = 0

(b)

	Unsigned Value	Signed Value
0 1 1 1 1 1 1 1	127	+127
0 0 0 0 0 0 0 1	+1	+1
1 0 0 0 0 0 0 0	128	-128
	No overflow	Overflow
	Carry bit = 0	Two's-complement overflow bit = 1

Example 4-6

Do the following binary additions and show what the carry (C), two's-complement overflow (V), sign (S), and zero (Z) bits are after the addition.

```
1 0 1 0 1 1 0 1     1 0 1 0 1 1 0 1     1 0 1 0 1 1 0 1
1 0 1 1 0 0 1 0     0 1 0 0 1 1 0 1     0 1 0 1 0 0 1 1
```

Solution

```
0 1 0 1 1 1 1 1     1 1 1 1 1 0 1 0     0 0 0 0 0 0 0 0
C = 1,   V = 1      C = 0,   V = 0      C = 1,   V = 0
S = 0,   Z = 0      S = 1,   Z = 0      S = 0,   Z = 1
```

Example 4-7

For each of the 8-bit binary additions shown in Example 4-6, assume that the data are unsigned binary numbers. Give the decimal equivalents of each operand and the answer, and state whether overflow has occurred.

Solution

```
1 0 1 0 1 1 0 1   173   1 0 1 0 1 1 0 1   173   1 0 1 0 1 1 0 1   173
1 0 1 1 0 0 1 0   178   0 1 0 0 1 1 0 1    77   0 1 0 1 0 0 1 1    83
0 1 0 1 1 1 1 1    95   1 1 1 1 1 0 1 0   250   0 0 0 0 0 0 0 0     0
Overflow                No overflow             Overflow
```

Example 4-8

For each of the 8-bit binary additions shown in Example 4-6, assume that the data are two's-complement binary numbers. Give the decimal equivalents of each operand and the answer, and state whether two's-complement overflow has occurred.

Solution

```
1 0 1 0 1 1 0 1   -83   1 0 1 0 1 1 0 1   -83   1 0 1 0 1 1 0 1   -83
1 0 1 1 0 0 1 0   -79   0 1 0 0 1 1 0 1    77   0 1 0 1 0 0 1 1    83
0 1 0 1 1 1 1 1    95   1 1 1 1 1 0 1 0   - 6   0 0 0 0 0 0 0 0     0
Overflow                No overflow             No overflow
```

Complex Conditional Branch Instructions

The conditional branch instructions illustrated so far test and branch based on whether a single condition code register bit is set or reset. More complex conditional branch instructions may be found also. For example, consider comparing two signed (two's-complement) binary numbers to determine which is the greater. This cannot be done with a simple *branch-if-carry*, *-overflow*, *-sign*, or *zero* instruction. If, however, the two numbers are subtracted (to set or reset the condition code register bits, usually done with a *compare* instruction), the logic that tells if the minuend (A, the first number) is greater than the subtrahend (B, the second) is as follows.

If the (zero bit OR (sign bit EXCLUSIVE-OR overflow bit)) = 0 then the minuend is greater than the subtrahend (Figure 4-1).

The conditional branch instruction that performs this logic is likely called *branch-greater-than*. Its complementary instruction is *branch-less-than-or-equal.*

This logic and the conditional branch instructions work only if the data are signed numbers in two's-complement code. An equivalent set of instructions with different logic must be used

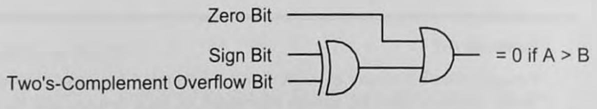


Figure 4-1 Signed greater-than logic.

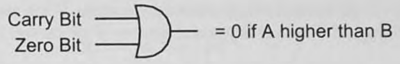


Figure 4-2 Unsigned higher-than logic.

for unsigned data. The logic fix detecting which is the *higher* when two unsigned numbers are being subtracted (or compared) is as follows:

If the (carry bit OR zero bit) = 0, then the minuend is higher than the subtrahend (see Figure 4-2).

You must use the correct conditional branch instruction following signed or unsigned arithmetic.

Your microcontroller will have instructions similar to *branch-higher* or *branch-lower-or-same* to be used for unsigned numbers. Notice the difference in terminology to distinguish between signed and unsigned data conditional branches. Terms like *great-than* and *less-than* are used for the signed numbers. *Higher-than* and *lower-than* denote unsigned data. You must be very careful to choose the correct instruction depending on whether you are doing signed or unsigned arithmetic.

Exercise 4-6

Investigate your microcontroller's instruction set and list the conditional branch instructions that allow you to compare signed data.

Exercise 4-7

Investigate your microcontroller's instruction set and list the conditional branch instructions that allow you to compare unsigned data.

Using the Condition Code Register

The condition code register (or flags register) is attached to the sequence controller for use by the conditional branch instructions. With these we can answer questions like the following:

Is bit zero on the I/O port equal to one?

Are the contents of the A register greater than those of the B register?

Is the sign of the result minus?

Has an overflow error occurred?

Notice that the answer to each of these questions must be yes or no. When we write programs, we would like to do one thing if the answer is yes and another if the answer is no.

4.5 The Programmer's Model

The *programmer's model* is the set of registers that the programmer can manipulate and must manage during the programming of the processor. It includes the accumulators and data registers, the memory addressing registers, the stack pointer register, and the condition code register. As we will see when we learn more about assembly language programming, the programmer is also responsible for selecting the memory locations used for data storage.

Exercise 4-8

Draw the programmer's model for your microcontroller showing all registers.

4.6 Conclusion and Chapter Summary Points

- The CPU contains a variety of registers. Some are data registers and accumulators, and some are used for addressing memory.
- Manufacturers use a register transfer language to describe each operation in the instruction set.
- The condition code register contains bits that are modified when various instructions, generally ALU instructions, are executed.
- Among the bits found in the condition code register are bits that indicate a carry, a two's-complement overflow, sign, a zero, and parity.
- The condition code register bits are used by conditional branch instructions to allow yes/no decisions to be made.
- The programmer's model includes the registers the programmer is responsible for managing during the program.

4.7 Problems

Explore

4.1 List the CPU registers available in the microcontroller you are studying.

4.2 In Example 4-1, what is the decimal result of the unsigned addition? Of the subtraction? [a]

4.3 In Example 4-4, what is the 8-bit result of the two's-complement binary addition? [a]

4.4 What is overflow? [a]

4.5 What is the meaning of sign bit = 1 when unsigned binary coded numbers are added? [a]

4.6 What is the meaning of sign bit = 1 when two's-complement binary coded numbers are added? [a]

4.7 What is the meaning of carry bit = 1 when unsigned binary coded numbers are added? [a]

4.8 What is the meaning of carry bit = 1 when two's-complement binary coded numbers are added? [a]

4.9 What is the meaning of zero bit = 1 when unsigned binary coded numbers are added? [a]

4.10 What is the meaning of zero bit = 0 when two's-complement binary coded numbers are added? [a]

4.11 What is the meaning of two's-complement overflow bit = 1 when unsigned binary coded numbers are added? [a]

4.12 What is the meaning of two's-complement overflow bit = 1 when two's-complement binary coded numbers are added? [a]

Stimulate

4.13 Show by example that two's-complement overflow cannot occur when numbers of opposite sign are added. [b]

4.14 Show by example that two's-complement overflow can occur when the numbers of the same sign are added. [b]

4.15 Do the following 8-bit binary additions, and for each case give the expected result in the carry, zero, sign and overflow flags. [a]

```
a.  1010 0011     b.  1111 1111     c.  0111 0001
   +0011 1011        +0000 0001        +0100 0000

d.  1010 0010     e.  0111 1111     f.  1010 1010
   +1000 0000        +1000 0000        +0101 0101
```

4.16 For Problem 4.15, assume that the binary numbers are in unsigned binary code. Show the equivalent decimal arithmetic operations and indicate whether overflow has occurred. [a]

4.17 For Problem 4.15, assume that the binary numbers are in two's-complement binary code. Show the equivalent decimal arithmetic operations and indicate if overflow has occurred. [a]

Challenge

4.18 Imagine that you have two 8-bit numbers in two registers (A7:A0 and B7:B0) that are to be added together and that the 8-bit output of the arithmetic and logic unit (ALU7:ALU0) and a carry bit (ALUCY) are available. What digital combinational logic hardware would be needed to produce a logic signal that is asserted high for the following conditions: [c]

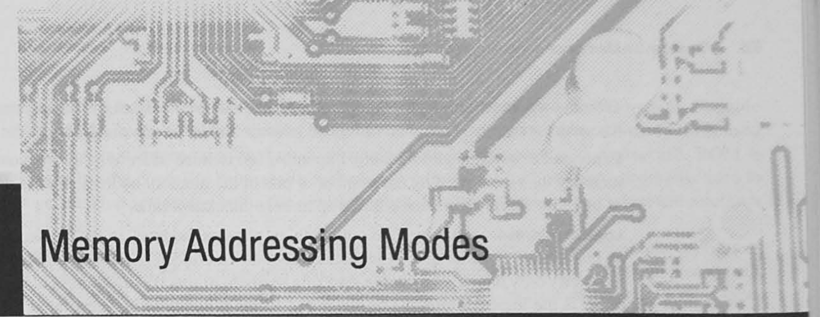
a. A carry has been produced by the addition.
b. The addition results in a two's-complement overflow.
c. The result of the addition is zero.
d. The result of the addition is a negative number.

4.19 For the multibyte addition shown in Example 4-2, state what kind of instruction you would expect the microcontroller to have to be able to do this. [e]

Reflect on Learning

4.20 What was the most useful thing you learned from this chapter?

4.21 List three concepts that you found important in this chapter and explain what they mean to you.



5 Memory Addressing Modes

Objectives

This chapter covers the basic principles for accessing data by describing the various ways your microcontroller instruction set addresses memory. If you are going to program your microcontroller in assembly language, you need to know these to be able to write efficient programs.

5.1 Introduction

The instruction set of a real processor has only a few categories of instructions, such as data transfers, arithmetic and logic operations, and branch and control instructions. Many instructions use one or more operands in registers or memory and often have several ways to address them. The different ways an instruction can specify operands are called *addressing modes*; if you learn these, along with the few categories of instructions, you will soon be writing assembly language programs.

In this chapter you will learn a variety of addressing modes that improve the efficiency of a CPU's operation either by allowing fewer bits to encode the instruction, by letting the CPU execute instructions faster, or both. In addition, some modes may allow instructions that can calculate an address at the time the program is running. For example, if you know the start of table of data and want to step through the table, you can calculate the next address by adding the number of bytes for each data element to the current address. In some computers, an address can be specified relative to the program counter. This is useful for branch instructions that do not branch very far from the current instruction.

5.2 Addressing Terminology

Auto increment and auto decrement: In some systems, registers that address memory can be incremented or decremented automatically during use. This feature provides very efficient addressing for stepping through tables of data.

Effective address: This term refers to an address that is calculated by the processor. The effective address may be a physical or logical address and is the actual address of the operand.

Expanded or extended addressing: Expanded or extended addressing allows you to have more memory than is allowed by the number of bits in the memory address. This is done by blocks of memory sharing the same address space, one block at a time.

Logical address: Sometimes the complete, or physical address, is not needed or provided by an instruction. For example, in segmented memory architectures as discussed shortly, we need to specify only the offset from the start of a segment to specify the address of an operand. This offset is the logical address. The physical address is computed or generated from the logical address and other segment information, depending on the memory architecture used.

Memory and I/O maps: A memory or I/O map shows what addresses are used for what purposes. A memory map may show which addresses contain ROM and which contain RAM, as well as any that have no memory installed at all.

Offset address: An offset address is one that is calculated from the start of a segment of memory or from a specified location in memory.

Physical address: The physical address is the actual address that must be supplied to the memory. The number of bits in the physical address fixes the maximum number of memory locations that can be addressed.

RAM: We can read from or write to random access memory.

Relative address: A relative address is found by adding an offset address to the current contents of the program counter.

ROM: Read-only memory can only be read from.

Segment address: A segment address gives the location of a block or segment of memory that is smaller than the full physical memory.

Stack: The stack is an area of RAM that is reserved for temporary data storage.

5.3 Memory Types

All computers have both *RAM* and *ROM.*

The computer system designer has available two types of memory, RAM and ROM. Every computer system has both types; the choice of how much of each type, and its location in the memory map, depends on the computer system being designed. *Random access memory*, or RAM, may be read from and written to. The semiconductor RAM used in systems today is volatile. Anything stored in memory is lost when the power is removed.

ROM, on the other hand, is *read-only memory*. Once programmed, either at the integrated circuit factory as part of the manufacturing process, or in the field, for field-programmable devices, it can only be read. ROM is nonvolatile; it retains its information when the power is turned off.

5.4 Computer Types and Memory Maps

We will distinguish between two general types of computer. These are (1) *desktop* or multiple-application systems and (2) *embedded* systems. Both have RAM and ROM, and

RAM is used for variable information. In desktop, multiple-application systems, variable information can be both data used by programs and the programs themselves. ROM is used for constant information that must be retained while the power is disconnected. ROM is used for the complete program in embedded systems and for "boot-up" programs used to get desktop systems going in the morning when we turn the power on or when the computer is reset.

The amount of RAM and ROM depends on the type of system.

You are probably most familiar with desktop multiple-application computers like personal computers. Embedded system computers, which do a single task or set of tasks, include microcontrollers in vending machines and the computer that controls the fuel injection system in an automobile. These systems are very different from the desktop system even though the CPU, memory, and I/O concepts learned in the preceding chapters apply equally to both. The amount of RAM and ROM in these systems and their use of input and output devices distinguishes one from another.

The Desktop Computer System

Desktop systems use ROM for the basic I/O software and large amounts of RAM for programs and data.

Figure 5-1 shows a desktop system. It has a powerful CPU, copious amounts of RAM for programs, and some ROM for the boot-up code and low-level system I/O drivers. There are disk and CD/DVD systems for program and data storage, and human-oriented I/O such as a keyboard, liquid crystal display, printer, and user I/O ports.

Many application programs, including word processors, spread sheets, assemblers, compilers, and debuggers run on these systems. All are loaded into the RAM memory from the disk by a *disk operating system*. There is an additional component of software in ROM. This code is the *basic input/output software*, or *BIOS*. It loads, or *bootstraps*, the operating system from the disk before executing other programs. The memory map of a desktop system is shown

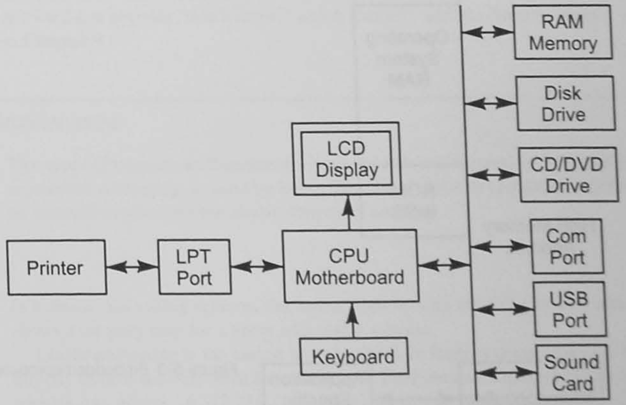


Figure 5-1 Desktop computer system.

in Figure 5-2. We see that most of the memory is RAM and is used for the operating system resident code and for application programs.

The Embedded System

An embedded system contains much more ROM for the program and less RAM for data storage than we find in desktop computers.

An embedded system (Figure 5-3), is one in which the computer is designed to do some particular job or jobs. Embedded systems differ from desktop systems in the following ways:

- They contain the least amount of hardware to accomplish the job at the least cost.
- Unless it is part of the application, there is little or no human-oriented I/O such as displays keyboards.
- The program is in ROM. There is no disk system from which the program can be loaded.
- Only data variables and the stack are in RAM.

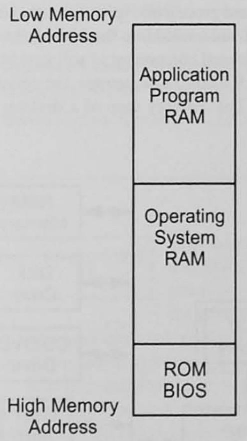


Figure 5-2 Desktop computer memory map.

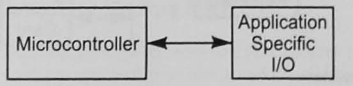
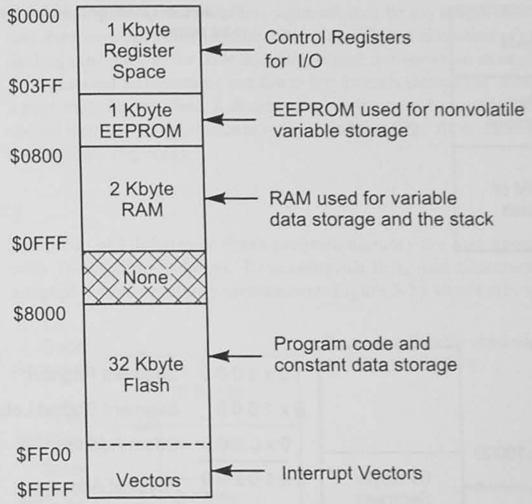


Figure 5-3 Embedded computer system.



$0000
1 Kbyte Register Space
Control Registers for I/O
$03FF
1 Kbyte EEPROM
EEPROM used for nonvolatile variable storage
$0800
2 Kbyte RAM
RAM used for variable data storage and the stack
$0FFF
None
$8000
32 Kbyte Flash
Program code and constant data storage
$FF00
Vectors
Interrupt Vectors
$FFFF

Figure 5-4 Embedded system memory map.

Figure 5-4 shows the memory map for an embedded system; it includes only enough RAM and ROM to do the job. The entire memory map does not have to be filled; if memory is not needed, it is not included in the system. The system designer gains an additional benefit from this other than just reducing the cost of the memory. Memory addresses that are not used can become "don't cares," which simplify address decoder design, as we will see in Chapter 9.

5.5 Memory Architectures

Two types of memory architecture are linear addressing, favored by Freescale processors, and segmented addressing, as used by Intel. The type of memory architecture directly affects how an instruction generates the physical memory address.

Linear Addressing

In a *linear addressing* scheme, the instructions specify the full physical address. Figure 5-5 shows a memory map for a linear addressing scheme.

Linear addressing is the easiest to understand. In large systems, however, instructions that directly address memory must have many bits. The Freescale ColdFire microcontroller's 24-bit address can access 16,777,216 (16 Mbyte) locations. An Intel Pentium with a 4-gigabyte address space requires a full 32-bit address.

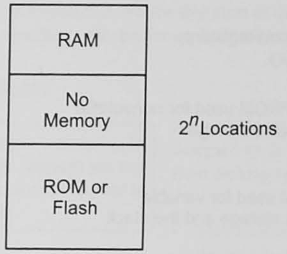


Figure 5-5 Linear addressing memory map for an *n*-bit address.

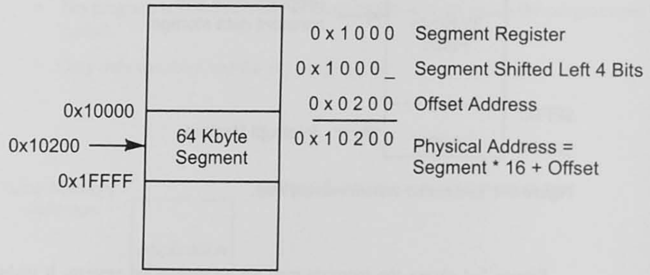

0x1000 Segment Register
0x1000_ Segment Shifted Left 4 Bits
0x0200 Offset Address
0x10200 Physical Address = Segment * 16 + Offset
0x10000
0x10200
0x1FFFF
64 Kbyte Segment

Figure 5-6 Intel segmented memory addressing.

Segmented Addressing

Segmented addressing allows an instruction to carry fewer address bits than are needed for the full physical address.

As the amount of memory that the processor can address increases, so does the number of bits needed to form the physical address. This means that each instruction must contain more bits, and more memory is needed for the program. The segmented memory architecture offers one way around this dilemma.

The Intel segmented addressing scheme uses a *movable segment architecture*. For example, a total memory space of one megabyte can be organized into segments, or blocks of memory. These segments may range in size from 16 bytes to 64 Kbyte. Figure 5-6 shows a one-megabyte memory. The full 20-bit physical address consists of a *segment address* and an *offset address*. Because segment addresses are maintained in separate *segment registers*, the program counter and other memory addressing registers can be only 16 bits. Each memory reference instruction generates a 16-bit offset that is added to a 16-bit segment register. The CPU constructs the physical address by shifting the segment register contents left 4 bits and then adding the offset. In this way the 64 Kbyte segment can be located on any 16-byte boundary. All memory reference addresses are generated as shown in Figure 5-6.

In this segmented architecture, segments may be any length from 16 bytes to 64 Kbyte and may even overlap. This flexibility allows an efficient allocation of memory to various parts of the program, such as for code and data. Figure 5-7 shows an example.

Segmented architectures use fewer bits in each instruction because only the offset within a page must be specified. A disadvantage is the need for special programming techniques or special instructions to cross over a page boundary or to allow data elements that are larger than the 64 Kbyte segments.

Expansion Memory

Hundreds of kilobytes of Flash program memory are now appearing in microcontrollers with 16-bit address buses. To accomplish this, manufacturers such as Freescale have adopted a *paged* memory architecture. Figure 5-8a shows this with four 16 Kbyte pages.

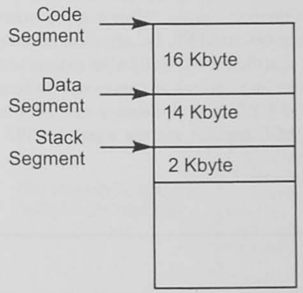


Figure 5-7 Variable sized segments used in a segmented memory architecture.

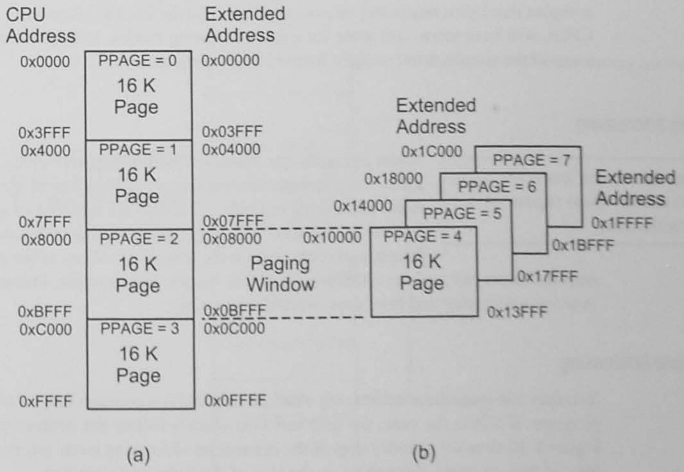


Figure 5-8 Expansion memory.

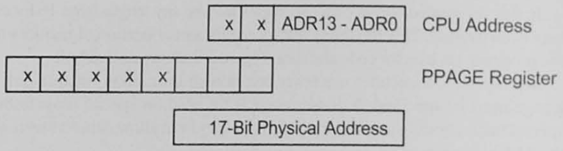


Figure 5-9 Expansion memory address.

The 16-bit CPU address is sufficient to address any location in this 64 Kbyte space. Figure 5-8b shows four additional 16 Kbyte memory pages to complete a total of 128 Kbyte. The CPU uses a register called the PPAGE register to create the 17-bit physical address needed for these expansion memory pages. Whenever an instruction generates an address in the paging window, 0x8000–0xBFFF, the physical address is generated as shown in Figure 5-9. The three least significant bits of PPAGE extend bits ADR13–ADR0 from the CPU address. Usually special instructions are necessary to access the expanded memory. Because this example is for a 128 Kbyte memory microcontroller, the higher bits (signified by the x's) in the PPAGE register are not used. A CPU design that uses these bits can add more memory.

5.6 Addressing Modes

You do not have to learn many addressing modes to program most microcontrollers. The more complex microprocessors and microcontrollers, like the Intel Pentium and Freescale ColdFire CPUs, will have more, and more complex, addressing modes. In this section we will explain some of the simple, more straight forward addressing modes.

Segr

Seg
inst
thar
add

Register Addressing

Register addressing needs only a few bits to define which register(s) are used for the data.

When operands are contained within registers in the CPU, such as in a `MOV A,B` instruction, the *register addressing* mode is used. Memory is not addressed, and only a few bits are required to specify the limited number of registers. Thus, register addressing instructions are among the fastest to execute and use the fewest bits of any of the instructions. Some manufacturers call register addressing by other names. For example, Freescale uses the term *inherent addressing* and Intel uses *implied addressing*.

Immediate Addressing

You may use *immediate addressing* when an operand is a constant known when you write the program. If this is the case, the data can *immediately* follow the instruction in the memory. Figure 5-10 shows a memory map of the immediate addressing mode where the data may be 8 bits, 16 bits, or more, depending on the size of the destination register.

Direct Addressing

In *direct memory addressing*, the instruction contains the address of the data. It may be the full physical address in linear addressing architectures or an offset for segmented architectures. In any event, the location of the data for this instruction is constant. There are two variants of this mode: (1) *direct addressing* and (2) *base page*, or *reduced direct addressing*. Manufacturers use different terminology (e.g., absolute addressing, extended direct addressing, long and short absolute addressing).

Direct addressing means that the address of the data, either the full address or an offset, is in the instruction. Figure 5-11 shows direct addressing for a processor with a 16-bit address space (64 Kbyte). It is a *single-level* addressing mode because the instruction contains the address of the data. The 16-bit address follows the operation code in the memory.

Direct addressing is the simplest mode to understand, and many beginning students try to use it exclusively. Often, however, this mode needs more bits than other addressing modes,

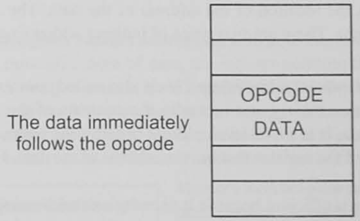


Figure 5-10 Immediate addressing.

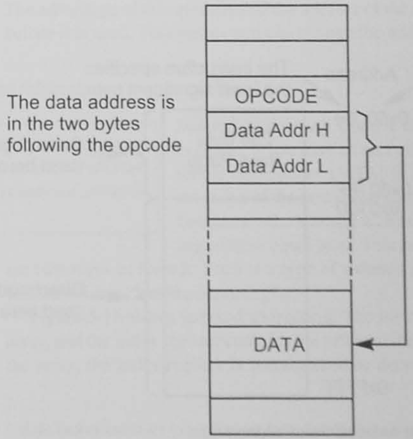


Figure 5-11 Direct memory addressing.

and the location of the data addressed by the instruction is fixed. The direct mode is especially unsuitable for addressing elements in a table of data.

Direct Base Page Addressing

In a CPU with *base page addressing*, the computer designers provide instructions that specify only the least significant bits of the full address. The processor then generates the complete address by filling the most significant bits with 0s. For example, in a machine with 16 address bits and a 256-byte base page, the base page addressing instructions specify only the 8 least significant bits. The CPU provides the 8 most significant bits, as shown in Figure 5-12.

Base page addressing offers the advantage that the instruction has to specify only 8 bits of the full 16-bit address. This saves program bytes and makes the instructions execute faster. The disadvantage is that usually only a few memory locations are available for data storage.

Indirect Addressing

Indirect addressing is a *two-level* addressing mechanism. The instruction provides the first-level address, which specifies the location of the *address* of the data. The second level then specifies the location of the *data*. There are two types of indirect addressing: register indirect and memory indirect.

A register points to the data in *register indirect addressing.*

Register indirect addressing: This is also called *pointer register addressing* because the register (actually the contents of the register) "points" to the data. It is a two-level address because the instruction contains the address of the register that has the address of the data. Figure 5-13 shows this addressing mode.

Register indirect addressing is efficient because it uses register addressing, and thus only a few bits, for the first-level address. Another advantage is that the address of the data can be calculated at run time, as you might do when stepping through a table of data. Remember, though, to initialize the register with an address before using it.

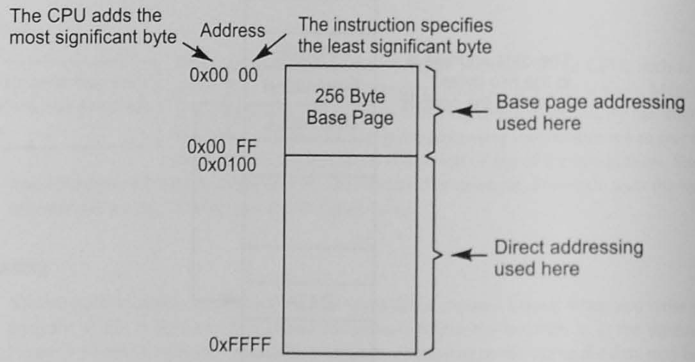

The CPU adds the most significant byte
Address
The instruction specifies the least significant byte
0x00 00
256 Byte Base Page
Base page addressing used here
0x00 FF
0x0100
Direct addressing used here
0xFFFF

Figure 5-12 Base page or reduced direct addressing.

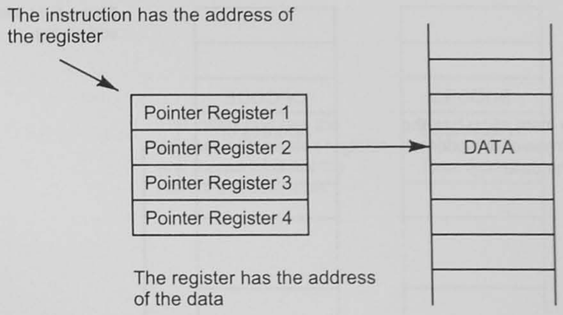

The instruction has the address of the register
Pointer Register 1
Pointer Register 2
Pointer Register 3
Pointer Register 4
DATA
The register has the address of the data

Figure 5-13 Register indirect addressing.

Register indirect addressing with auto increment and auto decrement: When you are stepping through a table of data, the register pointing to the data must be incremented or decremented. Some processors have an addressing mode that automatically increments or decrements the register. You may have a choice of preincrementing or predecrementing, where the register is incremented or decremented before it is used. In postincrementing/-decrementing, the register is incremented/decremented after it has been used.

A memory address contains the address of the data in *memory indirect addressing.*

Memory indirect addressing: In memory indirect addressing, the instruction contains the memory address of the address of the data. Figure 5-14 shows this addressing mode. Memory indirect addressing is less efficient than direct memory or register indirect addressing because the CPU first reads the address of the address, then the address, and finally the data. The advantage of this mode is that the address of the data can be calculated and stored in RAM before it is used. This means you can change the address while your program is running.

Indexed and Based Addressing

When the effective address is the sum of a register and a constant value, the mode is *indexed* or *based* addressing.

Indexed addressing finds a memory location based on an index. For example, if you have an array of bytes of data, you might refer to the individual elements as DATA[0], DATA[1], . . . , DATA[n]. The [n] is called the index of the array. The address of any element in the array consists of two parts—the starting address of the array and an offset from the starting address equal to *n*.[1] This sum is called the effective address and there are two ways to form it. Each is a type of indexed addressing, although some manufacturers call the second "based addressing".

Figure 5-15 shows indexed addressing. The instruction contains the starting address of the array, and the index register contains the offset to the element being addressed. To step through the array, the index register is incremented or decremented, either explicitly with a program

[1] If the data elements are larger than one byte, the offset is *n* times the size of the element in bytes.

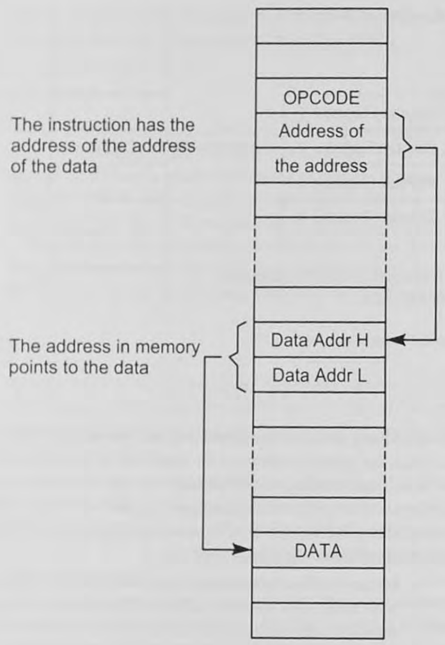

OPCODE
Address of
the address
The instruction has the address of the address of the data
Data Addr H
Data Addr L
The address in memory points to the data
DATA

Figure 5-14 Memory indirect addressing.

instruction or automatically in processors that have the autoincrement/-decrement addressing mode. Some processors can also scale the increment by the size of the data element. For example, if your data array were 4 bytes per element, incrementing the index register would add 4 to point to the next element in the array.

This form of indexed addressing must have the direct address of the start of the array in the instruction; it uses more bytes than register indirect addressing. Consequently, manufacturers have included based addressing to reduce the number of bits carried by the instruction.

Based addressing: Figure 5-16 shows how based addressing works. Here the "index" or "base" register has the starting, or base, address of the array. The instruction provides the offset into the array. The CPU adds the index register and the offset to calculate effective address. This is different from indexed addressing described earlier because the instruction contains the offset rather than the direct address of the start of the data. This scheme can reduce the number of bytes in the instruction when the offset is smaller than the full address range, say 8 bits instead of 16.

Based addressing is almost as efficient as register indirect addressing. There is an additional byte (or two) to be fetched from memory for the offset, and time is taken to add the offset to the contents of the index register to create the effective address. Unfortunately, to add to the

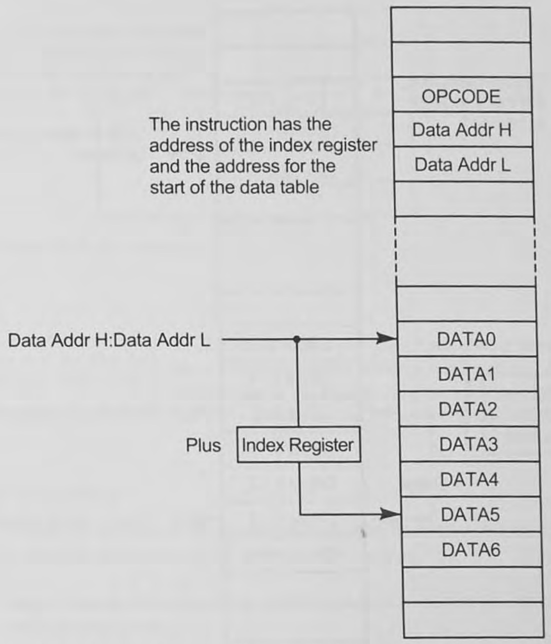


Figure 5-15 Indexed addressing.

confusion of the beginning student, based addressing is called indexed addressing in some processors.

Relative Addressing

Relative addressing is used for branching short distances in the program.

Relative addressing modes calculate the effective address by adding an offset to the current value of the program counter. Well-written programs use this addressing mode for branch instructions because branches jump over only a few bytes of code. Thus, a programmer can save memory by using relative branch instructions. Figure 5-17 shows relative addressing.

The actual value of the offset is calculated from the memory location labeled *Next Opcode* in Figure 5-17 and is usually a two's-complement number to allow branching forward and backward.

Bit Addressing

Many microprocessors and microcontrollers input and output individual bits. For example, you might want to read one or more switches and act depending upon whether the switch is open

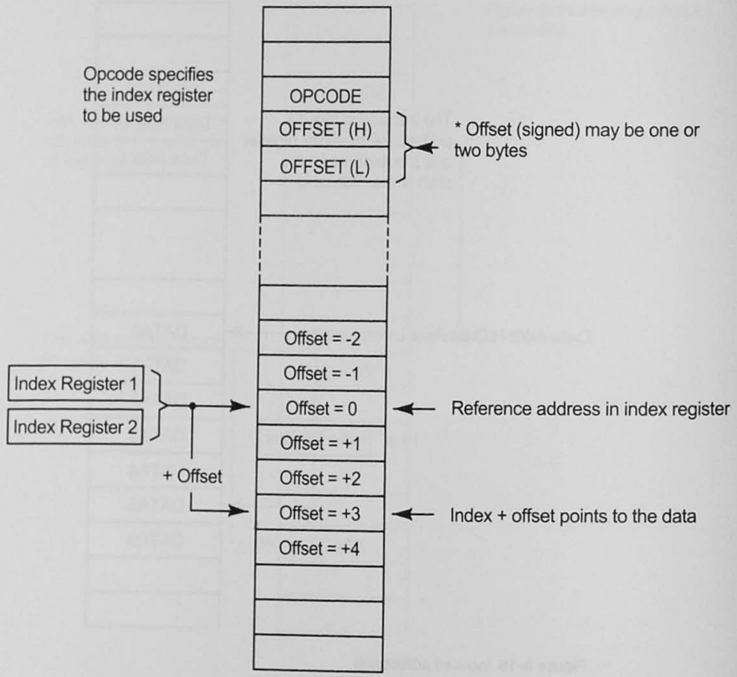

Opcode specifies the index register to be used
OPCODE
OFFSET (H)
OFFSET (L)
* Offset (signed) may be one or two bytes
Offset = -2
Offset = -1
Offset = 0
Offset = +1
Offset = +2
Offset = +3
Offset = +4
Index Register 1
Index Register 2
+ Offset
Reference address in index register
Index + offset points to the data

Figure 5-16 Based addressing.

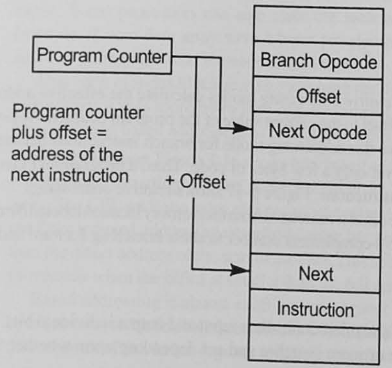


Figure 5-17 Relative addressing.

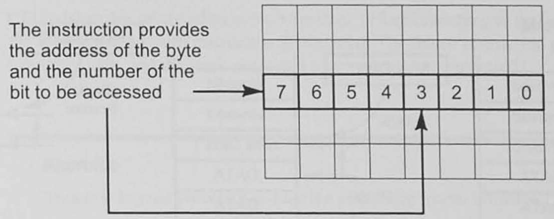

The instruction provides the address of the byte and the number of the bit to be accessed
7
6
5
4
3
2
1
0

Figure 5-18 Bit addressing.

Bit addressing can save memory by allowing eight binary variables to be saved in one byte.

or closed. Designers provide *bit addressing* to read or write one bit at a time. Usually the bit is within a byte location, either in memory or I/O, so the instruction must supply the address of the byte plus a mask to specify which bit within this byte is to be addressed. Figure 5-18 shows bit addressing.

Other Addressing Combinations

In some more powerful microprocessors you may find addressing modes that are combinations of the basic modes described in the previous sections.

Based indexed addressing: The effective address is the sum of a base register, index register, and a displacement.

Relative addressing with index plus displacement: The effective address is the program counter plus an index register plus a displacement.

5.7 Stack Addressing

The *stack* contains data and return addresses for subroutines.

The stack is an area of RAM that is reserved for temporary data storage. It operates on a last-in, first-out (LIFO) basis. That is, the last information stored on the stack is the first to be retrieved. The stack operates like the pile of plates in a dining hall. You always take the plate on the top of the stack (i.e., the last one put there). Disaster awaits those who try to remove plates from the middle of the stack! Information is stored to and retrieved from the stack with a CPU register called the *stack pointer (SP)*. The stack pointer points either to the last information pushed onto the stack or to the next available location,[2] and it must be initialized to point to the memory used for the stack.

Figure 5-19 shows a stack memory map. Memory maps are usually drawn as shown with higher memory addresses at the bottom and lower at the top. Figure 5-19a shows the stack pointer pointing at the last location that was used when information was placed onto the stack.

[2] Which of these design strategies is used is immaterial because the processor automatically handles the stack pointer properly.

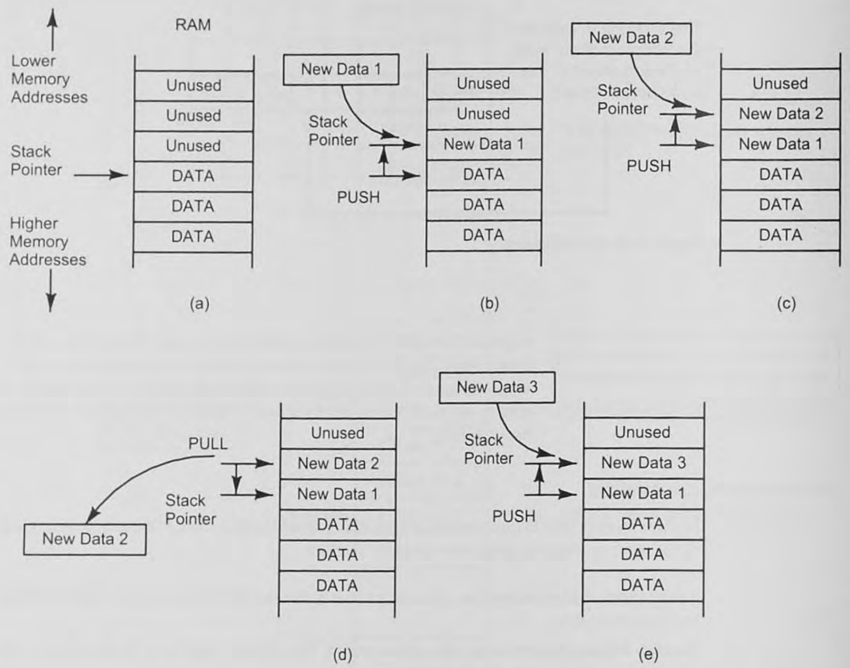


Figure 5-19 Stack operations. (a) Stack pointer before stack operations. (b) Stack pointer after a push. (c) Stack pointer after a second push. (d) Stack pointer after a pull. (e) Stack pointer after a third push.

Push and Pull Operations

Placing data onto the stack is called a *push*. In this example, the stack pointer points to the last memory location used and is automatically decremented by the push instruction before the new data byte is stored. The result of pushing two bytes is shown in Figure 5-19b and 5-19c. *New Data1* and *New Data2* are now in memory, and the stack pointer has been decremented twice. The result of a *pull* (called a *pop* in some processors) is shown in Figure 5-19d; for a subsequent push operation see Figure 5-19e.

Subroutine Call and Return Operations

The stack saves the return address when the program branches to a subroutine. After it has fetched the branch opcode and the subroutine address from the program memory, the program counter is pointing to the instruction to be executed after the return from the subroutine. The CPU pushes this return address onto the stack before branching to the subroutine. At the end of the subroutine, a return instruction is executed. The return instruction pulls the return address from the stack, and the stack pointer is automatically incremented.

Exercise

Turn now to your processor and list the addressing modes available.

5.8 Chapter Conclusion and Summary Points

- Computer systems have both RAM and ROM memory.
- RAM is volatile; it is used for variable data in embedded systems and variable data and programs in desktop systems.
- ROM is nonvolatile; it is used for programs in embedded systems and the BIOS in desktop systems.
- Addressing modes are the different ways a processor specifies the location of an operand.
- Different addressing modes give the programmer flexibility in accessing data elements.
- The effective address is the physical or logical address of the data.
- Register addressing specifies data located in registers.
- Immediate addressing is used for constant data known when you write the program.
- A direct addressing instruction contains the address of the data.
- Indirect addressing is a two-level addressing mode; the instruction specifies the address of the address of the data.
- In indexed addressing, the effective address is calculated by adding the contents of a register to a direct address contained in the instruction.
- In based addressing, the effective address is calculated by adding an offset contained in the instruction the contents of a register.
- Based addressing is also called indexed addressing.
- A relative addressing instruction contains an offset from the current value of the program counter. Relative addressing is used mostly for branch instructions.
- The stack is an area of RAM set aside for program use.
- The stack pointer register is used to store data and subroutine return addresses on the stack.

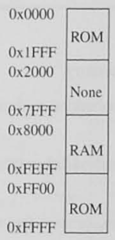
5.9 Problems

Explore

5.1 List the addressing modes available in the CPU you are studying.

5.2 Briefly explain the following terms: physical address, effective address, memory map. [g]

5.3 A microcontroller is to be used in an embedded system with the following memory map:



a. In what memory addresses must code and constant data be located? [c]
b. In what memory addresses must variable data and stack storage be located? [c]

5.4 If all bits in the PPAGE register shown in Figures 5-8 and 5-9 are used to generate an expanded address, how much memory in total can be added to the microcontroller? [a]

5.5 Name at least five ways to address an operand. [a]

5.6 What kind of addressing mode is used to transfer data from one register to another? [a]

5.7 What are the names of the addressing modes that form the effective address from a constant and the contents of a register? [a]

5.8 What address mode is best to use when you want to compare what is in the A register with a constant? - (immediate, direct, extended, or indexed)? [a]

5.9 To increase the memory address space in a computer system, one must (a) increase the number of data lines, (b) increase the number of read and write control bits going to the memory, (c) increase the number of address lines. [a]

5.10 A pointer is (a) an area in memory used for address storage, (b) a memory address held in a register, (c) a subroutine address held in the stack pointer. [a]

5.11 A register indirect address instruction (a) has the address of the operand in the instruction, (b) has the address of the operand in a register, (c) uses the program counter to calculate the offset address of the operand. [a]

Stimulate

5.12 In the movable segment architecture shown in Figure 5-6, why is the segment located on a 16-byte boundary? [b]

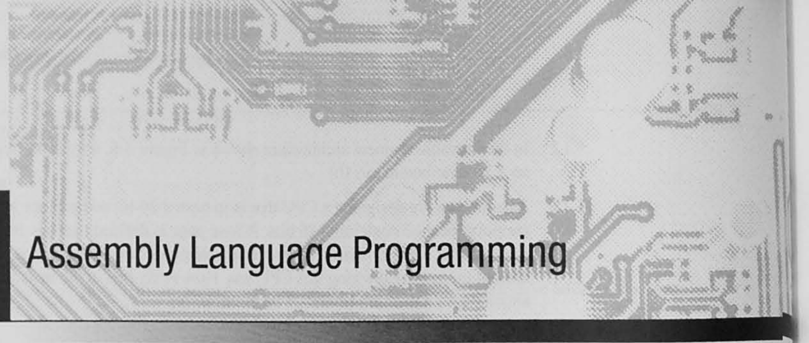
5.13 Assume you are designing a CPU that is to have a 20-bit address bus with each memory location containing 16 bits. A base page is defined that has 1024 locations. Assume that memory indirect addressing using base page addresses is the ONLY kind of memory addressing this CPU has. How many bits in the instruction must be allocated for a memory reference instruction? [c]

Reflect on Learning

5.14 Of all the addressing modes in the processor you are studying, which do you have the most difficulty understanding?

5.15 What experiments could you try with your system to be able to learn more about the addressing modes.

5.16 List five things that you learned about memory addressing modes in this chapter.



6 Assembly Language Programming

Objectives

This chapter will show programming techniques and suggest an assembly language programming style. An example using the Freescale CodeWarrior relocatable assembler, is given and explained. We will also show how to write structured assembly language programs that meet the goals of top-down software design presented in Chapter 3. Although our assembly language examples are for the HCS12 microcontroller of Freescale Semiconductor, Inc., you should be able to easily translate the essence of the code to your own microcontroller.

6.1 Assembly Language Programming Style

You will need to learn the syntax requirements of your assembler. Most assemblers are very similar, however, with the fields of each program line separated by white spaces. In addition to being aware of the syntactical requirements of each line, you should adopt a standard format or style for the programs you write. This will make the programs more readable for colleagues who may have to modify your code or collaborate on a software engineering project.

Source Code Style

A consistent style can make your programs easier to read.

Any program is a sequence of program elements, from the top to the bottom, and these elements should be organized in a readable and consistent style. Adopt a standard format and use it for all assembly language programs. Table 6-1 shows a format that can serve as an outline for your programs. The subsections that follow provide program examples of the individual elements; Example 6-1 then lists the completed program.

Table 6-1 Assembly Language Program Elements

Program Element	Purpose
Program header	Briefly describes the purpose of the program
External symbol definitions	References for symbols defined in some other source file
Internal symbol definitions	References for symbols defined in this source file
Assembler equates	Definition of constants used in the program
Code section start	Defines the following bytes to be in the code segment or section in ROM
Program initialization	Initializes the stack pointer, I/O devices, and other variables
Main program body	The main program
Program end	Starts the main program again or terminates it in some way
Program subroutines	Subroutines and functions used in the main program
Constant data section start	Defines the following bytes to be in the constant data segment in ROM
Constant data definitions	Definitions of constants in ROM
Variable data section start	Defines the following bytes to be variable data elements in RAM
Variable data allocation	Allocation of space for variable data elements

Program Header

After reading the header, you should know what the program does, not in any great detail, but at least in general. The author's name should be here so praise (or blame) can be apportioned correctly. The date of original code release and modification record is good information too. The modification record should tell what has been done to the original code, when it was done, and by whom.

Program Element | **Program Example**

Program header

```
; MC68HCS12 Assembler Example
;
; This program is to demonstrate a
; readable programming style.
; It initializes the A/D converter
; and a bank of LEDs. It then reads the
; value on the A/D, displays it, and delays
; about 0.5 second. It then displays the
; last value it converted for about 0.5
; second and repeats.
; Source File:  M6812EX1_REL.ASM
; Author:  F. M. Cady
; Created: 7/26/2009

; Modifications:  None
```

External Symbol Definitions

When you are using a relocatable assembler, source files may reference a symbol or label that is defined in some other source file. It is the job of the linker to evaluate the symbol and to provide the value for it. The CodeWarrior XREF directive tells the assembler to leave the resolution of the symbol for the linker.

Program Element	Program Example
External symbol definitions	see listing below

```
;*******************************************
; External symbol definitions
        XREF  get_AD, init_AD
        XREF  enable_LED, put_LED
        XREF  delay_X_ms
        XREF  __SEG_END_SSTACK
;*******************************************
```

Internal Symbol Definitions

Whenever there is an external symbol definition (XREF) in a source file, there must be an accompanying definition of the symbol (XDEF) in some other source file that is part of the project. This section of the program provides the necessary definition.

Program Element	Program Example
Internal Symbol Definitions	see listing below

```
;*******************************************
; Internal symbol definitions
        XDEF  Entry, main
;*******************************************
```

Assembler Equates

Equates are often found at the beginning of the program.

Equates are like the #define statements in a C program. They are used to define a constant value for the assembler. Some programmers put equates at the top of the program, and some argue that it is more useful to put a constant definition right where it is used. We suggest that all equates be in one area in the program and that they appear before they are used.

Program Element	Program Example
Constant Equates	see listing below

```
;*******************************************
;   Constant Equates
DELAY: EQU   500   ; Used for delay
 subroutine
;*******************************************
```

Code Section Start

Each section in a relocatable assembly language program should have a name. This allows you to easily locate the sections with the linker parameter file. The linker provides a file

showing where all program elements can be found, and when sections are named it is easy to identify them in the file.

Program Element	Program Example
Code Section Start	see listing below

```
;*******************************************
; Code Section Start
MyCode: SECTION
```

Program Initialization

The *stack pointer* must be initialized before it is used for subroutine calls, interrupts, and data storage. Do it as the first instruction in the program. *Variables* must be initialized at run time. Put the section of code to do this here. In a C program this is done automatically by the startup code.

Program Element	Program Example
Stack Pointer Initialization I/O Devices Initialization Variable Data Initialization	see listing below

```
Entry:
main:
; Initialization section
; Initialize stack pointer
        lds   #__SEG_END_SSTACK
; Initialize all I/O devices
        jsr   init_AD     ; Init the A/D
        jsr   enable_LED  ; Enable LED port
; Initialize the last data value
        clr   Last_Val
```

Main Program Body

The main program starts here. Typically it will be short and consist of several subroutine calls.

Program Element	Program Example
Main Program Body	see listing below

```
;*******************************************
; Main process loop starts here:
loop:
; Get value from A/D
        jsr   get_AD
        pshb          ; Save it
; Display on LEDs
        jsr   put_LED
; Delay about 0.5 seconds
        ldx   #DELAY
        jsr   delay_X_ms
; Now display the last value
        ldab  Last_Val
        jsr   put_LED
        pulb          ; Get the value back
        stab  Last_Val; Save it for next time
; Delay 100 milliseconds
        ldx   #Delay1
        jsr   delay_X_ms  ; Delays # ms in X reg
```

Program End

When you develop software on an evaluation system, such as a manufacturer's evaluation board, you must return control to the monitor at the end of your program. This is often done with a *software interrupt* instruction. In programs that run continuously with no need to return to the debugging monitor, make a branch or jump back to the beginning of the process loop.

Program Element	Program Example

Return to the Beginning of Main Loop	`; Do forever`
	`            bra   loop`
	`;*******************************************`

Program Subroutines

It is good programming practice to make the main program a sequence of calls to subroutines. You may place subroutines anywhere in the source program, or they may be in other source files if a relocatable assembler is used. In this program example we choose to do the latter.

Program Element	Program Example

Subroutines and Functions	`;****************************************************`
	`; Subroutines and functions`
	`;   (This relocatable assembler program does`
	`;    not place any subroutines in the main`
	`;    module. If you want to include subroutines,`
	`;    however, this is the place to put them.)`
	`;****************************************************`

Constant Data Section Start

Constants will be located in the ROM memory. You do not have to create a constant data section, but it is good programming practice to do so.

Program Element	Program Example

Constant Data Section Start	`;*************************************`
	`; Constant data area in ROM`
	`MyConst:SECTION`

Constant Data Definitions

Constants are located in ROM. Usually, to decrease the danger of executing data, it is best to have constants at the end of all code sections. However, some programmers group constants with the section of code that uses them (i.e., constants used in a subroutine).

Program Element	Program Example
Main Program Constants and Strings	`Delay1: DC.W   DELAY`

Variable Data Section Start

Variable data are located in the RAM memory. The name you give allows you to locate the variable data with the linker.

Program Element	Program Example

Variable Data Section Start	`; Variable data area in RAM`
	`MyData: SECTION`

Variable Data Storage Allocation

Use the DS to allocate storage for all variable data elements.

Program Element	Program Example
Allocation of Data Areas	`Last_Val: DS.B  1`

The Completed Program

Example 6-1 shows this program as a complete assembler source file. We have not shown the subroutines that initialize the LED display or the A/D converter or the subroutines for getting data from the A/D and displaying it on the LEDs.

Example 6-1 The Completed Program

```
; MC68HCS12 Assembler Example
;
; This program is to demonstrate a
; readable programming style.
; It initializes the A/D converter
; and a bank of LEDs. It then reads the
; value on the A/D, displays it, and delays
; about 0.5 second. It then displays the
; last value it converted for about 0.5
; second and repeats.
; Source File: M6812EX1_REL.ASM
; Author: F. M. Cady
; Created: 7/26/2007
; Modifications: None
;
;*******************************************
; External symbol definitions
        XREF  get_AD, init_AD
        XREF  enable_LED, put_LED
        XREF  delay_X_ms
        XREF  __SEG_END_SSTACK
;*******************************************
```

```
; Internal symbol definitions
        XDEF Entry, main
;******************************************
;   Constant Equates
DELAY: EQU 500 ; Used for delay sub
;******************************************
; Code Section Start
MyCode: SECTION
Entry:
main:
; Initialization section
; Initialize stack pointer
        lds   #__SEG_END_SSTACK
; Initialize all I/O devices
        jsr   init_AD    ; Init the A/D
        jsr   enable_LED ; Enable LED port
; Initialize the last data value
        clr   Last_Val
;******************************************
; Main process loop starts here:
loop:
; Get value from A/D
        jsr    get_AD
        pshb               ; Save it
; Display on LEDs
        jsr    put_LED
; Delay about 0.5 seconds
        ldx    #DELAY     ; Show the use of an EQU
        jsr    delay_X_ms ; Delays # ms in X
; Now display the last value
        ldab   Last_Val
        jsr    put_LED
        pulb              ; Get the value back
        stab   Last_Val   ; Save it for next time
; Delay about 0.5 seconds
        ldx    Delay1     ; Show the use of a constant in ROM
        jsr    delay_X_ms ; Delays # ms in X
; Do forever
        bra    loop
;******************************************
; Subroutines and functions
;   (This relocatable assembler program does
;    not have any subroutines in the main
;    module. If you want to include subroutines,
;    however, this is the place to put them.)
;******************************************
; Constant data area in ROM
MyConst:SECTION
```

```
Delay1: DC.W    DELAY
String: DC.B    "This is a string of constants."
;******************************************
; Variable data area in RAM
MyData: SECTION
Last_Val: DS.B 1
```

To Indent or Not to Indent

Indentation is not used very often in assembly language programming.

In high-level languages, indentation shows lower levels of the design and makes the code more readable. Indentation is not generally used in assembly language programming. Historically, assemblers were used long before high-level language compilers that allowed indentation were developed. Also, an assembler's syntax is generally fixed. Often, labels must start in the first space on the line, and there must be white space between labels, mnemonics, operands, and comments. Assembly language programmers are used to seeing the program with the fields all nicely lined up because it is easier to identify the operations and operands. However, you may want to try a few programs with indented code to see how you like it.

Upper-case and Lower case

The use of *upper-* and *lowercase* letters can make your programs more readable.

Upper- and lowercase letters can make your code more readable. The goal is to be able to look at a name or label and tell what it is without searching further. For example, uppercase labels can be used for constants and lowercase for variables. Mixed case used for multiple-word labels can make them easier to read. Table 6-2 shows examples of all three label types. Some assemblers are not case sensitive and some are.

Table 6-2 Examples of Upper-, Lower-, and Mixed-Case Labels

Case	Examples
Uppercase	

Constants defined by EQU	`NULL:    EQU   0x0`
	`PORT_H: EQU   0x24`

Constants defined by DC.B, DC.W	`STRING: DC.B  This is a string.`
	`CRLF:    DC.W  0x0D0A`

Assembler directives	`ORG, EQU, DC`
Lowercase	
Instruction mnemonics	`ldaa, jsr, bne`

Labels	`loop: . . .`
	`        bne   loop`

Variables	`data: DS      10`
Mixedcase	

Multiword variables and labels	`PrintData:`
	`NumChars:`
	`InputDataBuffer:`

Multiword subroutine names	`Jsr   PrintData`
Comments	`; Write complete sentences for comments.`

Use Equates, Not Magic Numbers

Equates make programs easier to read and easier to change in the future.

A number that just appears in the code is called a magic number. For example, if the program statement

```
ldab #8
```

appeared in the program, you would have to ask, "What significance does the 8 have in the program?" Is it used as a counter or as an output value? You do not know. However, the following code

```
COUNTER: EQU   8
         ldab  #COUNTER
```

is much clearer. Furthermore, if the counter is used in several places in the program and needs to be changed, it is easier to change the equate than to search for and change all places it is used. Always use the `EQU` directive to define constants in your program.

Using Include Files

An *include file* can contain frequently used symbols and definitions.

Assembly language programs often use the same equates in each program. More powerful assemblers allow include files to be used similar to using #include in C programs

Commenting Style

There are various commenting styles. Some programmers would have a comment on each program line. Another style is to place comments in blocks that explain what the following section of code is to do (i.e., on the design or function of each block). Then, within the block of code, place comments on lines that may call for further explanations. Using high-level, pseudocode design statements as comments in the program is very effective also. Table 6-3 shows useful information that can be included as comments in each subroutine's header.

Table 6-3 Subroutine or Function Header

```
; Subroutine calling sequence or invocation
; Subroutine name
; Purpose of subroutine
; Name of file containing the source
; Author
; Date of creation or release
; Input and output variables
; Registers modified
; Global data elements modified
; Local data elements modified
; Brief description of the algorithm
; Functions or subroutines called
```

6.2 Structured Assembly Language Programming

You can write assembly language program segments to do high-level language structures.

Pseudocode is a tool often used in program design. After we have completed our design, we must write the assembly language code for it. There are two parts of the assembly language code to do structured programming. The first is a comment. This normally can be taken from the pseudocode design document. The second part is the code that implements the comment. Let us look at the three structured programming elements as they might appear in assembly language.

Sequence

The sequence is straightforward. There should be a block of comments describing what the next section of assembly code is to do. Remember that the flow of the program is in at the top and out at the bottom. We must not enter or exit the code between DO_A and ENDO_A except to call and return from a subroutine. Do not jump into or out of the middle of a sequence block. See Example 6-2.

Example 6-2 Assembly language for sequence block

```
; ************************************************************
; DO_A
; Comments describing the function of this sequence block
....(Assembly language code to do the function)
; ENDO_A
; ************************************************************
```

IF-THEN-ELSE Decision

Example 6.3 shows a pseudocode design that uses the decision element; the associated assembly language code is shown in Example 6-3. See also Example 6-4 and 6-5.

The IF-THEN-ELSE code always has the same form. The **bold** lines in Example 6-3 will appear in every decision structure. Notice that in the assembly code the indentation familiar from our work with high-level languages is not used, although we may retain the indentation of the pseudocode comments.

Example 6-3 Decision Element Assembly Language Program

Pseudocode Design

```
; Get the temperature
; IF Temperature > Allowed Maximum
;     THEN Turn the water valve off
;     ELSE Turn the water valve on
; END IF temperature > Allowed Maximum
```

Structured Assembly Code

```
1.    ; 68HCS12 Structured assembly code
2.    ; IF-THEN-ELSE example.
3.    ; Equates defne constants needed by the code
4.    AD_PORT:    EQU   0x91 ; A/D Data Port
5.    MAX_TEMP:   EQU   128  ; Maximum temperature
6.    VALVE_OFF:  EQU   0    ; Bits for valve off
7.    VALVE_ON:   EQU   1    ; Bits for valve on
8.    VALVE_PORT: EQU   0x258; Port P for the valve
9.    ;
10.   ;
11.   ;           . . .
12.   ; Get the temperature
13.          ldaa  AD_PORT
14.   ; IF Temperature > Allowed Maximum
15.          cmpa  #MAX_TEMP
16.          bls   ELSE_PART
17.   ;   THEN Turn the water valve off
18.          ldaa  VALVE_OFF
19.          staa  VALVE_PORT
20.          bra   END_IF
21.   ;   ELSE Turn the water valve on
22.   ELSE_PART:
23.          ldaa  VALVE_ON
24.          staa  VALVE_PORT
25.   END_IF:
26.   ; END IF temperature > Allowed Maximum
```

Explanation of Example 6-3

Lines 12, 14, 17, 21, and 26: These lines contain the pseudocode design as comments in the source code.

Line 15: Following the IF statement is code to set the condition code register for the conditional branch in *line 16* to the ELSE part.

Line 16: There will always be a conditional branch to the ELSE part, as shown here, or to the THEN part. When you branch to the ELSE part, the conditional branch instruction is the complement of the logic in the IF statement. In this example, the ELSE part is to be executed if the temperature is lower or the same as the allowed maximum because the THEN part is done when the temperature is higher.

Lines 18 and 19: This is the code for the THEN part.

Line 20: The THEN part *always* ends with a branch-always or jump to the END-IF label. This branches around the ELSE part code.

Line 22: The label for the ELSE part conditional branch is always here.

Lines 23 and 24: This is the code for the ELSE part.

Line 25: The IF-THEN-ELSE always ends with an END_IF label.

Example 6-4

For each of the logic statements, give the appropriate assembler code for your microcontroller to set the condition code register and to branch to the ELSE part of an IF-THEN-ELSE. Assume that P and Q are 8-bit, signed numbers in memory locations P and Q.

```
A. IF P >= Q
B. IF Q > P
C. IF P = Q
```

Solution (for Freescale HCS12)

```
A. ; IF P >= Q
       ldaa   P
       cmpa   Q
       blt    ELSE_PART   ; Branch if P is less than Q
B. ; IF Q > P
       ldaa   Q
       cmpa   P
       ble    ELSE_PART   ; Branch if Q is less than or equal to P
C. ; IF P = Q
       ldaa   P
       cmpa   Q
       bne    ELSE_PART  ; Branch if P is not equal to Q
```

Example 6-5

For each of the logic statements, give the appropriate assembler code for your microcontroller to set the condition code register and to branch to the THEN part of an IF-THEN-ELSE. Assume that P and Q are 8-bit, unsigned numbers in memory locations P and Q.

```
A. IF P >= Q
B. IF Q > P
C. IF P = Q
```

Solution (for Freescale HCS12)

```
A. ; IF P >= Q
       ldaa   P
       cmpa   Q
       bhs    THEN_PART  ; Branch if P is higher or the same as Q
```

```
B. ; IF Q > P
     ldaa  Q
     cmpa  P
     bhi   THEN_PART  ; Branch if Q is higher than P
C. ; IF P = Q
     ldaa  P
     cmpa  Q
     beq   THEN_PART  ; Branch if P is equal to Q
```

WHILE-DO Repetition

The WHILE-DO structure is shown in Example 6-6. The elements common to all WHILE-DOs are in **bold**.

Example 6-6 Assembly Code for a WHILE-DO

Pseudocode Design

```
; Get the temperature from the A/D
; WHILE the temperature > maximum allowed
;    DO
;       Flash light 0.5 sec on, 0.5 sec off
;        Get the temperature from the A/D
;    END_DO
; END_WHILE the temperature > maximum allowed
```

Structured Assembly Code

```
1.    ; 68HCS12 Structured assembly code
2.    ; WHILE - DO Example
3.    ; Equates needed
4.    AD_PORT:     EQU 0x91   ; A/D Data port
5.    MAX_ALLOWED:EQU 128    ; Maximum Temp
6.    LIGHT_ON:    EQU 1
7.    LIGHT_OFF:   EQU 0
8.    LIGHT_PORT:  EQU 0x258  ; Port P
9.    ;  - - -
10.   ; Get the temperature from the A/D
11.           ldaa  AD_PORT
12.   ; WHILE the temperature > maximum allowed
13.   WHILE_START:
14.           cmpa  MAX_ALLOWED
15.           bls   END_WHILE
```

```
16.   ;   DO
17.   ;     Flash light 0.5 sec on, 0.5 sec off
18.           ldaa  LIGHT_ON
19.           staa  LIGHT_PORT  ; Turn the light
20.           jsr   delay       ; 0.5 sec delay
21.           ldaa  LIGHT_OFF
22.           staa  LIGHT_PORT  ; Turn the light off
23.           jsr   delay
24.   ;     End flashing the light
25.   ;     Get the temperature from the A/D
26.           ldaa  AD_PORT
27.   ; END_DO
28.           bra   WHILE_START
29.   END_WHILE:
30.   ; END_WHILE the temperature > maximum allowed
31.
32.   ; Dummy subroutine
33.   delay:  rts
```

Explanation of Example 6-6

Lines 10, 12, 16, 17, 24, 25, 27, and 30: The pseudocode design appears as comments in the code.

Lines 14 and 15: A WHILE-DO tests the condition at the top of the code to be repeated. Thus, the conditional branch in *line 15* must be preceded by code that initializes the variable to be tested. The A register is initialized with the A/D value in *line 11*.

Line 13: There must be a label at the start of the conditional test code. This is the address for the BRA in *line 28*.

Line 14: Following the WHILE statement is code to set the condition code register for the subsequent conditional branch to the end of the WHILE-DO.

Line 15: A conditional branch allows us to exit this structure.

Lines 17–26: This is the code for the DO part.

Line 26: A special requirement of the WHILE-DO structure is code that changes whatever is being tested. If this were not here, the program would never leave the loop.

Line 28: The code block always ends with a branch back to the start.

As an assembly language programmer, you might be enough smarter than the average compiler to realize that *line 26* could be eliminated if the code to initialize the A register with the A/D value (*line 11*) is moved below the label WHILE_START.

DO-WHILE Repetition

Another useful repetition is the DO-WHILE. In this structure, the DO part is executed at least once because the test is at the bottom of the loop. An example of the DO-WHILE is shown in Example 6-7 where, again, the parts common to all DO-WHILEs are in **bold**.

Example 6-7 DO-WHILE Assembly Language Code

Pseudocode Design

```
; DO
;    Get data from the switches
;    Output the value to the LEDs
; ENDO
; WHILE Any switch is set
```

Structured Assembly Code

```
1.   ; 68HCS12 Structured assembly code
2.   ; DO-WHILE example
3.   ; Equates needed for this example
4.   SW_PORT: EQU 0x28 ; Switches are on Port J
5.   LEDS:    EQU 0x24 ; The LEDs are on Port H
6.   ;           - - -
7.   ; DO
8.   DO_BEGIN:
9.   ;   Get data from the switches
10.          ldaa  SW_PORT
11.  ;   Output the data to the LEDs
12.          staa  LEDS
13.  ; END_DO
14.  ; WHILE Any switch is set
15.          tst   SW_PORT
16.          bne   DO_BEGIN
17.  ; END_WHILE
```

Explanation of Example 6-7

Lines 7, 9, 11, 13, and 14: The pseudocode appears as comments.

Line 8: The start of the DO block has a label for the conditional branch instruction in *line 16.*

Line 9–12: These are the code lines for the DO part.

Lines 15 and 16: The DO-WHILE always ends with a test and a conditional branch back to the beginning of the DO block.

Line 17: A comment marks the end of the WHILE test code.

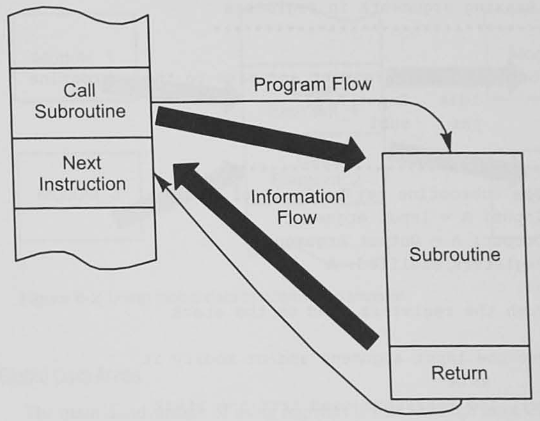


Figure 6-1 Information transfer between modules.

6.3 Interprocess Communication

Most programs pass information between one part of the program and another.

Interprocess communication, also called parameter passing, refers to information that is transferred from one part of the program to another. Most information transfer in well-designed programs is between a subroutine or function and its calling function, as shown in Figure 6-1. In choosing how information is transferred between modules, a goal is to reduce the chance of the subroutine accidentally changing other data. There are several methods that can be used.

Information in Registers

The most efficient and fastest way to transfer information between parts of a program that is being written in assembly language is to use the registers. Another advantage of this method is that the subroutine does not access any other data areas and is thus more general. Documentation must be provided to show what registers are used for what purpose. A typical subroutine header describing the registers used is shown in Table 6-4. Using the registers is simple and straightforward (Example 6-8).

Example 6-8 Passing Information in Registers

```
;*******************************
; Parameter passing between modules
;*******************************
```

```
; Passing arguments in registers
;*******************************
; . . .
; Get the input argument and pass to the subroutine
        ldaa   Input_Arg1
        jsr    sub1
; . . .
;*******************************
; The subroutine may be local or external
; Input: A = Input Argument
; Output: A = Output Argument
; Registers modified: A
sub1:
; Push the registers used on the stack
; . . .
; Use the input argument and/or modify it
        asla
; Pull the registers used from the stack
; Return with the modified data
        rts
;*******************************
MyData: SECTION
; Place variable data here
Input_Arg1: DS.B  1
```

Table 6-4 Subroutine Header Comments

```
; *****************************************************
; *
; * Subroutine Name:      SQRT
; * Author: F. M. Cady
; * Date: July 19, 2009
; * Function: Calculate the square root of a 16
; *   bit integer number.
; * Input Registers:
; *   D = 16 bit integer number
; * Output Registers:
; *   B = 8 bit integer square root
; *   Carry flag = 1 if input number is negative
; *   Carry flag = 0 if input number is positive
; * Registers modified:
;*  B, condition code register
; * Global data modified: none
; * Functions called: none
; *
; *****************************************************
```

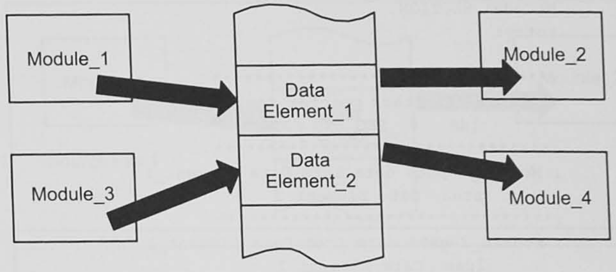


Figure 6-2 Using global data in information transfer.

Information in Global Data Areas

The main disadvantage of using registers is that although most CPUs have only a few, some functions may need many bytes of data. Using global data areas is a solution with advantages and potential problems. Global data are data elements that can be reached from any part of the program. Figure 6-2 shows four modules making use of two global data elements.

Using global data to pass information can cause hard-to-find problems in your programs.

The danger of maintaining global data is that a function may modify data that it shouldn't have disturbed. For example, let's assume that Module_1 shares Data_Element_1 with Module_2 and Module_3 shares Data_Element_2 with Module_4. Now let's assume that you make a mistake (a bug) in the code that is supposed to write data into Data_Element_1 and write into Data_Element_2 instead. (This could be done by using a 16-bit store operation instead of an 8-bit one, by having an incorrectly initialized pointer register, or simply by writing the wrong label in the operand field.) Now Module_4 is working with incorrect data. This is a difficult bug to find, particularly if the code in Module_1 is executed infrequently. Experienced assembly and high-level language programmers try to avoid using global data if other methods are available. Nevertheless, global data structures are widely used in assembly and high-level language programming. See Example 6-9.

Example 6-9 Passing Information in Global Data Area

```
;*******************************
; Passing arguments in global data
;*******************************
; Define the entry point for the main program
        XDEF  Entry, main
        XREF  __SEG_END_SSTACK
; Define the data names that are external in
; a global data buffer
        XREF  Data_Element_1, Data_Element_2
        XREF  Data_Element_3, Data_Element_4
```

```
MyCode: SECTION
Entry:
main:
;*******************************
; Initialize stack pointer register
       lds   #__SEG_END_SSTACK
;*******************************
; Module_1 puts data into Data_Element_1
       staa  Data_Element_1
;*******************************
; Module_2 gets data from Data_Element_1
       ldaa  Data_Element_1
;*******************************
; Module_3 puts data into Data_Element_2
       staa  Data_Element_2
;*******************************
; Module_4 gets data from Data_Element_2
       ldaa  Data_Element_2
;*******************************

;*******************************
; This is the global data definition
; The data storage allocations are done
; here and all data names are XDEFed to
; make them globally available
;*******************************
       XDEF  Data_Element_1, Data_Element_2
       XDEF  Data_Element_3, Data_Element_4
GlobalData: SECTION
; Place variable data here
Data_Element_1: DS.B  1
Data_Element_2: DS.B  1
Data_Element_3: DS.B  1
Data_Element_4: DS.B  1
```

Information in Local Data Areas

Local data areas invoke the principle of divide and conquer. Figure 6-3 shows modules and their common data elements, which are separately assembled source files. When a relocatable assembler is used, as it must be here, any names or labels are local to that source file only unless a special assembler directive called EXTERNAL or XREF is used. Thus, the assembler will show an error if you assemble the file with Module_1, Module_2, and Data_Element_1 and accidentally refer to Data_Element_2. However, as you can see, the data elements are global within these localized structures.

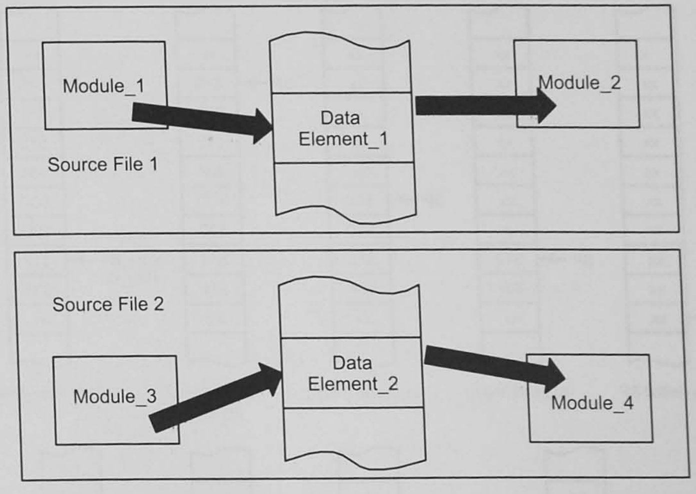


Figure 6-3 Information in local data areas.

Information on the Stack

The stack can also be used to transfer data to and from a subroutine. When this is done, the data elements are localized on the stack, and the subroutine is designed to operate with them alone. This reduces the chance of global data being accidentally corrupted. You must be careful when using the stack because, in addition to the data on the stack, the return address and any bytes pushed on the stack when the subroutine is entered are there also.

Example 6-10 and Figure 6-4 show how to use the stock to pass data to and from a subroutine. Figure 6-4a shows the initial position of the stack pointer and the contents of the stack. *Line 18* in Example 6-10 pushes 16-bit data onto the stack (Figure 6-4b), and the subroutine call is made in *line 19* (Figure 6-4c). The D and X registers are pushed onto the stack in the subroutine (*lines 35* and *36*, Figure 6-4d). The subroutine uses indexed addressing and the stack pointer (*lines 40* and *44*) to retrieve data from the stack and return data to the stack. The number of bytes between the current value of the stack pointer and the data to be pulled is given by *Num_B* + *Reg_B*. After the registers have been restored (*lines 47* and *48*) and after the return from subroutine (*line 49*), the main program can retrieve the returned data (*line 22*).

Using the stack to transfer information is very powerful and very general. Most compilers for high-level languages use this method. Programmers must be careful to make sure that stack operations are balanced; good documentation is essential.

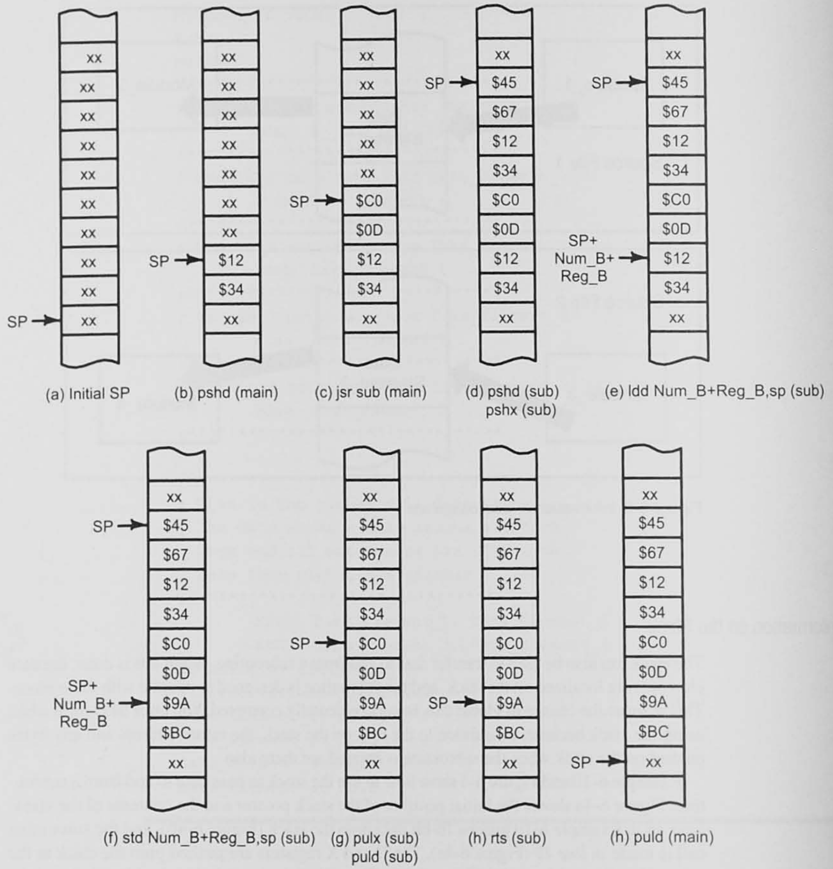


Figure 6-4 Using the stack for information transfer.

Example 6-10 Passing Information on the Stack

```
;*******************************
; Passing arguments on the stack
;*******************************
; Define the entry point for the main program
```

```
          XDEF  Entry, main
          XREF  __SEG_END_SSTACK
MyCode: SECTION
Entry:
main:
;*******************************
; Initialize stack pointer register
          lds   #__SEG_END_SSTACK
;*******************************
;
          ldd   #0x1234  ; Demo data
          ldx   #0x4567
; Put the data to be transferred on the stack
          pshd    ; Two bytes transferred
          jsr   sub1
; Get the returned data and clean up the
; stack pointer
          puld    ; Two bytes returned
; . . .
;*******************************
;*******************************
; Subroutine sub
; Input: 16-bit data on the stack
; Output: 16-bit data on the stack
; Registers modified: CCR
;*******************************
Num_B:  EQU   2  ; Number of data bytes on stack
Reg_B:  EQU   4  ; Number of register byes on stack
sub1:
; Push registers used in the subroutine onto the stack
        pshd
        pshx
; . . .
; Use indexed addressing to get the data passed in
; from the stack
        ldd   Num_B+Reg_B,sp
; . . .
; Put the return data back on the stack
        ldd   #0x9ABC
        std   Num_B+Reg_B,sp
; . . .
; Pull the used registers from the stack
        pulx
        puld
        rts
;*******************************
```

Using Addresses Instead of Values

Despite the potential problems and disadvantages, the use of global data areas is common, particularly when large amounts of data are to be shared between modules. One can avoid some problems by using a register to pass the address of the data. This tactic is useful when one is accessing data buffers where, perhaps, one module fills the buffer and another processes the data. The two modules can be given the starting address of the buffer and the number of data elements (to avoid running over the end of the buffer).

Passing Boolean Information

At times a Boolean, or logic, value must be returned to the calling program. For example, you might want to indicate whether a procedure was successful and then act accordingly in the calling program. A register or a memory location could be allocated for this; but if you are programming in assembly language, you can use a bit in the condition code register. All processors have the capability of setting or resetting the carry flag. This can be tested with a conditional branch instruction in the calling program.

6.4 Assembly Language Tricks of the Trade

Here are some tricks of the assembly language programming trade.

1. **Do not modify registers in a subroutine:** You should ensure that a subroutine in assembly language DOES NOT modify the contents of any of the registers unless a register is to return a value to a calling program. In assembly language programming we use registers to hold data from one instruction to another. Unseen instructions in a subroutine should not change registers. An exception to this rule may be the condition code register.
2. **Use register addressing when possible:** Instructions that use internal registers execute faster and use less memory.
3. **Use register indirect or indexed addressing:** These modes are the next most efficient, after register addressing. The address may be calculated at run time, allowing the location of the data to be a variable depending on the current state of the program. Often data must be stored in or retrieved from a buffer. An indirect addressing mode, such as register indirect or indexed addressing, is most efficient for this, especially if the register can be automatically incremented or decremented. Remember to load the register with the address before using it.
4. **Use the stack for temporary data storage:** When you are using the stack in subroutines, remember to pull the registers before returning to the calling program.
5. **Do not use the assembler to initialize variable data areas:** Assemblers have directives or pseudo-operations to initialize the contents of memory locations. This works well when a program in the system is downloaded each time it is run. However, in an embedded microcontroller application, where the program resides in ROM, all variable data areas must be initialized at run time by the program. Use directives that allocate memory storage locations for the variables and then initialize them in the program.
6. **Use assembler features, directives, and pseudo-operations:** Study and use the assembler directives and pseudo-operations. It is mandatory to use labels for symbolic addresses. Never refer to a memory location by a direct address. Using the assembler to evaluate expressions can make programs more readable and more easily transferred to other applications. If a macroassembler is available, use macros to make programs more readable.

6.5 Making It Look Pretty

Your programs will be judged by your peers, not only for how they run but also for how they look. It does not take much extra effort to make your program listings look good. Here are some helpful hints.

- Adopt a consistent indentation style. Most assembly language programmers indent and align the opcode, the operand, and the comment fields.
- Use frequent comments. A very effective commenting style is to include your design statements as comments preceding the code that implements the commented design. There is no need to place a comment on every line of assembly code. Simply comment assembly lines that need some explanation.
- Place your equates in one place in the program, often at the beginning, so readers will know where to go to find them.
- Use white space—blank lines—to separate sections of code.
- Use comment lines of asterisk characters (*) to mark off sections of code.
- Don't try to completely box in a section of comments with asterisk characters. If the comment lines change, you will have to spend extra time cleaning up the boxes. See Example 6-11.
- Use upper- and lowercase writing style. Don't put comments in all uppercase or all lowercase.
- Use both cases for constants, variable, labels, and so on. Adopt a style that looks good, and be consistent with it.

Example 6-11

```
;*************************************************
;* Even though it looks pretty, you should not    *
;* try to box in comments with asterisk (or other *
;* characters.) If the comments change, it is just *
;* more work to change the line.                  *
;*************************************************
```

6.6 Conclusion and Chapter Summary Points

In this chapter we have shown an example of a readable program style for assembly language programs. We have illustrated, and strongly urge you to adopt, a structured assembly language programming style that uses a structure pseudocode design implemented in assembly language.

6.7 Bibliography and Further Reading

Ganssle, J. G., *A Guide to Commenting*. The Ganssle Group, Baltimore, February 2006. http://www.ganssle.com/commenting.htm

6.8 Problems

Explore

6.1 For each of the logic statements, give the assembly language code for your microcontroller to set the condition code register and to branch to the ELSE part of an IF-THEN-ELSE. Assume that P and Q are 8-bit unsigned numbers in memory locations P and Q. [c, k]

a. IF P >= Q
b. IF Q > P
c. IF P = Q

6.2 For each of the logic statements, give the assembly language code for your microcontroller to set the condition code register and to branch to the ELSE part of an IF-THEN-ELSE. Assume that P and Q are 8-bit signed numbers in memory locations P and Q. [c, k]

a. IF P >= Q
b. IF Q > P
c. IF P = Q

6.3 If you have a C compiler for your microcontroller that can produce an assembly language list file, repeat Problems 6.1 and 6.2 and compare the compiled code with your solution.

Stimulate

6.4 For each of the logic statements, give the assembly language code for your microcontroller to set the condition code register and to branch to the ELSE part of an IF-THEN-ELSE. Assume that P, Q, and R are 8-bit signed numbers in memory locations P, Q, and R. [c, k]

a. IF P + Q >= 1
b. IF Q > P – R
c. IF (P > R) OR (Q < R)
d. IF (P > R) AND (Q < R)

6.5 Assume that K1 and K2 are 8-bit signed (two's-complement) integer variables and that K3 is a 16-bit unsigned integer variable. [c, k]

a. Show how to allocate storage for these variables in a relocatable assembly language program by using the assembler used for your microcontroller.

b. Write structured assembly language code for the following design. (Assume that K1, K2, and K3 have been initialized in some other part of the program.)

```
; IF K1 < K2
;   THEN
;     Set K1 to the most positive number
;   ELSE
;     Set K3 to the most positive number
;     Initialize K2 to the most negative number
; ENDIF K1 < K2
```

6.6 Insert code to implement the following structured design immediately after each design comment. Assume that the following structured design is just a small segment of an overall program.

Assume that the following 8-bit, two's-complement variable data allocations have been made and have been initialized in some other part of the program. [c, k]

```
Temp1:    DS.B  1
Temp2:    DS.B  1
 ; Implement the following design
 ; IF Temp1 < Temp2
 ;     THEN Temp1 = Temp2
 ;     ELSE Temp2 = Temp1
 ; ENDIF
 endif:
```

6.7 Insert code to implement the following structured design immediately after each design comment. Assume that the following structured design is just a small segment of an overall program.

Assume that the following 8-bit unsigned variable data allocations have been made and have been initialized in some other part of the program. [c, k]

```
Temp3:    DS.B  1
Temp4:    DS.B  1
Temp5:    DS.B  1

; Implement the following design
; WHILE Temp3 > Temp4
; DO
;     Temp4 = Temp4 + 1
;     Temp5 = 2 * Temp5
; ENDWHILEDO
enddo:
```

6.8 Write a section of assembly language code for your microcontroller to implement the given design, where K1 and K2 are 8-bit unsigned numbers in memory locations K1 and K2. [c, k]

```
; IF K1 < K2
;     THEN K2=K1
;     ELSE K1=64
; ENDIF K1 < K2
```

6.9 A 16-bit number is in sequential memory positions DATA1 and DATA1+1 with the most significant byte in DATA1. Write an assembly language code segment for your microcontroller to store the negative of this 16-bit number in DATA2 and DATA2+1. [a, c, k]

6.10 Write a section of assembly language code for your microcontroller to implement the design: [c]

```
IF Data1 > Data2
THEN Data2 = Data1
ELSE Data2 = 64₁₀
ENDIF Data1 > Data2
```

Assume Data1 and Data2 are memory locations containing 8-bit unsigned integer data. Structured code must be used and comments must be included:

Challenge

6.11 In Example 6-1 a constant defined by an equate is used to initialize a register with a constant value in *line 50*, and a constant stored in ROM memory is used to initialize a register in *line 58*. Comment on these two assembly language programming techniques. Which is better? [a]

6.12 Write assembly language code for your microcontroller that will implement the C structure [c]

```
for (i = 0; i < 10; ++i){
...
}
```

6.13 Write assembly language code for your microcontroller for the following pseudo-code design, assuming that K1, K2, and K3 are 8-bit signed or unsigned numbers in memory locations K1, K2, and K3. Assume that memory has been allocated for these data. [c, k]

```
; WHILE K1 does not equal 0x0d
while_start:
;   DO
;     IF K2 = K3
;       THEN
;          K1 = K1 + 1
;          K2 = K2 - 1
;       ELSE
;          K1 = K1 - 1
;     ENDIF K2 = K3
;   ENDO
enddo:
; ENDOWHILE
```

6.14 Write a structured assembly language code segment for the following pseudocode design. [c, k]

Assume that P, Q, and R are 8-bit signed integer variables in memory locations P, Q, and R. Insert the code needed for the design in the comments below. You may add more comments if you wish. [c, k]

```
; IF P does not equal Q
;   THEN P=R
; ENDIF P does not equal Q
; WHILE P < R
;   DO
;       P = P + 1
;   ENDO
; ENDWHILEDO
end_while:
```

6.15 Write a section of assembly language code for your microcontroller to implement the following design, where K1, K2, and K3 are signed 8-bit integer numbers stored at memory locations K1, K2, and K3. [c, k]

```
; WHILE K1 < K2
;   DO
;       IF K3 > K2
;          THEN   K2 = K1
;          ELSE K2 = K3
;       ENDIF K3>K2
;       K1 = K1+1
;   ENDO
; ENDWHILE K1<K2
```

6.16 For Problem 6.15, assume that K1=1, K2=3, and K3=–2. How many times should the code pass through the loop, and what final values do you expect for K1, K2, and K3? [b]

6.17 Write structured assembly language code for your microcontroller for the following design: [c, k]

```
; IF A1 = B1
;   THEN
;     WHILE C1 < D1
;       DO
;         Decrement D1
;         A1 = 2 * A1
;       ENDO
;     ENDWHILE C1 < D1
;   ELSE
;     A1 = 2 * B1
; ENDIF A1 = B1
```

Assume that A1, B1, C1, and D1 are 16-bit unsigned-binary numbers and that memory has been allocated in the program by the following code:

```
A1:    DS.B   2
B1:    DS.B   2
```

```
C1:    DS.B   2
D1:    DS.B   2
```

Assume that A1, B1, C1, and D1 are initialized to some value in an other part of the program.

6.18 For Problem 6.17, assume that A1=2, B1=2, C1=3, and D1=6. What final values do you expect after the code has been executed? [b]

6.19 Write a structured assembly language code segment for the following pseudocode design. [c, k]

Assume that P and Q are 8-bit unsigned integer variables in memory locations P and Q. Also assume that function X is implemented in a subroutine named X. Insert the code needed for the design in the comments below.

```
; If P = 0x1B
; THEN
;   WHILE Q < 186
;      DO function X
;      ENDDO
;   ENDWHILEDO
; ENDIF
```

6.20 Write a pseudocode design for the following program statement. [c, k]

The program is to prompt for a two-digit hexadecimal number and use a routine called getchar to accept it from a user. If the two digits entered by the user signify a printable ASCII character, the character is to be printed with an appropriate message. Otherwise, an error message is to be printed. The program is to continue until the user types two hex numbers that are not a code for a printable character. Your design must show at least one example of a repetition and one decision.

(*Example*: If the user types a 4 and then a 1, A should be printed along with an appropriate message.)

6.21 Use the principles of structured programming to write structured pseudocode (do not write assembly language code) for the following problem statement. [c]

The program is the prompt for and will accept a two-digit hexadecimal number from a user typing characters on the keyboard. These digits are to be converted to an 8-bit binary number and displayed on the LEDs. After a one-second delay, the complement of the byte is to be displayed on the LEDs for one second. After this delay, the LEDs are to be turned off and the process repeated starting at the prompt. The program is to continue until the user types two zeros ("00").

Your design should follow the principles of top-down design, and you may postpone consideration of such details as how to convert the two input characters to binary, and the details of the prompt and how it is to be printed.

6.22 Write a sub program for your microcontroller to find the largest of thirty-two 8-bit unsigned numbers in 32 successive RAM memory locations (BUF). Place the answer in the 33rd location (Result). [c]

6.23 Write a sub program for your microcontroller to find the smallest of thirty-two 8-bit unsigned numbers in 32 successive RAM memory locations (BUF). Place the answer in the 33rd location (Result). [c]

6.24 Write a sub program for your microcontroller to find the largest (most positive) of thirty-two 8-bit two's-complement numbers in 32 successive RAM memory locations (BUF). Place the answer in the 33rd location (Result). [c]

6.25 Write a sub program for your microcontroller to find the smallest (most negative) of thirty-two 8-bit, two's-complement numbers in 32 successive RAM memory locations (BUF). Place the answer in the 33rd location (Result). [c]

6.26 Write a sub program for your microcontroller to find the address of the largest of thirty-two 8-bit unsigned numbers in 32 successive RAM memory locations (BUF). Place the answer in the 33rd:34th location (Res-adr). If more than one location contains the largest number, use the lowest address as the result. [c]

6.27 Write a sub program for your microcontroller to find the address of the largest of thirty-two 8-bit unsigned numbers in 32 successive RAM memory locations (BUF). If more than one location contains the largest number, use the highest address as the result. Place the answer in the 33rd:34th location (Res-Adr). [c]

6.28 There are 4 bytes of data in 4 successive RAM memory locations (BUF). Write a structured assembly sub program to count the number of 1s in these bytes. Place the result in the fifth memory location (Result). [c]

6.29 Write a structured assembly sub program to reverse the order of 0x20 bytes in a buffer. Assume that the buffer is in 32 successive RAM memory locations (BUF). [c]

6.30 Write a structured assembly program to compute factorial 8. Store the result in a 2-byte memory location in RAM memory (Factorial). [c]

6.31 Write a structured assembly program subroutine to search a null-terminated string of characters for a specific substring and to return the address of the start of the substring. The input to the subroutine is to be the starting address of the string to be searched, the starting address of the substring to be searched for, and the number of characters in the substring. If the substring is found, return the address of the first character in the search string; otherwise return an address of 0x0000. [c]

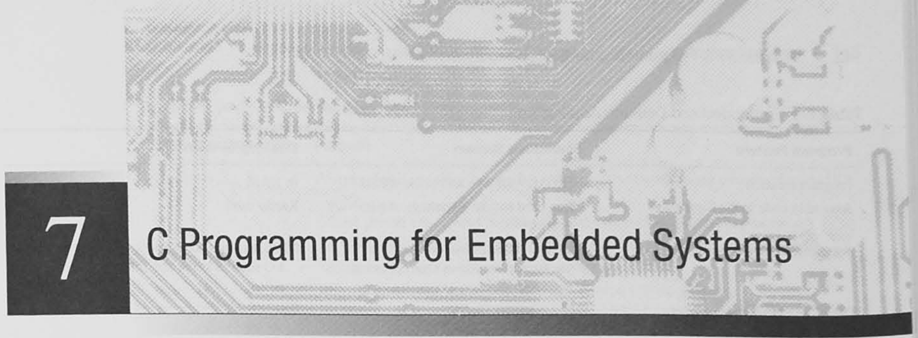
6.32 Write an assembly program showing how to transfer 4 bytes of data from the main to a subroutine using the stack. The subroutine does not return any data to the main. Show how the main puts data onto the stack, how the subroutine retrieves the data, and how the main program restores the stack pointer after the return from the subroutine. [c]

6.33 An 8-bit signed/magnitude number system is in use. Write assembly subroutines for the following: [c]

a. Add two 8-bit signed/magnitude numbers.
b. Subtract two 8-bit signed/magnitude numbers.
c. Multiply two 8-bit signed/magnitude numbers.
d. Divide two 8-bit signed/magnitude numbers.

Reflect on Learning

6.34 What was for you the most significant new thing you learned in this chapter?



7 C Programming for Embedded Systems

Objectives

In this chapter we show some of the changes in thinking needed to program in C for an embedded system microcontroller instead of a desktop or personal computer application. We assume that you have learned to program in C in another course and now wish to use C to create programs for your microcontroller.

7.1 Introduction

Although assembly language programs are still used in embedded applications, many are programmed in C because of the increased programming efficiency, portability, and documentation provided by this high-level language. Other high-level languages, such as C++, have been created for various microcontrollers, but C is still widely used. In this chapter we show some of the changes needed in the C language to create programs for embedded microcontrollers.

7.2 Major Differences Between C for Embedded and Desktop Applications

C programs for embedded applications are different from those for desktop applications.

The C programming language gives system engineers a high-level language for developing embedded system applications. This widely used tool gives us the programming efficiency and portability we have come to expect when comparing high-level languages with assembly language programming. Embedded system developers, however, must pay closer attention to the architecture of the processor and to the interface with the real world than is required of a developer of an application for a desktop or personal computer. Table 7-1 lists some of the differences found when comparing C programs written for these two applications.

One of the major differences between embedded and desktop applications is that in the former, the read-only memory contains the executable code (giving us the term *firmware*) instead of the copious RAM found in desktop system. To locate a program properly, the system engineer must

Table 7-1 Embedded and Desktop Applications

Program Feature	Embedded System	Desktop System
Program location	In ROM, with data and stack in RAM	In RAM
Assembly code in the C program	Often used to take advantage of hardware features	Rarely used
Operating system support	Rarely found in embedded systems, except for some real-time operating systems	Often used to take advantage of user-oriented I/O such as keyboard, display, and disk drives
Use of on-chip features (timers, A/D, etc.)	Often used	Rarely used
Calls to procedures written in assembly language and in-line assembly	Often used	Sometimes used when procedures are available for special-purpose hardware
Interrupt service routines	Often needed and used	If needed, often provided by operating system support

know where the ROM is located. Another difference is that often RAM in an embedded system is a scarce resource. The embedded system engineer must carefully evaluate how much RAM is needed for variable data storage and for the stack. C programs, though, may help us conserve this scarce resource. This is because automatic variables, declared inside a function, use the stack, and the storage space for these variables is released when the function exits.

Because embedded applications often control some hardware in real time, C programs can effectively use assembly language functions. To write these assembly language functions, you must know what calling convention (how the registers are used and how data are placed on the stack) is being used. A programmer can also introduce assembly language statements in-line with C statements to control directly the hardware.

Many embedded applications use hardware-generated interrupts to control the program flow. The routines written for these interrupts are called *interrupt service routines* or *interrupt handlers*. They require a special return from interrupt instruction to return to the interrupted part of the program. Compilers need nonstandard ANSI C features to provide this capability.

ANSI C versus Microcontroller Implementations

The ANSI C standard programming language is extended with features for embedded applications.

The American National Standards Institute (ANSI) established a committee in 1983 to produce "an unambiguous and machine-independent definition of the language C." The result was the ANSI standard for C. However, ANSI C was not designed for embedded controller applications. It lacks standard ways to assign pointers to specific memory locations, such as the I/O registers; it also lacks a way to implement interrupt service routines, which require a return from interrupt instruction, and extended addressing for microcontrollers with paged memory architectures. Compiler vendors tackle these problems in different ways, so you must check the documentation for your compiler's language extensions. Table 7-2 shows some typical compiler language extensions.

Paged or Expanded Memory

As the semiconductor manufacturing industry has matured, it has become possible to add far more memory to the microcontroller base memory than can be addressed by the address bus. Different microcontrollers use different strategies for accessing this memory. A switching mechanism is included for accessing purposes, and special call and return instructions are implemented.

Table 7-2 ANIS C Compiler Language Extensions

Language Extension	Purpose
@address	Assigns global variables to specific address; useful for accessing memory mapped I/O ports
far	Allows pointers to access the whole memory range supported by the processor
near	Keyword used when default addressing is far and the near calling convention is to be used
interrupt	Specifies a function to be an interrupt service routine
asm	Allows assembly language instructions to be placed in the program

Table 7-3 Compiler Data Types

		Data Range	
Data Type	Number of Bits	Minimum	Maximum
unsigned char	8	0	255
signed char	8	−128	127
unsigned short, unsigned int	16	0	65,535
signed short, enum, signed integer	16	−32,768	32,767
unsigned long, unsigned long long	32	0	4,294,967,295
signed long, signed long long	32	−2,147,483,648	2,147,483,647

Data Types

The size of data types varies from one compiler implementation to another depending on the target microcontroller. For example, in one microcontroller the size of an int variable may be 16 bits, while in another it will be 32 bits. As an example, Table 7-3 shows the size of data types for a typical compiler designed for a 16-bit architecture.

Portability

A major benefit of ANSI C is the standardization that allows a program to be "ported," or moved, to another processor. However, it is usual for the compiler vendor to extend the C language to support a specific microcontroller. Using these extensions reduces the portability of the code to another, different processor. It is a good design practice to organize your code so that any processor-specific language enhancements are in easily identified modules. You can then replace these if you move to a different microcontroller. Writing the rest of the code in ANSI C will allow increased portability.

Exercise 7-1

What ANSI C extensions does your compiler provide?

7.3 Architecture of a C Program

Embedded applications have program in ROM and use RAM for variable data and the stack.

In an embedded system, as we discovered in Chapter 2, the executable code is "burned" into read-only memory (ROM), with variable data and stack segments located in read-write memory (RAM). Often these memory types are not contiguous, and to link and locate the final executable code, the software or firmware engineer must know where the various types of memory are located.

Figure 7-1 shows the memory map of a typical microcontroller. The 2 Kbyte of RAM is used for variable data storage and the stack, which in a C program provides the storage locations for *automatic* variables. Our executable program resides in the 32 Kbyte of Flash EEPROM. In comparison to a desktop personal computer system, there is not very much RAM, and the entire program is located in the Flash EEPROM read-only memory.

The compiler and the linker/locater for C programs written for an embedded application must allow us to position the code in the ROM and to use the RAM for the variable data storage. In addition, it must allow us to make specific reference to particular memory locations to access the control registers in the 1 Kbyte register space shown in Figure 7-1. The executable code consists of the code you write, starting with your main() program and all procedures linked together, plus a section of code normally provided by the C compiler called the *startup* code.

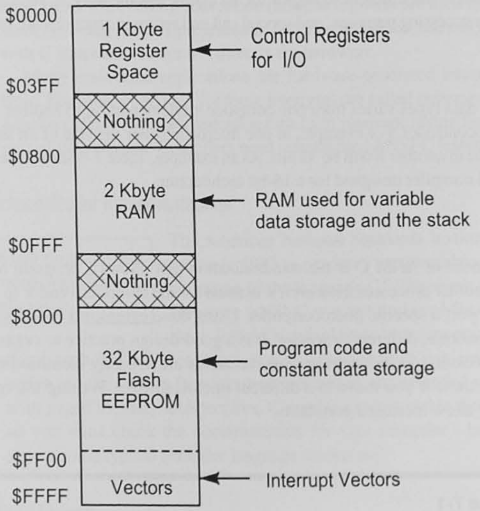


Figure 7-1 Microcontroller memory map.

Start-up Code

Start-up code is automatically included by the compiler and initializes variables and hardware features before the main program executes.

The compiler vendor provides start-up code to initialize a variety of microcontroller hardware features plus any initialized program variables. Typically, it will do the following initializations.

- Initialize hardware needed to run the microcontroller in a default state.
- Define register values for paged or expanded memory available in some microcontrollers.
- Initialize registers to move (remap) RAM, EEPROM, and I/O registers from their default memory addresses to new ones if required.
- Initialize to zero any static data locations allocated in RAM.
- Initialize to their starting values any variables initialized by the program.
- Initialize the stack pointer register.
- Call the main() program to transfer control to your embedded system.

void main(void);

Your programming efforts start with the *void main(void)* program segment. The last thing the start-up code does after its initialization steps is to call your main program. The first thing the compiler-generated code does in your main() program (and any other module) is to allocate storage for variables and to initialize those that have not already been taken care of in the start-up code.

Variables

Automatic Variables

Automatic variables are stored on the stack during the execution of the module in which they are used.

Automatic variables are those declared within a procedure; no other procedure has access to them, and their lifetime (accessibility and validity) ends when the procedure's execution time ends. Because these variables come and go with the function, they do not retain their value from one invocation to another.

Automatic variables are placed on the stack, and the compiler generates code to access these in a variety of ways. Embedded system programmers must ensure that the system has enough stack memory to accommodate the automatic variables in a function. This mechanism for storing variable data is very efficient. RAM in most microcontrollers is a finite and scarce resource. By using automatic variables, we make it possible for succeeding functions to reuse these RAM locations.

Automatic variables may be initialized to some value or not. If they are not initialized, the value is undefined.

Static Variables

Static variables are allocated storage space in RAM.

Static variables may be declared either inside a function or outside it. They are located in RAM and are reserved for use throughout the program. The start-up code initializes static variables either to zero or to whatever value is specified in the program.

Volatile Variables

Volatile variables will be on the stack if they are automatic or in RAM if they are static.

A volatile variable is one whose value may change as a result of outside forces. For example, an A/D converter may be loading a register with new conversion values. A variable that is being set by reading that location should be declared volatile to ensure the compiler does not eliminate code that it considers redundant or not necessary. Example 7-1 shows two variables, `volatile static BYTE A_Val` and `B_Val`, which is nonvolatile. Depending on how your compiler optimizes code, `A_Val` will be written twice (because it is declared volatile) but `B_Val` only once (because the compiler eliminates what it thinks is redundant code).

Example 7-1 C Volatile Variables

Source Code

```
typedef unsigned char BYTE;
    BYTE PORTA;
void main(void) {
volatile static BYTE A_Val;
        static BYTE B_Val;
  /* Read from Port A */
  A_Val = PORTA;    /* A_Val should be writttten twice */
  A_Val = PORTA;

  /* */
  B_Val = PORTA;    /* B_Val may be written only once */
  B_Val = PORTA;

  }
```

7.4 Assembly Language Interface

Compiler-Produced Assembly Language Code

Embedded system designers must be aware of all parts of the code and be able to understand what the compiler is doing. In any C program, we may not be aware of code (e.g., the start-up

Compilers can produce a listing file that shows all the assembly language code that has been generated.

code) produced in support of our application program. Before the application code for any function starts to execute, the compiler generates code to initialize automatic variables that have initial values. This overhead could degrade the execution time of our program and cause unwanted effects. Fortunately, most compilers can produce a listing file showing the actual code produced.

Example 7-2 shows the compiler-generated code to initialize variables in a main function. The static char `A_Val` array is initialized as part of the startup code because it is a static variable. The automatic char `B_Val` array must be initialized before any code in main is executed. As you can see from the listing, this is done by the code immediately following *line 9*. The `C_Val` array and the `i` variable are not initialized prior to use.

Example 7-2 C Program Overhead to Initialize Variables

Source Code

```
/****************************/
void main(void) {
 static char A_Val[] =
      {2,90,53,8};
 char B_Val[] =
      {0,7,255,34};



 char C_Val[4], i;
/****************************/
  i = 2;
  C_Val[i] = A_Val[i]+B_Val[i];





}
```

Listing Showing Compiled Code

```
.
.
.
.
  9:         char B_Val[] = {0,7,255,34};
0000 69a8        CLR   8,-SP
0002 c607        LDAB  #7
0004 6b81        STAB  1,SP
0006 86ff        LDAA  #255
0008 6a82        STAA  2,SP
000a c622        LDAB  #34
000c 6b83        STAB  3,SP
.
.
.
  12:    i = 2;
  13:    C_Val[i] = A_Val[i]+B_Val[i];
 000e f60000     LDAB  A_Val:2
 0011 eb82       ADDB  2,SP
 0013 6b86       STAB  6,SP
 .
 0015 1b88       LEAS  8,SP
 0017 3d         RTS
```

Explanation of Example 7-2

The listing of the compiled code in Example 7-2 shows how invoking a function like main() produces code we may not see. The static `A_Val` has been initialized in the start-up code, but the automatic variable `B_Val` is initialized here and placed on the stack (e.g. `LDAB #7, STAB 1, SP`). Although this example shows code produced for main(), similar code will be found for any procedure with initialized automatic variables.

Using Assembly Language in C

Embedded system programmers sometimes resort to assembly language functions to reduce the amount of ROM program memory needed or to solve some timing problem by using assembly modules that execute faster than a comparable C program. A frequent scenario is that an application program is written entirely in C and then analyzed to find the bottlenecks. In this procedure, called *profiling*, software contributing to the bottlenecks can be rewritten in assembly language to improve performance. The overriding C program can call these assembly modules the same as any C function.

To create an assembly module called by a compiled C program, we must know how the compiler transfers arguments into and back from the assembled module. Table 7-4 shows methods used by two popular compilers for the HCS12 microcontroller. If you are using another vendor's compiler, you should be sure to check its documentation to find out what calling convention is used.

In-Line Assembly

Assembly language statements may be intermixed with C program statements.

You can insert assembly language instructions into a C program by using *in-line assembly*. All compilers have slightly different syntax for implementing this useful feature. You must be cautious when mixing assembly language instructions and C program statements. In general, the C program does not save or restore register contents in normal operation, although some compilers allow a register type variable that is kept in a register and the register's contents is maintained. Typically, when you use in-line assembly instructions you cannot rely on register contents to be preserved from one assembly instruction to another if there are C statements in between. On the other hand, the C compiler will sometimes recognize when it can assign a variable in a register, thus making the contents vulnerable to being clobbered by in-line assembly code. If you have a reason to write complex in-line assembly code, you should put the code in a separate, assembly-only function.

Exercise 7-2

How are function arguments passed to an assembly language program in your compiler?

Exercise 7-3

Does your compiler allow in-line assembly code? If so, how is it done?

Table 7-4 C Compiler Assembly Language Interface

	CodeWarrior	COSMIC C
Multiple arguments passed to the function		
Fixed number of parameters	Pushed onto the stack in left-to-right order. If possible, the last argument is transferred in a register. If it cannot be, it is pushed onto the stack (Pascal convention).	All arguments pushed onto the stack in right-to-left order (normal C convention).
Variable number of parameters	Pushed onto the stack in right-to-left order up to the last argument. If possible, the last argument is transferred in a register. If it cannot be, it is pushed onto the stack (C calling convention).	
Single arguments passed into the function		
Byte argument (char, unsigned char)	Register B	Bytes are extended to 16 bits (short) and returned in Register D
16-bit argument (int, unsigned int, pointer)	Register D	Register D
3 bytes (far data pointer)	Register B (high byte): Register X (low word)	
32-bit argument (double, long, float)	Register X (high word): Register D (low word)	Register X (high word): Register D (low word)
Single arguments returned by the function		
Byte argument (char, unsigned char)	Register B	Bytes are extended to 16 bits (short) and returned in Register D.
16-bit argument (int, unsigned int, pointer)	Register D	Register D
3 bytes (far data pointer)	Register B (high byte): Register X (low word)	
32-bit argument (double, long, float)	Register X (high word): Register D (low word)	Register X (high word): Register D (low word)
Registers on return	Except for the return value in the registers defined above, registers and the condition code registers are undefined.	Except for the return value in Register D, registers and the condition code registers are undefined.

7.5 Bits and Bytes: Accessing I/O Registers

Sometimes it is better to read or write a complete byte; in other cases it is better to access individual bits in an I/O port or memory location.

There are many control registers and control bits in microcontrollers that must be initialized to enable and disable hardware features. The choices for setting and resetting control register bits in most microcontrollers: use load and store instructions to write the whole byte, and bit-set and bit-clear instructions to set and reset bits.

When you write to a byte with an assembly language store instruction, you are writing all bits in the byte. Microcontrollers with bit-set and bit-clear instructions modify only the bits specified by the instruction. In general, when you enable or disable control bits it is better to use the bit-set or bit-clear instruction so that you do not modify other bits in the control register that may be set in some other part of the program.

In C there are a variety of ways to access specific memory locations such as I/O registers. Each register has a specific address in memory and the C code must allow reading from and writing to these addresses. The C bit-wise operators AND, OR, and exclusive-OR (&, |, and ^) may or may not use bit-set or bit-clear instructions, depending on your compiler.

Byte Addressing

Example 7-3 shows how to address a port when it is appropriate to access a byte, such as when you are writing to or reading from an I/O port or resetting flags in a flag register.

The line of code

```
#define PORTB  (*(volatile unsigned char *) 0x0001)
```

declares the `PORTB` to be the contents of the volatile unsigned char pointer 0x0001. This line of code works as follows:

```
volatile unsigned char
```

declares the unsigned char to be a volatile;

```
(volatile unsigned char *)
```

defines a pointer to this type;

```
(volatile unsigned char *) 0x0001
```

sets the pointer value to 0x0001, and

```
(*(volatile unsigned char *) 0x0001)
```

defines `PORTB` to be the contents of this memory address.

A second, portable way to address a port at a fixed memory location is to declare a pointer and then use pointer addressing. See Example 7-4.

The line of code

```
#define p_PORTB  (volatile unsigned char *) 0x0001
```

declares `p_PORTB` to be a pointer to a volatile unsigned char and assigns the value 0x0001 to `p_PORTB`.

```
*p_PORTB = 26;
```

writes 26 to 0x0001.

Many compilers have an extension that allows us to assign global variables to specific addresses. Example 7-5 shows how this is done in the CodeWarrior compiler.

Example 7-3 Port Addressing for Byte Accesses in C

```
/*************************************************************/
/* Declare PORTB to be the contents of a memory location
```

```
 * pointed to by the volatile unsigned char pointer 0x0001 */
#define PORTB  (*(volatile unsigned char *) 0x0001)
/*************************************************************/

void main(void) {
    PORTB = 26;       /* Write to PORTB */
}
```

Example 7-4 Using Pointer Addressing for Port Addressing for Byte Accesses

```
/*************************************************************/
/* Declare PORTB to be the contents of a memory location
 * pointed to by the volatile unsigned char pointer 0x0001 */
#define p_PORTB  (volatile unsigned char *) 0x0001
/*************************************************************/

void main(void) {
    *p_PORTB = 26;       /* Write to PORTB */
}
```

Example 7-5 Using a Compiler Extension to ANSI C for Port Addressing for Byte Accesses

```
/*************************************************************/
/* Define PORTB to be a volatile unsigned char
 * at address 0x0001 */
volatile unsigned char PORTB @0x0001;
/*************************************************************/

void main(void) {
    PORTB = 26;       /* Write to PORTB */
}
```

Bit Addressing

ANSI C provides a way to use a bit-field structure to define bit fields that are addressed individually. This is ideal for microcontrollers with bit-set and bit-clear operations. Example 7-6 shows an 8-bit field defined for the volatile variable PORTB located at 0x0001. When compiled, statements such as `PORTB.BIT0 = 1` are conveniently treated as bit-set and bit-clear instructions.

Example 7-6 Bit Addressing

```
/**************************************************************/
/* Define a bitfield type as unsigned int */
typedef unsigned int BITFIELD;
/**************************************************************/
/* Define an eight-bit field for the PORT */
typedef struct {
  BITFIELD BIT0 : 1;
  BITFIELD BIT1 : 1;
  BITFIELD BIT2 : 1;
  BITFIELD BIT3 : 1;
  BITFIELD BIT4 : 1;
  BITFIELD BIT5 : 1;
  BITFIELD BIT6 : 1;
  BITFIELD BIT7 : 1;
} PORT;
/* Define PORTB to be a volatile structure of bits at 0x0001 */
#define PORTB (*(volatile PORT *) 0x0001)

/**************************************************************/
/* Define a different way to access the bits */
#define PORTB_BIT1 PORTB.BIT1

/**************************************************************/

void main(void) {
  /* These instruction may generate bit-set and bit-clr
   * instructions */
  /* Strobe PORTB bit-0 */
  PORTB.BIT0 = 1;
  PORTB.BIT0 = 0;
  /* Strobe PORTB bit-1 */
  PORTB_BIT1 = 1;
  PORTB_BIT1 = 0;
}
```

Byte and Bit Addressing

It is convenient to be able to address a register as a byte in some situations and as a bit in others. For example, consider the hardware shown in Figure 7-2. Eight switches are connected to bits 7-0 on Port A and eight LEDs to bits 7-0 on Port B. It would be convenient to read all eight switches at once or, perhaps, to test each one individually. Similarly, controlling all eight LEDs at a time or individually can be done.

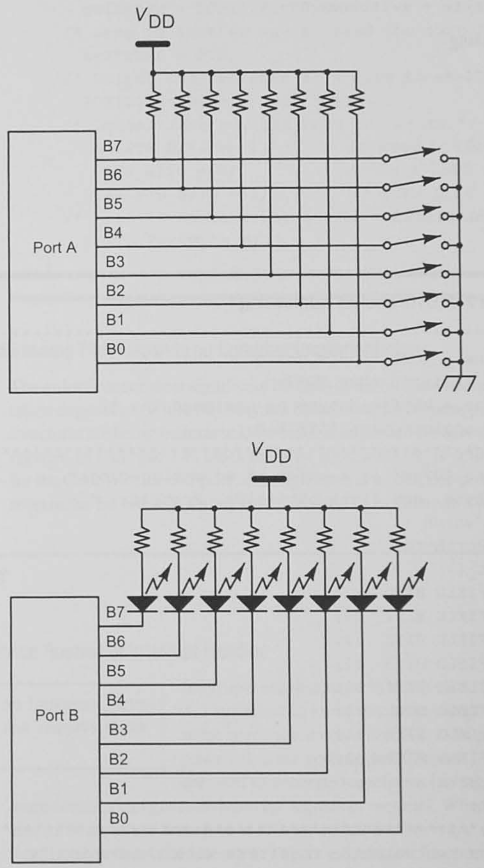


Figure 7-2 Hardware configuration to demonstrate bit addressing.

A program to accomplish this is shown in Example 7-7. A *union* of two storage classes, `unsigned char PortByte` and an 8-bit *bit-field structure* `PortBits`, is defined. Two registers of this type, `PORTA` and `PORTB`, are declared and located appropriately. One can access the register as a byte as in the statement

```
switches = PORTA.PortByte;
```

or

```
PORTB.PortByte = switches;
```

or as bits by using

```
if (PTA_BIT4 == 1)
```

and

```
PTB_BIT0 = 0;
```

or

```
PORTB.PortBits.BIT3 = 1;
```

Example 7-7 Byte and Bit Addressing

```
/*********************************************************/
/* Define a port type = unsigned char */
typedef unsigned char PORT;
/* Define a bitfield type as unsigned int */
typedef unsigned int BITFIELD;
/*********************************************************/
/* Define IOPort as a union of an unsigned char,
 * PortByte, and eight bitfields, PortBits  */
typedef union {
  PORT PortByte;
  struct {
    BITFIELD BIT0  :1;
    BITFIELD BIT1  :1;
    BITFIELD BIT2  :1;
    BITFIELD BIT3  :1;
    BITFIELD BIT4  :1;
    BITFIELD BIT5  :1;
    BITFIELD BIT6  :1;
    BITFIELD BIT7  :1;
  } PortBits;
} IOPort;
/*********************************************************/
/* Locate two volatile registers with this union */
#define PORTA (*(volatile IOPort *) 0x0000)
#define PORTB (*(volatile IOPort *) 0x0001)
/*********************************************************/
/* Define some different ways to access the registers */
#define PTA PORTA.PortByte          /* PTA is a byte */
#define PTB_BIT0 PORTB.PortBits.BIT0  /* PTB_BIT0 is a bit */
#define PTA_BIT4 PORTA.PortBits.BIT4  /* PTA_BIT4 is a bit */
void main(void) {
  volatile unsigned char switches;
/*********************************************************/
  /* Read from the switches on Port A*/
```

```
    switches = PORTA.PortByte;
  /* Here is another way to read the switches */
    switches = PTA;
  /* Output the switches as a byte to the LEDs on Port B*/
    PORTB.PortByte = switches;
  /* Demonstrate one bit read and write */
    if (PTA_BIT4 == 1)        /* Read bit-4 switch */
      PTB_BIT0 = 0;           /* Turn LED0 on */
    else PTB_BIT0 = 1;        /* Turn LED0 off */
  /* Here is another way access a bit field */
    PORTB.PortBits.BIT3 = 1;
}
```

Caution on Bit Addressing That Depends on Compiler Implementation

The order (least or most significant bit first) of the bit-field addressing just described is implementation dependent. While this may not be an issue for some applications where internally defined structures are being maintained, the order is important for accessing the defined control bits in the register. In Example 7-6, bit-0, the least significant bit, is defined first. This is the default order for the CodeWarrior compiler, although there is a compiler-predefined macro that can be used to reverse the bit order. With any compiler, you must determine what order is being used.

7.6 Interrupts

The Interrupt Service Routine or Interrupt Handler

An *interrupt* is an important, asynchronous event that requires immediate attention.

Interrupts are discussed in much more detail in Chapter 10; at this point, it is sufficient to know that an interrupt is an important event signaled by some hardware mechanism. For example, many microcontrollers have a powerful timer system (see Chapter 14) from which periodic interrupts are used to generate waveforms having a specific frequency. Other hardware features can generate interrupt signals or requests. Whenever one of these events happens, the software must branch to a routine called the *interrupt service routine* (ISR) or interrupt handler to process the interrupt. This is a hardware function call, and it is processed like any function call except that no arguments may passed into or back from the function except for globally declared variables. In addition to this restriction, a special return from interrupt instruction is used instead of the return from subroutine instruction. Furthermore, in some processors all registers are pushed onto the stack before entering and pulled while leaving.

An interrupt service routine function uses a keyword, such as *interrupt*, to cause the compiler to generate the correct code when the interrupt request occurs. You may use automatic variables whose lifetime is the life of the function execution. Sometimes, however, an interrupt service routine must modify a variable each time the ISR is entered. In such a case, the variable must be declared static so that it "lives" from one interrupt to another. Finally, in all interrupt service routines the interrupting source, often a flag or bit in a register, must be reset before one leaves the routine.

Locating the Interrupt Service Routine

Many microcontrollers use an interrupt vector to find the correct interrupt service routine when the interrupt request is received. The interrupt vector is simply the address of the start of the interrupt service routine, and each interrupting source has a specific address that stores its own vector. It is our job to provide the linker program with the starting address so it can be placed in the correct vector location. This can be done in a variety of ways, depending on your compiler.

As we will discuss more in Chapter 10, all interrupt vectors should be initialized to an address and not left uninitialized. This allows you to catch unexpected interrupts that may occur. You may wish to consider lighting an error condition LED to indicate when these unplanned-for interrupts occur because it means that something is going wrong in the software or hardware.

Exercise 7-4

Does your microcontroller signify an interrupt service routine or interrupt handler in your C programs?

7.7 Conclusion and Chapter Summary Points

In this chapter we have discussed the use of the C programming language for programming embedded systems by means of microcontrollers.

- There are two kinds of memory in a microcontroller system—RAM and ROM.
- An embedded system's program is in ROM with variables in RAM.
- A desktop system's program and data are in RAM.
- The embedded system engineer must know the microcontroller's memory map to be able to locate the program and data correctly.
- The ANSI C standard does not provide all of the features needed in an embedded system (e.g., direct memory access, interrupt control, in-line assembly statements).
- Compilers that are ANSI C compliant also offer extensions necessary for embedded systems.

7.8 Bibliography and Further Reading

Kernighan, B. W., and D. M., Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, 1988.

7.9 Problems

Explore

7.1 What ANSI C extensions are supported by your C compiler? [a]

7.2 What assembly code does your compiler produce for the following statements? Assume that data1 is unsigned char type. [a]

```
a. data1 = data1 | 0x01;  /* Set bit 0 */
b. data1 |= 0x01;  /* Set bit 0 */
c. data1 = data1 & 0x01;  /* Reset bits 7 - 1 */
d. data1 &= ~0x01;  /* Reset bit 0 */
e. data1 ^= 0xFF;  /* Toggle all bits in data 1 */
```

7.3 How do you use your compiler to address a specific memory location to access a particular control register? [a]

7.4 How many bits does your compiler allocate for char, int, and long data types? [a]

7.5 What does the start-up() code do for your microcontroller? [a]

7.6 Give the memory addresses used in your microcontroller for the following: [a]
a. Data memory
b. Program memory
c. Control registers

7.7 What must you do for your compiler to be able to use interrupts? [a]

Stimulate

7.8 Write a program in C to reverse the order of 0x20 bytes in a buffer. Assume the buffer is in memory locations DATA[0]–DATA[31]. [c]

7.9 Write a C function to search a null-terminated string of characters for a specific substring and to return the address of the start of the substring. The input to the subroutine is to be the starting address of the string to be searched, the starting address of the substring to be searched for, and the number of characters in the substring. If the substring is found, return the address of the first character in the search string; otherwise return an address of 0x0000. [c]

Challenge

7.10 Determine how your compiler treats volatile variables by writing a program similar to Example 7-1. [a]

7.11 Write a program for your microcontroller in C and then in assembly (or vice versa) to find the largest of thirty-two 8-bit unsigned numbers in 32 successive memory locations. Place the answer in the next available location. [c]

7.12 Write a program for your microcontroller in C and then in assembly (or vice versa) to find the largest of thirty-two 8-bit two's-complement numbers in 32 successive memory locations. Place the answer in the next available location. [c]

7.13 There are 4 bytes of data in variable data array DATA[0]–DATA[1]. Write a program in C to count the number of 1s in these bytes. Place the result in NUM_ONES. [c]

7.14 Write a program in C to compute factorial 8. Store the result in a 2-byte memory location in RAM memory. [c]

7.15 An 8-bit signed/magnitude number system is in use. Write assembly or C subroutines or functions for the following: [c]

a. Add two 8-bit signed/magnitude numbers.
b. Subtract two 8-bit signed/magnitude numbers.
c. Multiply two 8-bit signed/magnitude numbers.
d. Divide two 8-bit signed/magnitude numbers.

Reflect on Learning

7.16 What have you learned about C programming for embedded systems that is different from programming in a desktop environment?

7.17 List five new things you have learned about using C to program a microcontroller for an embedded application.

8 Debugging Microcontroller Software and Hardware

Objectives

This chapter describes debugging strategies and techniques useful in helping you find problems in your programs.

8.1 Introduction

By now you will have experienced writing and running simple assembly language or C programs on your laboratory equipment. You have also probably experienced the programmer's nightmare: your program does not work; perhaps it does not even appear to run. It is time for some program debugging.

8.2 Program Debugging

Program debugging is like solving a mystery. We start the program, fully expecting it to work perfectly, and it does not. Often, when beginning students are asked, "What is your program doing?" they respond "Nothing!" The computer *cannot* be doing nothing; it is doing something all the time: fetching opcodes, executing them, incrementing the program counter, and fetching the next opcode. Remember that *you* are responsible for the opcodes the computer is executing, and you should know what every instruction does at every step along the way. You must do some detective work to find the difference between what you expect the program to be doing and what it is actually doing. Debugging is the process of finding the clues and interpreting them to find the problem.

Programs that are not working properly should be *analyzed* to find out what they are doing before anyone tries to fix them.

There are two approaches to fixing bugs in programs. The first is a *synthesis* approach in which you try to fix the problem by changing the code somewhere. This is wrong! You must find out what the program is doing before you can fix it. Thus the second approach: *analyzing the problem*. You first find out *what* is the program doing, then *why* it is doing that. Probably by then you will have enough clues to be able to *fix* it.

Programs have only two parts—the data and the logic. The program inputs data values, stores and manipulates them, and outputs the data in some form. The program's logic determines the sequence in which program steps are executed and how the data are manipulated. Most program bugs are in the logic. We mean for the computer to do one thing but we have programmed it to do another. Normally the data affect the program's flow, and this can help us find the debugging clues. When we use the analytical debugging technique, we are trying to match what we think the program should do for a particular input data set with what the program actually is doing.

Figure 8-1 shows analytical debugging. We choose an input data set and predict what the program will do with these data at each step of the program and what the program will do next. This is a model of what the program should do. Now run the program and, using the tools described shortly, look for data values and program steps that differ from the model. Once we have found out where the program deviates from the model, we are well on our way to finding out why it is going wrong and what will be needed to fix it.

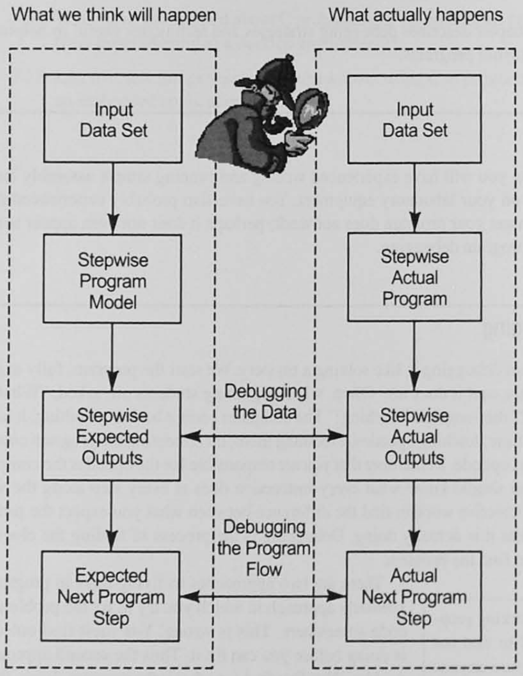


Figure 8-1 Analytical debugging model.

8.3 Debugging Your Code

Design for Debugging

Designing your code and including comments are critical components in the creation of programs that can be tested and debugged.

Shortly after writing their first programs, most software developers realize that usually, unless the program is very, very small, there will be some problems that must be found and corrected. Projects never go as well as we would like, and debugging our software always takes some time and effort.

To produce code that can be debugged, we must use top-down design methods and structured programming techniques. Code that is disorganized or is produced by a programmer who does not understand the specifications usually hides more problems than can be found. Indeed, many large systems are never completely free of program bugs.

Comments are critical to the development and debugging efforts. Most programmers, novice and experienced alike, do not like to comment their programs. Beginning programmers, who may be concentrating on just learning the language, are loath take time away from producing code. Chapter 3 shows an approach to comments that follows the top-down design philosophy. Some design must be done, at least to some level, and the design statements are included comments in your program. These design comments, which tell what the code is to do, are followed by the code that implements the design. Of course, the code itself should have comments as well explaining how the code is implementing the design.

After we have written our code, we turn to the debugging phase. The design comments and the code comments help us to understand what is supposed to be done (it helps us develop "What we think will happen" from Figure 8-1). The code comments help us understand how the code is supposed to work. Compare Examples 8-1 and 8-2. Which would you prefer to work with if you were assigned to find a problem in the code? If you find problems that require changes, make sure you update the comments!

Another useful strategy is to include comments to remind you of things to test when it comes time to test and debug the code. See Example 8-3.

Example 8-1 Code with No Design Comments

```
/**********************************************************
 * Hex keypad scanning module
 *********************************************************/
/* Define Grayhill Series 96 4x4 keypad */
#define NUM_ROWS 4    /* Number of rows */
#define NUM_KEYS 16    /* Number of keys */
/*********************************************************/
/* Define constants */
#define ROW3 0x0e
#define ROW2 0x0d
#define ROW1 0x0b
#define ROW0 0x07
#define OUTPUTS 0x0f
#define INPUTS 0xf0
```

```
#define COL3 0x70
#define COL2 0xb0
#define COL1 0xd0
#define COL0 0xe0
#define KEY_MASK 0xf0
#define NO_KEYS 0xf0
#define END_MARK 0xff
/*****************************************************/
/* Define arrays to store the scan codes, key codes and a
 * lookup table for the return value */
unsigned char Row_Codes[] = {
 ROW3,
 ROW2,
 ROW1,
 ROW0
};
unsigned char Good_Codes[ ] = {
 COL3 | ROW3,
 COL2 | ROW3,
 COL1 | ROW3,
 COL0 | ROW3,
 COL3 | ROW2,
 COL2 | ROW2,
 COL1 | ROW2,
 COL0 | ROW2,
 COL3 | ROW1,
 COL2 | ROW1,
 COL1 | ROW1,
 COL0 | ROW1,
 COL3 | ROW0,
 COL2 | ROW0,
 COL1 | ROW0,
 COL0 | ROW0,
 END_MARK
};
/* User defined key codes. These are ASCII. */
unsigned char Key_Codes[ ] = {
 "123A456B789C*0#D"
};
/*****************************************************
 * Define the ports on the microcontroller to connect to the
 * keypad
 *****************************************************/
#define DDRAD (*(volatile unsigned char *) 0x0272)
#define PERAD (*(volatile unsigned char *) 0x0274)
#define ATDDIEN (*(volatile unsigned char *) 0x008D)
#define PTAD (*(volatile unsigned char *) 0x0270)
/*****************************************************/
/************* Module Start ****************************/
```

```
unsigned char hex_key_scan ( void ){
/*****************************************************/
 unsigned char key_hit;
 unsigned char col_code;
 unsigned char scan_code;
 unsigned char key_code;
 unsigned char i;
/*****************************************************
 * Initialize your microcontroller's I/O port connected
 * to the keypad.
 *****************************************************/
/* Initialize the PortAD bits 3-0 for output */
  if (( DDRAD & OUTPUTS ) != OUTPUTS ){
   DDRAD |= OUTPUTS;
   PERAD |= INPUTS;
   ATDDIEN |= INPUTS;
  }
/*****************************************************/
/* Output each row code and read the col code each time.*/
 i = 0;
 col_code = NO_KEYS;
 while ( (i < NUM_ROWS ) & (col_code == NO_KEYS)){
 /* Output the row scan code */
   PTAD = Row_Codes[ i ];
   ++i;
 /* Read the scan code */
   scan_code = PTAD;
   col_code = scan_code & KEY_MASK;
 }
 if ( col_code == NO_KEYS ){
  key_hit = 0;
 } else {
  i = 0;
  key_code = 0;
 while ((key_code != END_MARK) && (key_code != scan_code)) {
  key_code = Good_Codes[ i ];
  ++i;
 }
 if ( key_code == END_MARK ) {
  key_hit = 0;
 } else {
  --i;
  key_hit = Key_Codes[ i ];
 }
 }
 return( key_hit );
}
```

Example 8-2 Code with Design Comments

```
/************************************************************
 * Hex keypad scanning module
 *  unsigned char hex_key_scan( void );
 * This module scans a 16-key keypad
 * attached to Port AD. It returns an unsigned char ASCII code
 * for the key pressed. It returns the first key pressed
 * when scanning, It does not check for multiple keys pressed
 * at the same time and it does not debounce key strokes.
 * Author: F. M. Cady
 * Source File: hex_keypad.c
 * Revision: 1
 * Revision date: 1 February 2009
 ***********************************************************/
/* Hardware Definitions */
/**********************************************************
 * Port AD bits:
 *  PAD-3 - PAD-0: Output: Scan row scan codes
 *  PAD-7 - PAD-4: Input: Column code
 *          |     Col3 Col2 Col1 Col0
 *      Row | Col  Code
 * Row Code |1111 0111 1011 1101 1110
 * ---------|-------------------------------
 * 3 1110   |None  1    2    3    A   Key
 * 2 1101   |None  4    5    6    B   Pressed
 * 1 1011   |None  7    8    9    C
 * 0 0111   |None  *    0    #    D
 * ---------|-------------------------------
 ***********************************************************/
/* Define Grayhill Series 96 4x4 keypad */
#define NUM_ROWS 4    /* Number of rows */
#define NUM_KEYS 16    /* Number of keys */
/* Define where they are connected to the microcontroller
 * PTAD Bit  Grayhill Keypad Pin
 *  0    1
 *  1    2
 *  2    3
 *  3    4
 *  5    6
 *  6    7
 *  7    8
**********************************************************/

/**********************************************************/
/* Define constants */
#define ROW3 0x0e    /* Row 3 scan code */
#define ROW2 0x0d    /* Row 2 scan code */
```

```
#define ROW1 0x0b      /* Row 1 */
#define ROW0 0x07      /* Row 0 */
#define OUTPUTS 0x0f   /* Row outputs */
#define INPUTS 0xf0    /* Col inputs */
#define COL3 0x70      /* Col 3 scan code */
#define COL2 0xb0      /* Col 2 */
#define COL1 0xd0      /* Col 1 */
#define COL0 0xe0      /* Col 0 */
#define KEY_MASK 0xf0
#define NO_KEYS 0xf0  /* Code for no keys pressed */
#define END_MARK 0xff /* End of Good_Codes array */
/************************************************************/
/* Define arrays to store the scan codes, key codes and a
 * lookup table for the return value */
unsigned char Row_Codes[] = {
 ROW3,     /* Row 3 scan code */
 ROW2,     /* Row 2 scan code */
 ROW1,     /* Row 1 */
 ROW0      /* Row 0 */
};
/************************************************************
 * This lookup table contains the 8-bit scan codes for all
 * keys on the keypad
 ***********************************************************/
unsigned char Good_Codes[ ] = {
 COL3 | ROW3, /* "1" 0x7e */
 COL2 | ROW3, /* "2" 0xbe */
 COL1 | ROW3, /* "3" 0xde */
 COL0 | ROW3, /* "A" 0xee */
 COL3 | ROW2, /* "4" 0x7d */
 COL2 | ROW2, /* "5" 0xbd */
 COL1 | ROW2, /* "6" 0xdd */
 COL0 | ROW2, /* "B" 0xed */
 COL3 | ROW1, /* "7" 0x7b */
 COL2 | ROW1, /* "8" 0xbb */
 COL1 | ROW1, /* "9" 0xdb */
 COL0 | ROW1, /* "C" 0xeb */
 COL3 | ROW0, /* "*" 0x77 */
 COL2 | ROW0, /* "0" 0xb7 */
 COL1 | ROW0, /* "#" 0xd7 */
 COL0 | ROW0, /* "D" 0xe7 */
 END_MARK   /* End marker */
};
/************************************************************
 * This lookup table returns the ASCII code for the key.
 ***********************************************************/
/* User defined key codes. These are ASCII. */
unsigned char Key_Codes[ ] = {
  "123A456B789C*0#D"
  };
  /*******************************************************
   * Define the ports on the microcontroller to connect to the
   * keypad
   *******************************************************/
  /* Data dir reg */
  #define DDRAD (*(volatile unsigned char *) 0x0272)
  /* Pullup enable */
  #define PERAD (*(volatile unsigned char *) 0x0274)
  )/* ATD Input */
  #define ATDDIEN (*(volatile unsigned char *) 0x008D
  /* PTAD Data */
  #define PTAD (*(volatile unsigned char *) 0x0270)
  /*******************************************************/
  /************* Module Start *****************************/
  unsigned char hex_key_scan ( void ){
  /*******************************************************/
  /* 8-bit ASCII code for the key or 0 */
   unsigned char key_hit;
  /* 4-bit col code ret from keypad scan */
   unsigned char col_code;
  /* 8-bit col_code, row_code */
   unsigned char scan_code;
  /* 8-bit scan_code for the key pressed */
   unsigned char key_code;
  /* An indexing variable */
   unsigned char i;
  /*******************************************************/
   /*****************************************************
    * Initialize your microcontroller's I/O port connected
    * to the keypad.
    *****************************************************/
  /* Initialize the PortAD bits 3-0 for output */
  /* Check to see if the port has been set up */
  /*  IF the data direction register is not set
   *  to output on the ROW_OUT bits */
    if (( DDRAD & OUTPUTS ) != OUTPUTS ){
    /* Then initialize the data direction register,
     * enable the input pull-ups and enable the ATD
     * input bits */
     DDRAD |= OUTPUTS; /* Set the Data Direction Register */
     PERAD |= INPUTS; /* Pull ups enabled */
     ATDDIEN |= INPUTS;/* ATD inputs enabled */
   }
  /*******************************************************/
  /* Output each row code and read the col code each time.*/
   i = 0;
```

```
   col_code = NO_KEYS;
   while ( (i < NUM_ROWS ) & (col_code == NO_KEYS)){
   /* Output the row scan code */
    PTAD = Row_Codes[ i ];
    ++i; /* Increment the pointer. Note: This code is here
          * to cause a little delay after outputting the row
          * code before reading the scan code. This is often
          * done to give the hardware time to settle to the
          * correct value. */
  /* Read the scan code */
   scan_code = PTAD;
   col_code = scan_code & KEY_MASK;
  }
  /* If the col_code is NO_KEYS, return zero */
  if ( col_code == NO_KEYS ){
   key_hit = 0;
  } else {
   /* Otherwise, find the key that was pressed */
   /* The variable scan_code has the col and row */
   /* code. Just scan through a look up table to */
   /* find a match and return the user defined code.*/
   i = 0;
   key_code = 0;
  while ((key_code != END_MARK) && (key_code != scan_code)) {
   key_code = Good_Codes[ i ];
   ++i;
  }
  if ( key_code == END_MARK ) {
   /* Must have reached the end of the table and not
    * found a match. Return zero */
    key_hit = 0;
  } else {
   /* Retrieve the key */
   --i;
   key_hit = Key_Codes[ i ];
  }
   }
  return( key_hit );
}
```

Example 8-3 Comments Included to Help with Testing and Debugging

```
/**********************************************************
 * This module calculates the square root of a signed 16-bit
 * signed integer.
 * Reminder for testing:
 * Test the module's behavior when a negative number is input. It
 * should return the correct error code.
 *************************************************************/
```

Code Walkthroughs

Probably the single most effective debugging technique is to remove bugs before you put them into the firmware. A *code walkthrough* is often used (although not used often enough, it seems) to eliminate problems before they end up in the code.

Code walkthroughs are sometimes called peer code reviews. The code developer invites other technical experts, both familiar and unfamiliar with the particular project, to review the source code. The object is to look at the code line by line to develop the program model suggested in Figure 8-1, and to look for problems and for better ways to accomplish the programming task.

Code walkthroughs are very effective in increasing the quality of the code and eliminating problems before they occur. The developer can learn about other ways to accomplish the tasks and, as a side benefit, reviewers can learn about techniques that may help them in their assignments. The result of a code walkthrough is better performance of the developer, the program, and the application.

Top-Down Debugging

Top-down design leads to top down debugging.

Top-down design is discussed in Chapter 3. If you follow these principles and design in levels of increasing detail, but postpone details until later levels, the structure that Figure 8-2 shows might be the result. Let us assume that function_a, function_b, and function_c have been completed, and we are working on function_f. Other engineers on the project have been assigned to generate function_d, function_e, and function_g. By using the complete design structure to test and debug all functions completed to date, and including stubs for those functions not completed, we can test and debug the entire program repeatedly as we complete the functions now coded as stubs. A stub, then, is a dummy function that returns a value to use in testing and debugging. For example, assume that function_d is to return a value from an analog-to-digital converter. In the early days of the development, we might not even have had the A/D hardware, and so naturally delayed writing that code. The function_d_stub, then, can serve as a placeholder for the actual code and can return a representative value for function_b and function_c.

The Debugging Plan

Your debugging plan should be to first find the section of the code where the problem is occurring and then find the problem.

A debugging plan will help isolate where the problem is occurring and then find out what is going wrong. Well-designed programs consist of separate, independent sections of code written to do a particular function. For example, a program may simply input data, process it, and output it as shown by the flowchart in Figure 8-3. How do you know if the program is performing correctly? You must choose test data for which you know the correct output. Using these data, you can look for the problem. Let us assume that

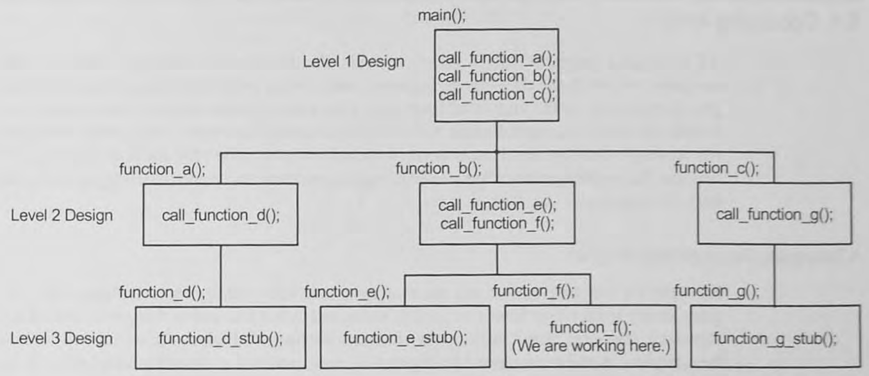


Figure 8-2 Top-down testing and debugging.

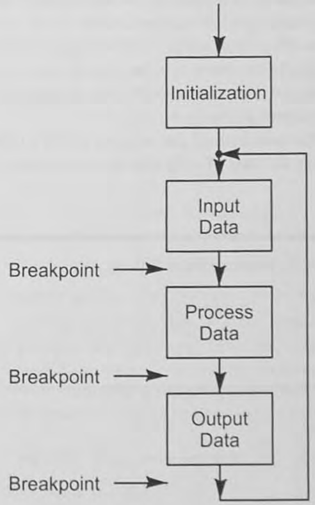


Figure 8-3 Typical program flowchart.

a problem exists in the program somewhere. Your first step is simply to analyze the program output. This can sometimes give clues about where the program is going wrong. If not, plan to set breakpoints after each section and inspect the data to see where it deviates from what is expected. This will isolate the problem area. You can now move your debugging strategy into the offending block of code and continue the process.

8.4 Debugging Tools

All debugging programs offer a variety of tools. The features in a debugger depend on the computer on which the debugging program is run. Today, personal computer–based debuggers, particularly for the high-level languages, have many features that can either operate in a simulation mode or interact directly with the microcontroller hardware. Other, older debuggers run on simple development boards or single-board computers and offer more limited range of features. These debuggers, called *monitor* programs, contain rudimentary debugging tools and basic I/O functions.

A Debugging Demonstration Program

Example 8-4 is a program we will use to demonstrate some debugging techniques. This program simply reads a byte from a `source[]` string and calls a function to change the case of any alphabetic character. Nonalphabetic characters are unchanged. To debug it, we need to develop the analytical model as suggested by Figure 8-1. Array addressing is used to index through the array. Note that the `source` and `destination` arrays and the `returned_byte` character are static arrays. This is a trick you can use to help your debugging. It makes the compiler store the data in RAM, where it is easy to display on your debugger memory screen. If they have been declared automatic variables, which could have happened, it would be much harder to find them in memory. Another trick to make debugging a little easier is to use array index addressing, as we have done here. Code that is more efficient would probably use pointer addressing.

The `change_case()` function checks to ensure that the `input_char` is either uppercase or lowercase before exclusive-ORing the character with 0x20 to change the case. Any input that does not meet this criterion is returned unchanged.

We expect the program to transfer each byte of the `source` string to the `destination` string one byte at a time and to change the case of each alphabetic character.

Example 8-4 Sample Program to Demonstrate Debugging

```
/************************************************************
 * Sample program to demonstrate a debugger program.
 * The program simply reads a null terminated string from
 * one memory buffer, changes upper to lower and lower to upper
 * case, and stores it in another memory buffer.
 ***********************************************************/

char change_case( char );  /* Change the case of the input */

 void main(void) {
 static char source[]={
   "This is a string!"
 };
 static char destination[ 20 ];
 static char returned_byte;
 static int i;
 static int j;
```

```
 /* Initialize the array pointers */
   i = 0;
   j = 0;
 /* While the end of the source string hasn't been found */
  while ( source[ i ] != 0 ){
   /* Do move a byte from source to destination */
      /* Get the byte */
      returned_byte = change_case( source[i] );
      /* Put it in the dest array */
      destination[ j ] = returned_byte;
      ++i;    /* Increment the array indices */
      ++j;
 }

 for(;;) {} /* wait forever */
 /* please make sure that you never leave this function */
}
/************************************************************
 * Demonstrate a function to change an uppercase character to
 * lowercase and vice versa.
 * char change_case( char );
 ***********************************************************/
 char change_case( char input_char ) {
  char output_char;
  /* If the input char is an alphabetic char, change the case */
     /* Check uppercase */
     if ( (input_char >= 'A') && (input_char < 'Z')
                               ||
     /* Check lowercase */
          (input_char >= 'a') && (input_char < 'z')) {
            output_char = input_char ^ 0x20;
          } else
            output_char = input_char;
 return( output_char );
 }
```

Debugging Program Flow and Logic

Tracing and *setting breakpoints* allow us to follow the program flow.

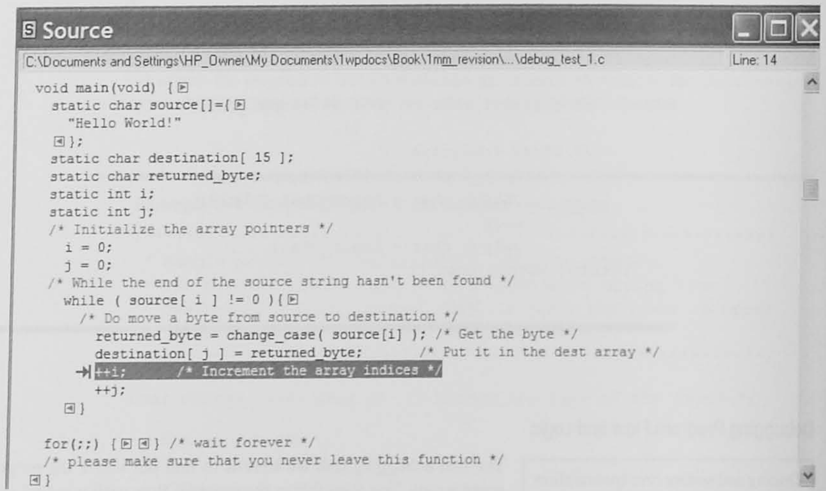
The first debugging task we have is to find out *where* the program is going wrong. You must follow the program flow until you find a deviation from the expected flow or an unexpected data modification. There are two ways to follow the program flow.

1. **Program trace:** Tracing is stepping through the program, one statement at a time. In the more powerful, high-level language debuggers, you may display data elements, including

the contents of memory locations and registers, while tracing the code. In less powerful, assembly language debuggers, the register set is shown at each step, but data elements in memory must be inspected manually.

2. **Breakpoints:** The program trace is a slow way to get through a program. It is quicker to find out where problems are occurring by running the program at full speed to a breakpoint. A breakpoint is a set of conditions that interrupt the program flow and return control to the debugging program. Normally breakpoints are set at program statements, but they also may be generated by a combination of other conditions. For example, in some debuggers, breakpoints can be generated when a particular data element becomes some specific value. In some systems, hardware breakpoint generators may create breakpoints when a condition or a set of bits on the computer's bus is detected.

Figure 8-4 shows a screen snapshot of the CodeWarrior debugger[1] with a breakpoint in the program from Example 8-4. Let us say we suspect that the change_case function is not properly changing the case of the input character. A good place to set a breakpoint is right after the function returns the `returned_byte` value. When the program hits the breakpoint, we can look at the value to see if it makes sense. Later, Figure 8-5 will show that when the breakpoint is hit after one pass through the loop, `source[0] = 'H'` and `destination[0] = 'h'`. Thus, we would conclude that the change_case function is working properly.[2]



Source

C:\Documents and Settings\HP_Owner\My Documents\1wpdocs\Book\1mm_revision\...\debug_test_1.c   Line: 14

```
void main(void) {
  static char source[]={
    "Hello World!"
  };
  static char destination[ 15 ];
  static char returned_byte;
  static int i;
  static int j;
  /* Initialize the array pointers */
    i = 0;
    j = 0;
  /* While the end of the source string hasn't been found */
    while ( source[ i ] != 0 ){
      /* Do move a byte from source to destination */
        returned_byte = change_case( source[i] ); /* Get the byte */
        destination[ j ] = returned_byte;       /* Put it in the dest array */
        ++i;        /* Increment the array indices */
        ++j;
    }

  for(;;) {} /* wait forever */
  /* please make sure that you never leave this function */
}
```

Figure 8-4 Setting breakpoints.

[1] Freescale Semiconductor, Inc.

[2] Well, there actually is a bug in `change_case`. We will ask you to find it in Problem 8.4.

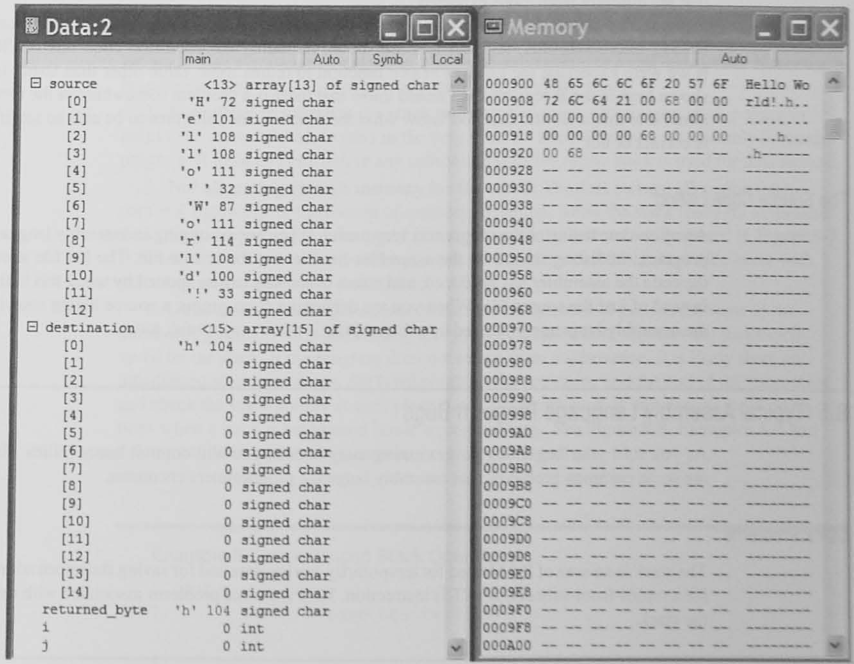
Debugging Data Elements

While you are following the flow of the program, you can observe data elements, both in the memory and, if you are programming in assembly language, the registers.

Memory: In high-level language debuggers, one can generally inspect any of the declared variables. Usually the display is formatted according to the type of declaration that has been made. In assembly language debugging monitors, the display of data elements is more crude, usually only in hexadecimal.

Figure 8-5 shows the CodeWarrior memory display. You may view memory contents as a formatted data value either in the Data pane or as hexadecimal values in the Memory pane. In this display we see that the program has transferred one byte from `source` to `destination`. The memory display shows both hexadecimal contents of memory as well as the ASCII character for those bytes that are printable.

Registers: When you know the data input, you should know the state of the registers at each step of an assembly language program. Assembly language debuggers usually display the contents of all the registers, including the condition code register, in a register window. While



Data:2

main   Auto   Symb   Local

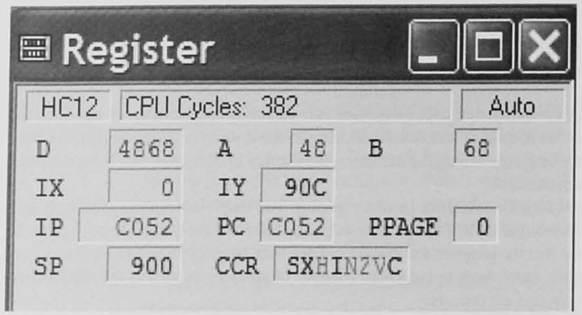
```
⊟ source          <13> array[13] of signed char
    [0]             'H' 72 signed char
    [1]            'e' 101 signed char
    [2]            'l' 108 signed char
    [3]            'l' 108 signed char
    [4]            'o' 111 signed char
    [5]             ' ' 32 signed char
    [6]             'W' 87 signed char
    [7]            'o' 111 signed char
    [8]            'r' 114 signed char
    [9]            'l' 108 signed char
    [10]           'd' 100 signed char
    [11]            '!' 33 signed char
    [12]                 0 signed char
⊟ destination     <15> array[15] of signed char
    [0]            'h' 104 signed char
    [1]                  0 signed char
    [2]                  0 signed char
    [3]                  0 signed char
    [4]                  0 signed char
    [5]                  0 signed char
    [6]                  0 signed char
    [7]                  0 signed char
    [8]                  0 signed char
    [9]                  0 signed char
    [10]                 0 signed char
    [11]                 0 signed char
    [12]                 0 signed char
    [13]                 0 signed char
    [14]                 0 signed char
  returned_byte    'h' 104 signed char
  i                      0 int
  j                      0 int
```

Memory

Auto

```
000900 48 65 6C 6C 6F 20 57 6F  Hello Wo
000908 72 6C 64 21 00 68 00 00  rld!.h..
000910 00 00 00 00 00 00 00 00  ........
000918 00 00 00 00 68 00 00 00  ....h...
000920 00 68 -- -- -- -- -- --  .h------
000928 -- -- -- -- -- -- -- --  --------
000930 -- -- -- -- -- -- -- --  --------
000938 -- -- -- -- -- -- -- --  --------
000940 -- -- -- -- -- -- -- --  --------
000948 -- -- -- -- -- -- -- --  --------
000950 -- -- -- -- -- -- -- --  --------
000958 -- -- -- -- -- -- -- --  --------
000960 -- -- -- -- -- -- -- --  --------
000968 -- -- -- -- -- -- -- --  --------
000970 -- -- -- -- -- -- -- --  --------
000978 -- -- -- -- -- -- -- --  --------
000980 -- -- -- -- -- -- -- --  --------
000988 -- -- -- -- -- -- -- --  --------
000990 -- -- -- -- -- -- -- --  --------
000998 -- -- -- -- -- -- -- --  --------
0009A0 -- -- -- -- -- -- -- --  --------
0009A8 -- -- -- -- -- -- -- --  --------
0009B0 -- -- -- -- -- -- -- --  --------
0009B8 -- -- -- -- -- -- -- --  --------
0009C0 -- -- -- -- -- -- -- --  --------
0009C8 -- -- -- -- -- -- -- --  --------
0009D0 -- -- -- -- -- -- -- --  --------
0009D8 -- -- -- -- -- -- -- --  --------
0009E0 -- -- -- -- -- -- -- --  --------
0009E8 -- -- -- -- -- -- -- --  --------
0009F0 -- -- -- -- -- -- -- --  --------
0009F8 -- -- -- -- -- -- -- --  --------
000A00 -- -- -- -- -- -- -- --  --------
```

Figure 8-5 CodeWarrior memory display.



```
HC12   CPU Cycles: 382        Auto
D    4868   A   48    B    68
IX      0   IY  90C
IP   C052   PC  C052  PPAGE 0
SP    900   CCR SXHINZVC
```

Figure 8-6 CodeWarrior register display.

tracing the program, you can watch the contents of the registers change and watch for values that are different from those expected.

Figure 8-6 shows the register display pane from the CodeWarrior debugger for a Freescale HCS12 microcontroller. All registers are shown including the condition code bits (CCR). If we were expecting the `change_case` function to return some value other than 0x68[3] (as shown here in the B register), we would know that there is a problem somewhere in the function. Note that it is vital that you know what the function should return to be able to see if it is correct or not.

The Source Code Listing

An up-to-date listing of the program is very useful. If you are debugging an assembly language program, the listing should be the assembler list file, not the source file. The list file shows the code the assembler has produced, and errors frequently can be spotted by using this listing instead of just the source file. When you are debugging C programs, a source listing showing the assembly language produced by a C compiler is often very useful, too.

8.5 Typical Assembly Language Program Bugs

As you start your beginning programming assignments, you will commit many follies. Here are some common problems that assembly language programmers encounter.

Stack Problems

The stack is an area of RAM used for temporarily storing data and for saving the return address for a return-from-subroutine (RTS) instruction. Here are some problems associated with using the stack.

[3] $0x68 = 104_{10}$ = 'h'.

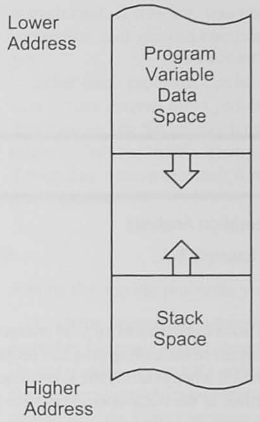


Figure 8-7 Program variable data and stack segments grow toward each other.

1. **Improper transfer to subroutines:** The return address from a subroutine must be on the stack. Use a branch-to-subroutine, a jump-to-subroutine, or a call instruction. Never use a branch or a jump that does not put the return address on the stack.

2. **Forgetting to initialize the stack pointer:** You must initialize the stack pointer to point to an area of RAM. Do this in the very first few lines of code in an assembly language program. It must be done before any subroutine is called or the stack is used for data storage.

3. **Not allocating enough memory for the stack:** The data storage allocation (static data in C) grows from the bottom of memory to the top, while the stack (used for automatic variables in C) grows from the top of memory toward the bottom, as illustrated in Figure 8-7. If the stack and data overlap, stack operations will write into data areas or vice versa with unknown, and usually dire, consequences.

4. **Unbalanced stack operations:** Make sure the number of pulls is the same as the number of pushes. This is particularly true in subroutines where registers are temporarily saved on the stack. If the program does not return from a subroutine, it is likely there are unbalanced stack operations. Set breakpoints at the beginning and the end of the subroutine and check the stack pointer at each place. Look for errors such as unbalanced stack operations when a stack is being used inside a program loop. See Figure 8-8. Examples 8-5 and 8-6 show unbalanced stack operation.

Example 8-5 Unbalanced Stack Operation

```
sub:
        pshx  ; Save the registers
        psha
;  . . .
        pshb  ; Temp save some data
```

```
6. ; . . .
7. ; . . .
8.       pula  ; Restore the registers
9.       pulx
10.      rts
```

Example 8-6 Unbalanced Stack Operation Analysis

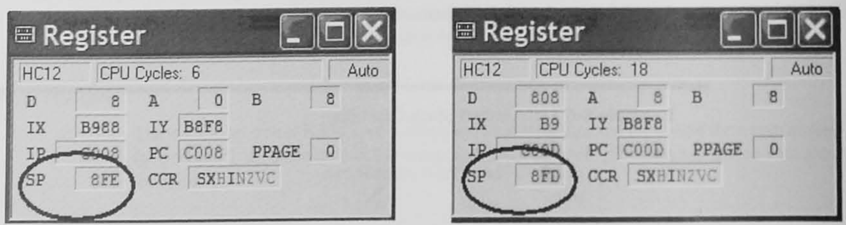
Analyze the stack problem illustrated in Example 8-5.

Solution

This is an unbalanced stack operation because there is an extra pshb instruction in the body of the subroutine. The subroutine will not return to the calling program properly. In Figure 8-8 we see that the stack pointer register is 0x8FE when entering the subroutine and 0x8FD when we are about to execute the `rts` instruction. If the stack operations were properly balanced these would be the same.

Finding Stack Problems

A program that executes properly up to a jump-to-subroutine instruction and then does not seem to return from the subroutine often suggests that there are problems with the stack. You can easily verify this by putting breakpoints at the subroutine jump and at the next instruction following the `jsr sub`. If you never hit the second breakpoint, you know there is a problem in the subroutine. With this simple step, you have been able to isolate *where* the problem is. The next step is to look for more clues. Set a breakpoint at the start of the subroutine and at the return-from-subroutine instruction at the end: for example, *line 2* and *line 10* in Example 8-5. Check the register display at all breakpoints to see if the stack pointer register is the same



```
HC12   CPU Cycles: 6          Auto
D     8   A   0   B   8
IX B988   IY B8F8
IP C008   PC C008   PPAGE 0
SP  8FE   CCR SXHINZVC
```

```
HC12   CPU Cycles: 18         Auto
D   808   A   8   B   8
IX   B9   IY B8F8
IP C00D   PC C00D   PPAGE 0
SP  8FD   CCR SXHINZVC
```

Figure 8-8 Register contents when entering and about to leave the subroutine.

in each case. If it is not, you know there is an unbalanced stack operation somewhere in the subroutine, and you can continue to look for clues until you find the trouble spot (see Figure 8-8).

Other stack problems can be caused by data overwriting the stack or stack overwriting the data. If you suspect these problems, and if your debugger allows it, you can write a specific data pattern into the uninitialized RAM memory before running the program (a fixed pattern might be 0x00 or 0xFF). Then, when problems occur, you can view the RAM memory to see if your data pattern is intact; if not, the stack or other data elements have used more locations than expected.

Register Problems

Among the register problems you are likely to encounter are the following.

1. **Using immediate addressing incorrectly:** One of the biggest problems that beginning assembly language programmers have is knowing how to use immediate addressing properly. Do not confuse immediate addressing with direct memory addressing. Remember that immediate addressing retrieves constant data from the memory location immediately following the operation code. Direct memory addressing retrieves data, which may be constant or variable, from some other memory location.

Finding problems with register contents is difficult. You must have a good program model and know what is expected in the registers at all times. Then, using trace or breakpoints, you watch the register display to see when a register's contents do not match your expectations.

2. **Using subroutines that wipe out registers:** Well-designed subroutines for assembly language programs do not modify registers that may be used in the calling program. Push all registers used in the subroutine onto the stack when you enter the subroutine, and pull them before returning to the calling program.

Example 8-7 shows subroutine code that incorrectly restores the registers when returning to the calling program. Because the stack is a last-in, first-out operation, the registers must be pulled in the reverse order of pushing. You can find this error easily by setting a breakpoint at the jump to subroutine and at the instruction just following the jsr. Comparing the contents of the registers at these two points will quickly show any problems.

Example 8-7 Improper Restoration of Register Contents

```
1. sub:
2.       pshx  ; Save the registers
3.       psha
4.       pshb
5. ; . . .
6. ; . . .
7.       pulx  ; Restore the registers
8.       pula
9.       pulb
10.      rts
```

3. **Transposed registers:** A difficult problem to find is one in which the operands for an instruction have been transposed. For example, if data are to be moved from the B to the A register, and the proper instruction is, say, `mov b, a`, it is very easy to transpose the register operands and write `mov a, b`. To find this error, trace the program, watch the registers, and compare them with what is expected.

4. **Not initializing pointer registers:** Register indirect and indexed addressing modes must have the register initialized with the address of the data.

5. **Not initializing registers and data areas:** Because the contents of registers and RAM memory are unknown when the computer's power is turned on, always initialize registers and data areas before using them. Do this at run time, not assembly time or load time.

6. **Using a 16-bit counter in memory:** Counters are used for many things in assembly language programming. The fastest and easiest way to implement a counter is to use a CPU register; however, when one runs out of registers, counters must be kept in memory. A typical error occurs during the incrementing or decrementing of a 16-bit memory counter with an 8-bit memory increment or decrement instruction. Instead of changing the full 16 bits, the programmer changes only 8 bits. You should load the 16-bit counter into a register, increment or decrement it, and store it back into memory. Make sure the store operation does not change the flags if a conditional branch based on the increment or decrement follows.

7. **Modifying a counter in a loop:** Another common problem with counters is reinitializing the counter inside the loop. Example 8-8 shows this problem. To fix it, move the label `loop:` below the counter initialization code at *line 4*.

Example 8-8 Improper Counter Initialization

```
1. COUNT: EQU  100
2. ; Initialize the counter in memory
3. loop:
4.        movb #COUNT,Counter
5. ; . . .
6.        dec  Counter
7.        bne  loop   ; Whoops!
8. ; . . .
9. Counter: DS.B 1  ; 8-bit counter
```

Finding Register Problems

Register problems are difficult to find, especially because data in registers are often variable. It is critical that you follow the analytical debugging model shown in Figure 8-1, closely monitoring what you expect the register contents to be and what the program is actually doing. When you find a deviation, you have found the area of the problem; now you must look further to find the cause.

Condition Code Register Problems

Almost all instructions modify the condition code register, and you must know which instructions modify which bits. Problems with the condition code register lead to problems in the program flow. These problems, too, are hard to find, especially when variable data are used. Two of the problems you are likely to encounter are the following:

1. **Modifying condition code contents before conditional branch instructions:** Be aware of all instructions that modify the contents of the condition code register. Make sure there are no instructions that change the condition code register between the time it is set and the time the conditional branch is executed. See Example 8-9.

Example 8-9 Load and Store Instructions Modify the Code Condition Register

Analyze the following code for a condition code register problem.

```
1. COUNT: EQU  8  ; Loop counter
2. ; . . .
3.         ldab #COUNT ; Initialize loop counter
4. ; . . .
5. loop:
6. ; Here is the code for whatever has to be
7. ; done in a loop. At the end of the loop, we
8. ; decrement the loop counter and branch back
9. ; if the counter hasn't been decremented to zero.
10. ; . . .
11.         decb       ; Decrement B register
12.                    ; and branch back if not zero
13.         ldaa #$64 ; But first load A with some data
14.         bne  loop
```

Solution

The programmer follows the `decb` instruction with an instruction to load A with 0x64. Depending on your microcontroller hardware, the `ldaa` may modify the condition code register bits so that the zero bit is zero and the `bne loop` instruction will always be taken. The program will never exit from the loop.

2. **Using the wrong conditional branch instruction:** There are different instructions for signed and unsigned numbers. Conditional branches with the words "greater" or "less" are used for signed numbers and those with the words "higher" or "lower" for unsigned numbers. Table 8-1 shows how three different microcontrollers use different branch instructions for signed and unsigned data. Also see Example 8-10 and Table 8-2.

Table 8-1 Signed and Unsigned Branch Instructions

Processor	Signed	Unsigned
Freescale HCS12	Greater than – BGT	Higher than – BHI, BCC
	Less than – BLT	Lower than – BLO, BCS
	Greater than or equal – BGE	Higher than or same – BHS
	Less than or equal – BLE	Lower than or same – BLS
	Minus – BMI	
	Plus – BPL	
	Overflow – BVS	Overflow – BCS
	No overflow – BVC	No overflow – BCC
	Equal – BEQ	Equal – BEQ
	Not equal – BNE	Not equal – BNE
TI MSP430	Greater than or equal – JGE	Lower – JNC, JLO
	Less than – JL	Higher or same – JHS, JC
	Equal – JEQ, JZ	Overflow – JC
	Not equal – JNE, JNZ	No overflow – JNC
	Negative – JN	
Atmel ATiny261	Greater than or equal – BRGE	Same or higher – BRSH
	Less than – BRLT	Lower – BRLO
	Minus – BRMI	Overflow – BRCS
	Plus – BRPL	No overflow – BRCC
	Overflow – BRVS	
	No overflow – BRVC	

Example 8-10 Using the Wrong Conditional Branch

A programmer intends to compare two 8-bit unsigned data values, the first in the A register and the second in a variable memory location `Data`. The program design requires a branch to `GREATER:` if the value in A is greater than the value in `Data`. The following program segment shows the code that was written:

```
. . .
cmpa  Data
bgt  GREATER
. . .
```

What is wrong with this code, and what data values would you suggest show that it works incorrectly?

Solution

Of course it all depends on the instruction set of your microcontroller; in general, however, there will be an instruction, such as the `bgt` instruction (branch greater than) that is used for signed, two's-complement data, not for unsigned data. The correct conditional branch instruction to use is the `bhi` (branch higher than) instruction. This is a particularly hard bug to find because sometimes it will work properly and sometimes not, as shown in Table 8-2.

Table 8-2 Data Illustrating That bgt is the Incorrect Instruction

A	Data	Is A Greater than Data?	Is A Higher than Data?	Explanation
0x55	0x05	Yes	Yes	0x55 is both greater than and higher than 0x05.
0x05	0x55	No	No	0x05 is not greater than or higher than 0x55.
0x7F	0x80	Yes	No	0x7F (+127) is greater than 0x80 (–128) (signed) but 0x7F (+127) is not higher than 0x80 (+128) (unsigned). `BHI` is the correct instruction for unsigned data.
0x80	0x7F	No	Yes	0x80 (–128) is not greater than 0x7F (+127) (signed) but 0x80 (+128) is higher than 0x7F (+127) (unsigned). `BHI` is the correct instruction for unsigned data.

Finding Condition Code Register Problems

Condition code register problems are similar to the register problems described earlier and are found in a similar fashion. You must carefully watch the condition code bits, as they are modified by your program while different sets of test data are being used.

The problem shown in Example 8-10 is very difficult to find because it depends on the data being compared. As Table 8-2 shows, for some data values the `bgt` gives the correct result and for others it does not. This is why you must carefully design your test values to test as many data cases as possible.

Test Data Strategies

Generating test data to test exhaustively your code and hardware is a complete topic itself, and we can only give some guidelines here. It is virtually impossible to test rigorously all possible combinations of inputs, outputs, and processing paths in our programs. Try to be as thorough as possible, but apply some strategy and judgment to make your testing program reasonable. Here are some suggestions for developing testing conditions for your programs.

- Choose data values or conditions that are representative of what you expect to test normal operation.
- Choose data values or conditions that are at the boundaries of what you expect. For example, if your code is dealing with signed data, test the most negative and most positive numbers. For unsigned data, be sure to test zero and the most positive numbers.
- Choose conditions that are outside anything you would responsibly expect. This is very important. Your program has to work with good values and bad values, too.
- If your program has user input, make sure to test all possible inputs, including more than one key pressed at a time and keys pressed rapidly.

8.7 Other Debugging Techniques

Eliminating Code

Sometime the debugging strategies suggested thus far do not lead to a problem solution. There may be interacting problems or code that obscures the problem. A technique that can help in these cases is to start eliminating sections of code to simplify the problem. Code to be eliminated can be commented out to allow you to restore it later. This is useful at times when the compiler is giving you an error and you cannot decide where the error is.

Hardware Additions

When doing your hardware design, reserve an I/O bit or two for testing the software. This bit can be connected to an LED or to a pin to which you would connect an oscilloscope. When you test the hardware and software of your system, an output to this pin can test a variety of software events such as entering or leaving subroutines and interrupt service routines. By setting the bit when a section of code is entered and resetting it upon leaving, you can measure the execution time.

Another hardware design consideration that will greatly help in system debugging is to add a liberal amount of test points to which you can connect an oscilloscope to observe signals. CPU timing signals such as read, write, and bus clock allow you to check other system timing relative to these fundamental signals. Be sure to include multiple places to ground the oscilloscope.

A common addition to many embedded systems is a "heartbeat" LED. When the code is operating normally, it flashes the LED at some rate, say 2 Hz. You can easily look at the heartbeat LED to see if the code is hung up or delayed.

If your microcontroller has I/O capabilities that your application is not using, such as an unused serial I/O port, consider connecting the I/O pins out to your printed circuit board. This can give you a spare debug port. If you have room on the PC board, it is very useful to leave space for interface chips that can be added during attempts to solve difficult debug problems. The chips can be removed or not installed for the delivered product.

If your microcontroller supports an in-circuit emulator or on-chip debugger, as described shortly, install the debug connector on the board. These are usually only one or two pins, or at most five for JTAG debuggers.

Using an Oscilloscope or Logic Analyzer

Because embedded systems usually connect to external devices, an oscilloscope is a critical piece of test equipment; you can use it to verify that your system is operating properly and to look for bugs. The oscilloscope measures and displays voltage waveforms as a function of time. The various uses for an oscilloscope in your testing and debugging phase include the following.

- Measuring the execution time of a function. You can measure the time spent in the module by putting some test code at the beginning of the function that sets an output bit on a port and resetting it at the end of the function.

- Checking that a pulse-width modulation waveform is being generated correctly.
- Measuring the time between assertion of an interrupt signal and entrance into the interrupt service routine. (With a dual-channel oscilloscope, you trigger the sweep with the interrupt signal and then, in the interrupt service routine, set and reset (toggle) an output bit on a port. See Chapter 10 for more information on interrupts.)
- If your system fails to start, the first things to check with your oscilloscope are the power supply to the chip and the clock oscillator.
- In a system with external address, data, and control buses, you can check that control signals are operating properly and that the data and address buses do not have any faults such as being stuck on zero (grounded) or stuck on one (connected to the positive supply).

Another useful tool is a logic analyzer. While most oscilloscopes have only two, or at most four, channels, a logic analyzer can display many more channels of logic signals. Be aware, though, that the logic analyzer is not like the oscilloscope because it doesn't show you the actual voltage levels.

In-Circuit Emulator or Background Debugger

The early microprocessors, such as the Intel 8080, and 8085, the Zilog Z80, and the Motorola 6800, were much smaller than those of today, with far fewer I/O pins. In addition, they were much less powerful, with few if any integrated I/O capabilities, and they operated with far lower frequency clocks. Development systems were manufactured that allowed the system developer to replace the microprocessor CPU with a special plug and cable that connected to the target system under development. These development systems, called *in-circuit emulators*, enabled the developer to run the target system to exercise all its hardware and software while using debugging tools like tracing, breakpoints, and inspecting and modifying memory. The development of these systems was a great leap forward and gave system designers a wonderful tool to test and debug their software and hardware.

Now fast forward to the microcontroller world of today. It is no longer feasible to use in-circuit emulators. The chips have far too many pins, have far too many internal devices that lack direct connection to I/O pins, and operate at much higher clock frequencies than before. A new debugging strategy has been developed. The new scheme is called *Background Debugging* (Freescale), *On-Chip Debugging* (Atmel, Microchip), *Embedded ICE®* (Cirrus Logic), or *JTAG Debug* (Maxim, TI, and others).

Figure 8-9 shows a Freescale background debug module (BDM) pod, which is connected to the printed circuit board (PCB) of the target microcontroller. Other manufacturers provide comparable debugging capability. Such devices allow access to the internal operation of the target system and do not interfere with the application hardware or software. You can read or write to target system memory locations without stopping the running application program. Background debugging systems use a separate serial I/O interface and no user memory; most use a separate serial I/O interface with only two or three pins, which means you can easily add the background debug capability to your product without much cost. The chips using the JTAG (Joint Test Advisory Group) interface require a 5-pin interface, but JTAG devices can do sophisticated on-chip testing called Boundary Scan testing in addition to background debugging.

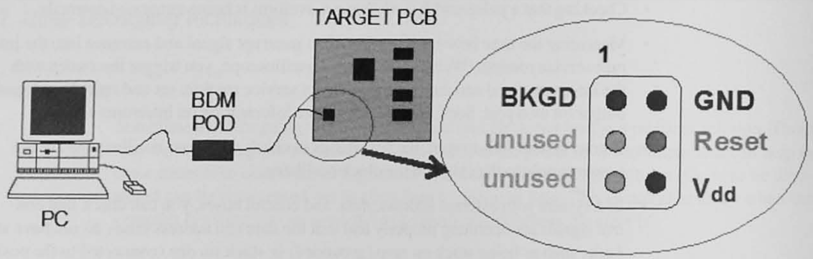


Figure 8-9 Background debug module interface.

The background or on-chip debugging systems include many of the following features:

- Reading and writing target memory
- Reading and writing CPU registers
- Starting and stopping application programs
- Tracing application program instructions
- On-chip, in-circuit emulation that operates like an external emulator and may have the following features:
  Hardware breakpoints that can trigger on selected address and data bus activity
  Real-time instruction trace buffer
  Instruction tagging to allow breakpoints to be set at an instruction

Although the on-chip debugging systems are primarily intended for system development and debugging, they are useful for other applications. They can be used to load or reload an application program into a target system after the product has been completely assembled. They can be used also to calibrate finished systems or perform field upgrades of operation software. In a data logging application, the background debugging system can retrieve logged data, thus making it unnecessary to add this function to the application software.

The PC shown in Figure 8-9 is connected to a BDM pod through a USB or serial port. Software that runs on the PC allows you to set breakpoints, trace, display memory, and undertake the other debugging tasks we described in Section 8.4.

8.8 Conclusion and Chapter Summary Points

In this chapter we have offered some debugging strategies to help you find those annoying bugs in your programs. Key elements of those strategies are the following.

- Analyze the program; do not try to synthesize new code to fix it.
- Develop a model of how you think the program should run, including data transformation, register use, and program flow.
- Compare the model of the program with what the program actually does.
- Find out where the program differs from the model.
- Use tracing and breakpoints to check the program flow.
- Use register and memory display to check the data.
- Carefully construct test cases that test the program's behavior as thoroughly as possible.
- Do not forget to test for unexpected inputs.
- The combination of an oscilloscope and test points included in the hardware allows us to debug both hardware and software problems.
- Modern microcontrollers may have on-chip debugging features like in-circuit emulation and real-time debugging.

8.9 Bibliography and Further Reading

Airaudi, T. J., *Using Background Debug Mode for the M68HC12 Family*. Freescale Semiconductor, Inc., Application Note, AN2104/D, 2001.
Cady, F. M., *Software and Hardware Engineering, Assembly and C Programming for the Freescale HCS12 Microcontroller*. Oxford University Press, New York, 2008.
Ganssle, J. G., *The Art of Programming Embedded Systems*. Academic Press, San Diego, CA, 1992.
Ganssle, J. G., *A Guide to Commenting*. Ganssle Group, Baltimore, February 2006. http://www.ganssle.com/commenting.htm
Sibigtroth, J., *Friendly Flash and the End of the ICE Age, Flash and Debug Support Continue to Evolve*. http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=01624684490DEC1CDB
Sibigtroth, J., *Serial Monitor for the MC9S08GB/GT*. Freescale Semiconductor, Inc., Application Note, AN2140/D, 2004.

8.10 Problems

Stimulate

8.1 Explain why 0x20 is exclusive-ORed with the input character in Example 8-4 to change the case. [c, g]

Challenge

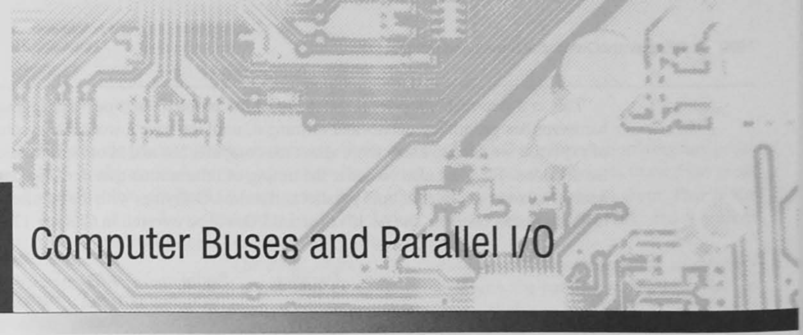
8.2 The sample program in Example 8-4 has a bug in main() so that it does not transfer the string 100% correctly. If you have a C program debugger, find the problem and state how would you fix it. [b]

8.3 Assume that main() in Example 8-4 is a test jig to test the `change_case` function. Comment on the thoroughness of the testing. What would you do to make the testing better and more rigorous? [b]

8.4 The `change_case` function in Example 8-4 does not work properly for all alphabetic characters because it has a bug in it. If you have a C program debugger, find the problem and state how you would fix it. [b]

Reflect on Learning

8.5 Make a list of debugging tricks and ideas that you did not know before reading this chapter.



9 Computer Buses and Parallel I/O

Objectives

Computers doing real jobs must input and output information. Two ways to do I/O are many bits at a time (in parallel) and one bit at a time (in serial). In this chapter we will explore parallel bus architectures and explain how to design the interfaces between external devices and the CPU. We will discuss the differences between memory-mapped and separate I/O and learn to solve I/O synchronization problems that arise between a fast processor and a slow I/O device. Some advanced bus ideas are covered.

9.1 Introduction

Parallel and serial I/O devices require an *interface* between the device and the bus.

Figure 9-1 shows a computer system with CPU, memory, I/O, and the interconnecting computer buses; this is the *von Neumann* architecture, where the memory and I/O interfaces share the address, data, and control buses. An alternative architecture, known as the *Harvard architecture*, has separate data, address and control buses for the data and program memory. In this architecture (Figure 9-2), the CPU can be accessing program instructions and data memory simultaneously. In earlier chapters we have emphasized the CPU, its resources, and how to program it to do a particular task. We now look to the design of the rest of the system hardware.

Many computer applications involve the transfer of information, either in parallel or in serial, in or out of the CPU. Both parallel and serial I/O require a hardware interface between the source or destination of the information and the CPU. Let us approach this topic by using the top-down design principles covered in Chapter 3. The preliminary problem specification is the following:

Design the hardware interface that will use a computer bus to transfer information from multiple sources to the CPU, and from the CPU to multiple destinations.

This is a reasonable top-level statement of the problem, and our task is to go through the hardware design, adding details and refining it, until we have a workable system. At our level of expertise we need to know more about the computer bus and about information sources and destinations. We must also consider the timing of information transfer. The design goal is an interface that is suitable for both parallel and serial I/O devices with the parallel case described in detail in this chapter. The details of serial I/O will be covered in Chapter 12.

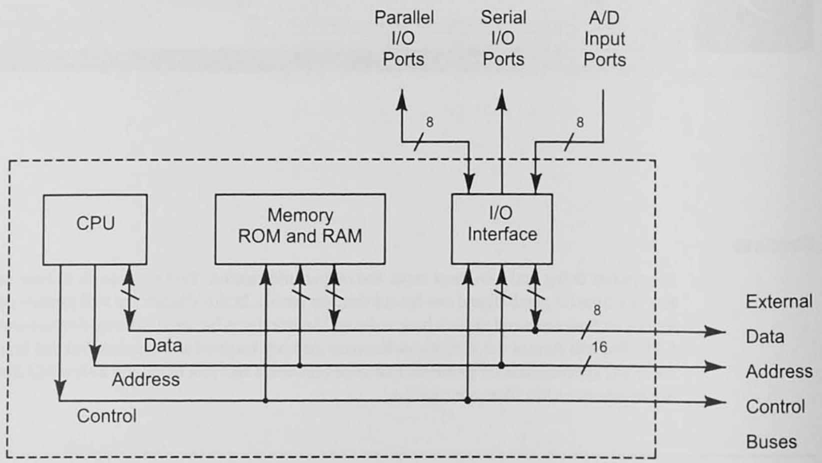


Figure 9-1 Von Neumann computer architecture.

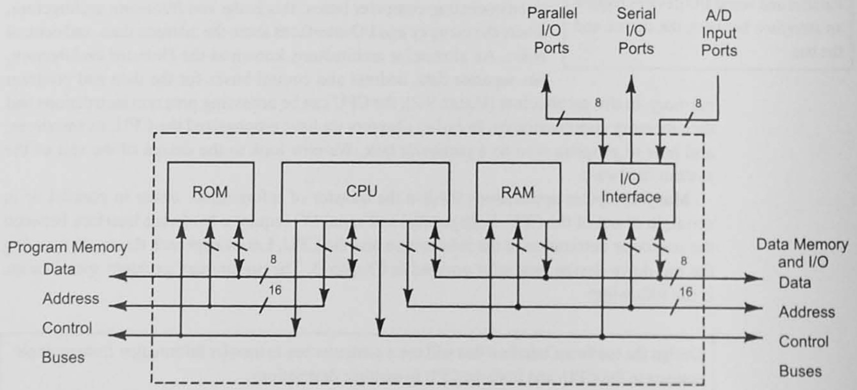


Figure 9-2 Harvard computer architecture.

9.2 The Computer Bus

What is the proper place to start the top-down design of the I/O interface? In solving any problem, begin work where you have some knowledge or expertise and progress toward the areas about which you need to learn more, filling in the details as you come to them. This is the essence of top-down design for hardware. Let us start our design at the CPU and work out toward the I/O devices.

A *bus* is a parallel, bidirectional, binary information pathway with multiple sources and destinations.

The *data*, *address*, and *control* buses connect the CPU to memory and to I/O devices. In circuit diagrams, the parallel wires of the bus are generally reduced to a single line as shown in Figure 9-3a. The number of bits in each bus depends on the design of the system, and more data bits means that the system can transfer data at a higher rate; having more address bits allows more memory to be addressed.

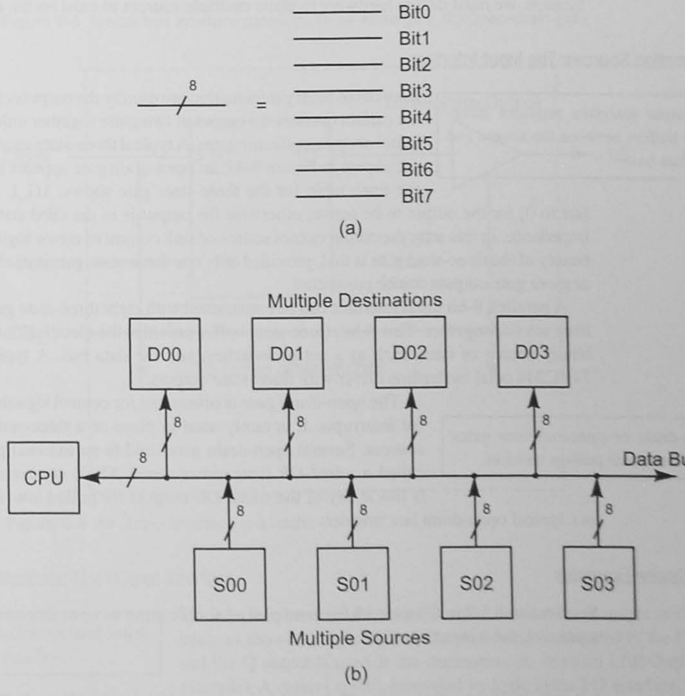


Figure 9-3 The computer bus. (a) Schematic notation. (b) Bus line with multiple sources and destinations.

The data bus is bidirectional and transfers information (memory data and instructions, I/O data) to and from the CPU. The address bus may be bidirectional (with more than one source of information); however, because the CPU is the only source of addresses, it is most often unidirectional. The control bus carries all other signals required to control the operation of the system.

There are several levels of bus design. The *component-level bus* is defined by the signals on the microprocessor chip, such as READ_L and WRITE_L (we will see how these signals are used shortly). Component-level signals will be different for different manufacturers and are used when designing single-board computers or embedded application systems. A *system-level* bus is one for which more generic signals, like MEMRD_L and MEMWR_L, are defined. A system-level bus is often designed for use as a *backplane* into which printed circuit boards are plugged. An example of a system-level bus is the PCI bus defined for desktop computers. A third type of bus is the *intersystem* bus, which allows different systems to be connected. A good example is the IEEE 488 (GPIB) instrumentation bus. In this chapter we will be concerned with the component-level bus.

Let us now consider the data bus in detail. Each line of the bus may have multiple sources and destinations for the information, as shown in Figure 9-3b. According to our design specification, we must design hardware to allow multiple sources to exist on the bus.

Information Sources: The Input Interface

An *input interface* provides three-state buffers between the source and the data bus.

Sources of binary information are usually the outputs of gates.[1] However, we cannot connect the output of two gates together unless they are *three-state*[2] or *open-collector* gates. A typical three-state gate and its truth table are shown in Figure 9-4a; an open-drain gate appears in Figure 9-4b. As the truth table for the three-state gate shows, 1G_L must be asserted (set to 0) for the output to be active; otherwise the output is in the third state, known as high impedance. In this state the output cannot source or sink current to create logic one or zero. The beauty of the three-state gate is that, provided only one three-state gate is enabled at a time, two or more gate outputs can be connected.

A parallel, 8-bit input interface can be constructed with eight three-state gates whose enable lines are tied together. This 8-bit, three-state buffer provides the electrical interface between a binary source of data, such as a set of switches, and the data bus. A typical device is the 74HC244 octal buffer/line driver with three-state outputs.

Open-drain or *open-collector* gates require external pull-up resistors.

The *open-drain gate* is often used for control signals such as requests for interrupts. It is rarely used in place of a three-state buffer for data sources. Several open-drain gates tied to an external pull-up resistor is called a *wired-OR* (sometimes wired-AND) connection. The bus line is low if any of the wired-OR outputs are pulled low. An MC74LCX06 is a typical open-drain hex inverter.

Input Device Examples

See Section 15-2 in Chapter 15 for examples of simple input devices that can be connected to a parallel, 8-bit input interface.

[1] It could also be some other circuit, such as a switch, that provides logic levels for binary 1s and 0s.
[2] These gates are also know as Tristate gates; the term is a trademark of the National Semiconductor Company. NSC's invention revolutionized the design of computer buses.

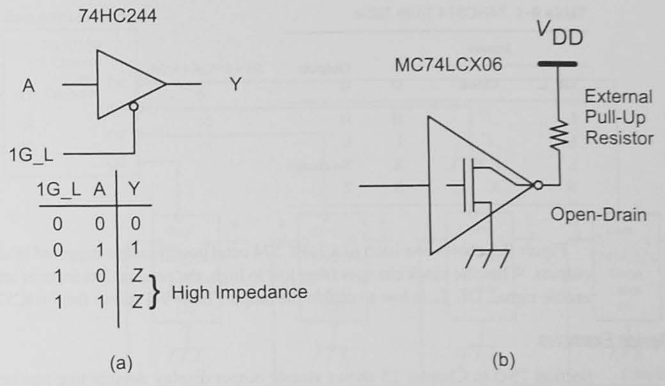


Figure 9-4 Typical bus interface gates. (a) Three-state gate. (b) Open-drain gate.

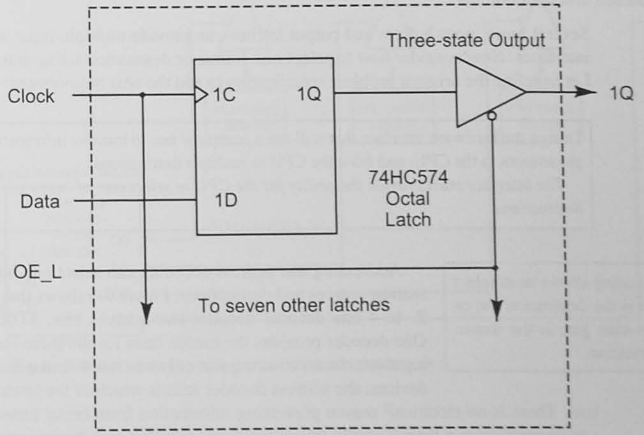


Figure 9-5 An output interface is a latch.

Information Destinations: The Output Interface

The *output interface* must latch information from the data bus.

The interface between the data bus and a destination or output device is a latch, as shown in Figure 9-5. A data bus line is connected to the D input, and the Q output is used at the destination (to drive an LED display, for example). A control signal, generated by logic in the I/O interface, latches the data from the bus. The logic for this clock, and the enable on the input interface's three-state buffers, is generated in part by address decoding and in part by timing signals, as we will see shortly.

Table 9-1 74HC574 Truth Table

Inputs			Output:
OE_L	Clock	D	Q
L	↑	H	H
L	↑	L	L
L	L, H, ↓	X	No change
H	X	X	Z

Figure 9-5 shows one latch of a 74HC574 octal positive edge-triggered latch with three-state outputs. When the clock changes from low to high, the present data input is latched. The output enable signal, OE_L, is low to enable the output. Table 9-1 shows the 74HC574 truth table.

Output Device Examples

Section 15-3 in Chapter 15 shows simple output display devices that can be used with these output interfaces.

Multiple Sources and Destinations

Several basic input buffers and output latches can provide multiple input and output device interfaces. Now consider how to select one source or destination for an information transfer. Let us refine the original problem specification to add the next requirement:

> Design the hardware interface that will use a computer bus to transfer information from multiple sources to the CPU, and from the CPU to multiple destinations.
> *The interface must provide the ability for the CPU to select one-of-many sources or destinations.*

> *Address decoding* allows us to select which latch is the destination for, or which three-state gate is the source of, the information.

Addressing and address decoding can select one-out-of-many information sources and destinations. Figure 9-6 shows that a 74HC139 dual 2- to 4-line decoder decodes two address bits, ADR_1 and ADR_0. One decoder provides the enable lines for the three-state buffers in the input interfaces, ensuring that only one is active at a time. For the output devices, the address decoder selects which of the latches is the destination. There is no electrical[3] reason preventing information from being transfer to more than one destination at a time. Usually this is not done, however, and the address decoder for the destination selects only one destination.

In the I/O interface shown in Figure 9-6, address bits A1 and A0 select which of the four input or output devices are to be used. Two control signals, WRITE_L and READ_L, are shown also. The CPU generates these to provide timing information for the data transfer in and out of the CPU. When READ_L is asserted, the addressed three-state gate is enabled to place the source data on the data bus. Similarly, when WRITE_L is asserted, the CPU output data on the data bus is latched into the output latches.

[3] Except device loading and fan-out.

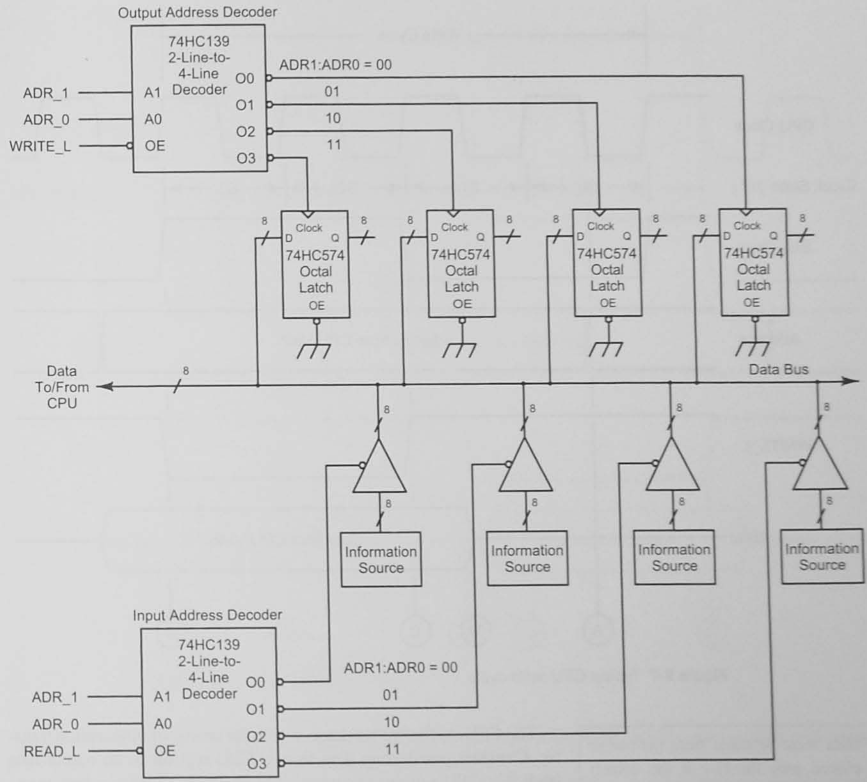


Figure 9-6 Address decoding for sources and destinations.

Timing Signals

We must add a timing requirement to the design specification:

> Design the hardware interface that will use a computer bus to transfer information from multiple sources to the CPU, and from the CPU to multiple destinations.
> The interface must provide the ability for the CPU to select one-of-many sources or destinations.
> *Provide timing and synchronization to ensure that the transfer of information occurs at the right time.*

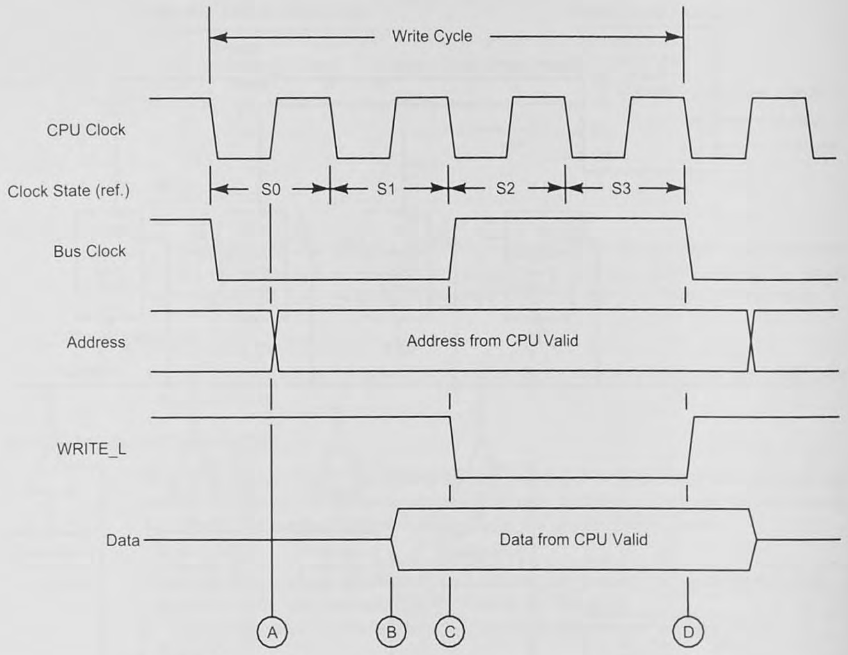


Figure 9-7 Typical CPU write cycle.

Data must be taken from the bus or placed onto the bus at the correct time. The CPU controls this timing.

The CPU, as the bus master, controls the timing of information transfer. Consider transferring data from a CPU register to an output data latch. The CPU's timing is controlled by its clock, and this output operation is called a *write cycle*. Figure 9-7 shows a typical CPU write cycle.

The CPU places the address on the address bus at point A. The data bits are supplied at point B, and the CPU asserts WRITE_L a short time later at point C. This signal creates the latch clock to latch the data at the correct time. Depending on both the type of latch and when WRITE_L is asserted, the data may be captured on the falling or rising edge.

The CPU provides *READ_L* and *WRITE_L* control signals to define when read or write actions are taking place.

Transferring information from an external source to the CPU is called a *read cycle*. The CPU reads the data from the input device, and a typical CPU read cycle is shown in Figure 9-8. Again, the CPU supplies an address at point A. The control signal READ_L is asserted at point B to signal the external device that the CPU is ready to take the data from the data bus, which it does at point D. READ_L is used with the decoded address signal to enable the three-state buffers at the correct time for the correct device. An important point to mention now is that the CPU reads the data bus at point D regardless of whether the input device has it ready. If it is not ready, some form of I/O

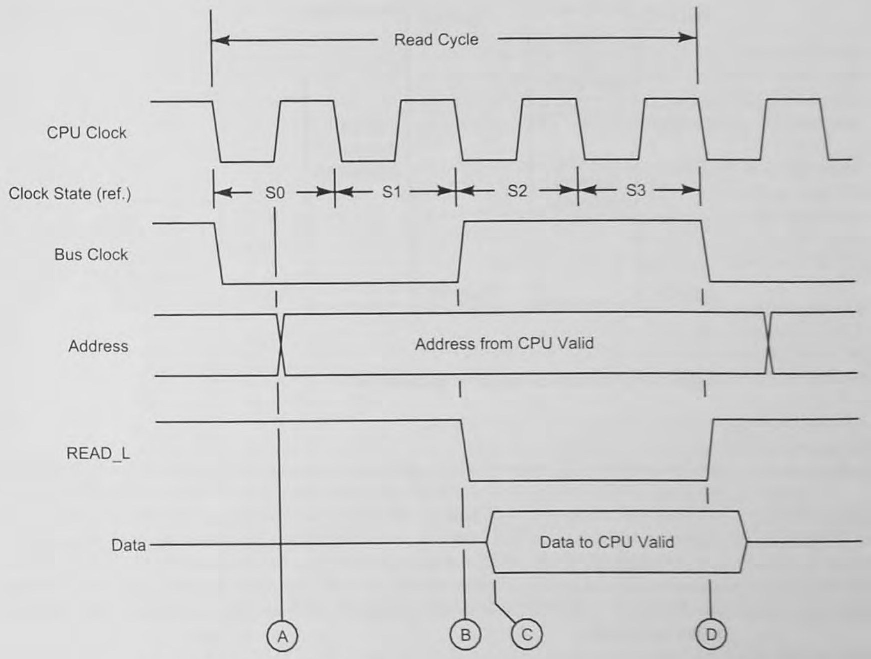


Figure 9-8 Typical CPU read cycle.

synchronization is needed. We will assume, for the moment, that the data is there and delay discussion on this problem until Section 9.7.

Figure 9-6 shows the addition of READ_L and WRITE_L control signals to the input and output interfaces. In each case, READ_L and WRITE_L control the output enable OE_L input on the address decoder. Thus, the three-state enables and the latch clock signals are not asserted until the correct address is on the address bus AND the correct time in the write or read cycle has arrived. Notice that the address for both the input and output devices can be the same. This is possible because the CPU does not read and write simultaneously, and the READ_L and WRITE_L control signals ensure that only one device is active.

9.3 I/O Addressing

I/O addresses may be either *memory mapped* or *separate* from the memory space.

The address bus shown in Figure 9-1 is decoded by both memory and I/O to select a particular memory location or I/O device. When an instruction retrieves data from memory, the address from which the data is to come is placed on the address bus by the CPU. Although we have not

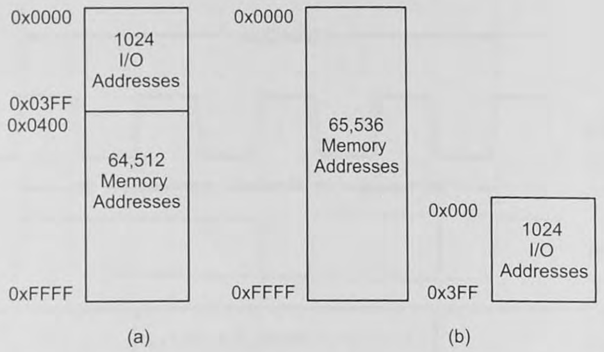


Figure 9-9 (a) Memory-mapped I/O. (b) Separate I/O.

yet discussed how the memory works, address decoders, much like those shown in Figure 9-6, are used in the memory circuits. The question that arises now is, "If the same address bus is used for both memory and I/O, how should the hardware be designed to differentiate between memory reads and writes and I/O reads and writes?" There are two solutions to this problem: *memory-mapped I/O* and *isolated* or *separate I/O*. These two choices affect how our programs access the I/O devices. In memory-mapped I/O, any instruction that reads or writes memory can also read or write I/O. For isolated I/O, the CPU designers must include separate input and output instructions.

Memory-Mapped I/O

Memory-mapped I/O may require that the full address bus be decoded, but any memory reference instruction can access I/O data.

The simplest I/O addressing scheme[4] is called *memory-mapped I/O*. Address decoders work equally well decoding memory or I/O addresses, and the control signals for timing the memory data transfers are available. Thus, the CPU designer does not have to add any special capabilities to the CPU. The entire address space is used by both memory and I/O addresses. This is shown in Figure 9-9a.

This design was popular in early minicomputers, such as the Digital Equipment Corporation PDP-8, and many microcontrollers use it today. It offers the advantage of a simpler CPU hardware design and allows any memory reference instruction to access any I/O device. There are two disadvantages. First, as shown in Figure 9-9a, I/O devices reduce the amount of memory available for application programs. Second, the I/O address decoders must decode the full address bus to avoid conflict with memory addresses. This makes them more expensive than decoders operating on fewer bits.

[4] From the point of view of the CPU designer.

Isolated or Separate I/O

The second method to resolve the dual use of the address bus is called *isolated* or *separate I/O*. Here, two maps, as shown in Figure 9-9b, represent the memory and I/O spaces. Notice that the I/O map is smaller than the memory map because most systems need far fewer I/O devices than memory. Fewer bits need to be decoded, resulting in less expensive address decoders than those needed for memory-mapped I/O.

In a *separate I/O* design, cheaper address decoders may be used, but an additional control signal must be provided by the CPU.

Separate I/O requires that the CPU be enhanced with additional hardware features. The address bus is used for both memory and I/O addresses, and because of this, the CPU must generate an additional control signal. This signal, called *IO/M_L*, prevents memory and I/O from trying to place data on the bus simultaneously. This signal is high for I/O use and low for memory.

The second change to the CPU hardware design is in the instruction decoder and sequence controller. Computers with separate I/O, and consequently the IO/M_L control signal, have I/O instructions that are separate from memory reference instructions. An easy way to decide if your computer has separate or memory-mapped I/O is to look for separate input and output instructions.

Figures 9-10 and 9-11, respectively, show I/O interfaces for memory-mapped I/O and separate I/O. Compare Figures 9-10 and 9-6 and notice that they are logically the same except the ADR_OK_L signals are ANDed with READ_L and WRITE_L signals external to the address decoder. This allows a single address decoder to be used for both reading and writing.

Figure 9-11 shows the additional logic required in a separate I/O interface. The IO/M_L control signal is ANDed with READ_L and WRITE_L to create IO_READ_L and IO_WRITE_L. In some processors the CPU generates IO_READ_L and IO_WRITE_L. An alternative to the design shown in Figure 9-11 is to use IO/M_L as an enable input to the address decoder. The object is to *qualify* the three-state enable and the latch clock so that they are asserted only when I/O addresses are present.

Figure 9-6 shows separate address decoders for each of the input and output devices. This configuration may or may not exist, depending on the way the system is put together. If the

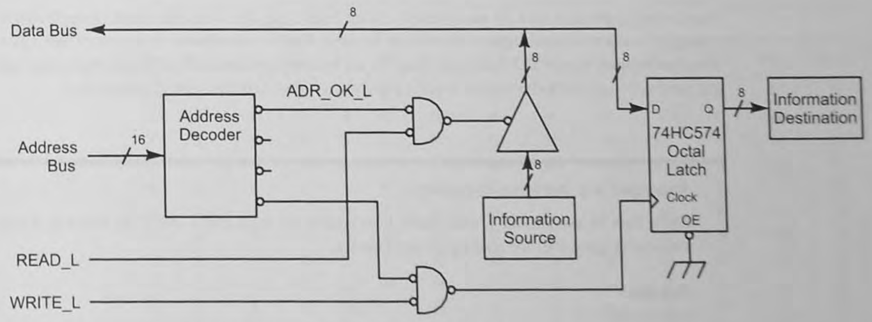


Figure 9-10 I/O interface for memory-mapped I/O.

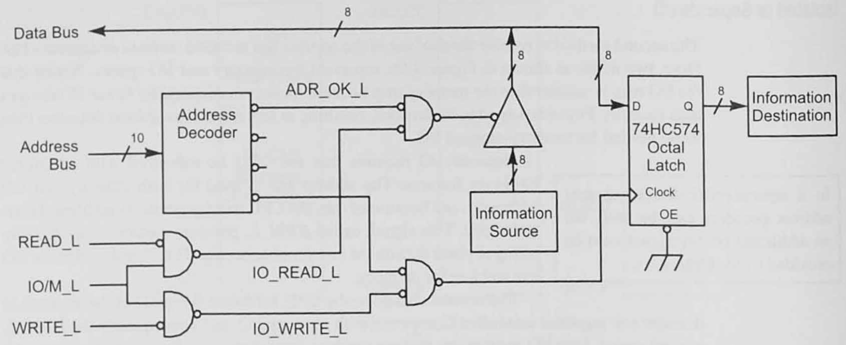


Figure 9-11 I/O interface for separate I/O.

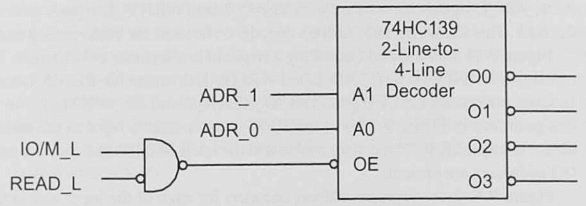


Figure 9-12 Using control signals to qualify the address decoder.

logic circuits are together on one printed circuit board, say for a microcontroller application, a single decoder with multiple outputs may be used. On the other hand, in a system like a personal computer, where I/O interfaces may be on separate printed circuit boards that plug into the motherboard, each device must use a separate address decoder. See Example 9-1.

Example 9-1 Address Decoding

Show how to use READ_L and IO/M_L to enable the output of a 74LS139 2-line-to-4-line decoder (Figure 9-6) for reading an input device.

Solution

See Figure 9-12.

Address Decoding

We have seen how address decoding can select a particular device. The example of Figure 9-6 shows two address bits selecting one of four devices, but more bits are needed to decode addresses for real I/O devices. A disadvantage of the memory-mapped scheme is that more address bits must be decoded to select uniquely either memory or I/O. This is, of course, a more expensive address decoder. In practical systems, the hardware designer chooses to decode only as many address bits as the system requires.

Full Address Decoding

In a system with many I/O devices, the designer must decode enough bits to select uniquely each device. At the upper limit, allowing the maximum number of I/O devices is *full address decoding*.

A typical address decoder is the 74HC138 (1-of-8 decoder/demultiplexer) shown in Figure 9-13. The truth table (Table 9-2) shows that the outputs are asserted low when the enable input E1 is high and both E2_L and E3_L are low. Address bits can be used as enable inputs, and decoders can be cascaded as shown in Figure 9-14b to decode the 10-bit address 0x3E8.

Discrete logic circuits can decode addresses. Figure 9-15 shows a 10-bit decoder for address 0x3E8; it uses a 74HC30 8-input NAND, a 74HC27 triple 3-input NOR, and one gate of a 74HC04 hex inverter. The inverter could be eliminated if an active-high decoder output were

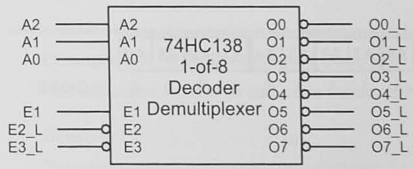


Figure 9-13 The 74HC138 1-of-8 decoder/demultiplexer.

Table 9-2 Truth Table for the 74HC138

Inputs						Outputs							
E1	**E2_L**	**E3_L**	**A2**	**A1**	**A0**	**O0_L**	**O1_L**	**O2_L**	**O3_L**	**O4_L**	**O5_L**	**O6_L**	**O7_L**
L	X	X	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	H	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	L	H	H	H	H	H	L	H	H	H	H
H	L	L	H	L	L	H	H	H	H	L	H	H	H
H	L	L	H	L	H	H	H	H	H	H	L	H	H
H	L	L	H	H	L	H	H	H	H	H	H	L	H
H	L	L	H	H	H	H	H	H	H	H	H	H	L

allowed. Notice that there are fewer chips in Figure 9-14b than in Figure 9-15 if the inverter is needed for an active-low decoder output. Notice also that the discrete decoder provides decoding for only one address where the 74HC138 decoders provide other addresses. See Examples 9-2 through 9-4 and Table 9-3, which follows Example 9-3. Your design can also use a combination of discrete logic and decoders.

Example 9-2 Design a Full Address Decoder to Decode the 10-Bit Address 0x3E8

Solution

Starting with the most significant bit, assign address bits to decoder inputs. For the most significant decoder, select the appropriate output to serve as an enable input for the next decoder in the cascade. Apply the remaining address bits to decoder data and enable inputs and then choose the correct output for the address required. See Figure 9-14 for a possible solution.

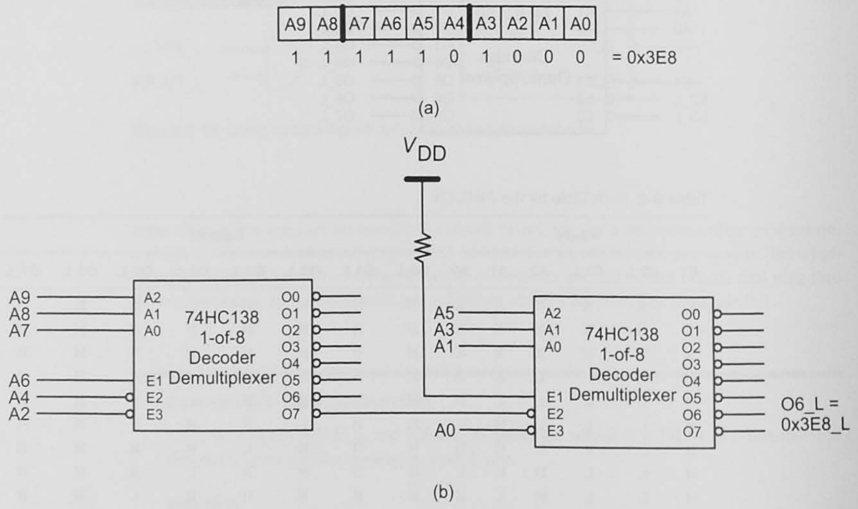


Figure 9-14 Full 10-bit address decoder. (a) 0x3E8. (b) Cascaded 74HC138 decoders.

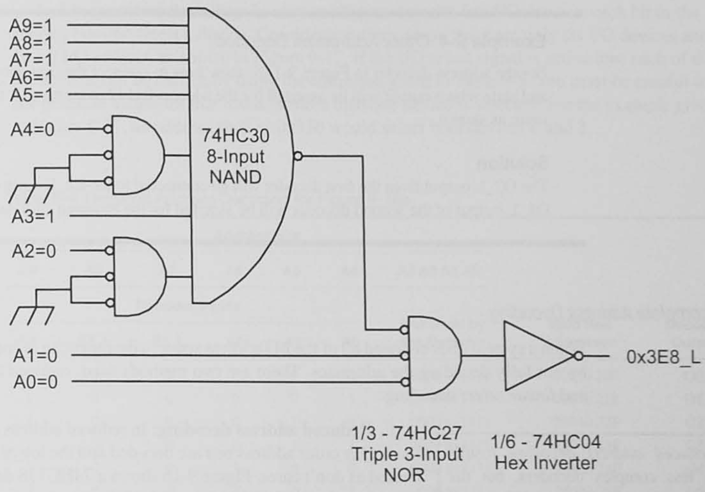


Figure 9-15 Discrete logic decoder.

Example 9-3 Multiple Addresses Decoded

Find the address decoded for each of the outputs of the second 74LS138 decoder in Figure 9-14b.

Solution

The address bits to the decoder are as follows:

Fixed inputs = A9 = 1, A8 = 1, A7 = 1, A6 = 1, A4 = 0, A2 = 0, A0 = 0

Variable inputs = A5, A3, A1

See Table 9-3.

Table 9-3 74HC138 Decoded Addresses

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	Address	Decoder Output
1	1	1	1	0	0	0	0	0	0	0x3C0	O0_L
1	1	1	1	0	0	0	0	1	0	0x3C2	O1_L
1	1	1	1	0	0	1	0	0	0	0x3C8	O2_L
1	1	1	1	0	0	1	0	1	0	0x3CA	O3_L
1	1	1	1	1	0	0	0	0	0	0x3E0	O4_L
1	1	1	1	1	0	0	0	1	0	0x3E2	O5_L
1	1	1	1	1	0	1	0	0	0	0x3E8	O6_L
1	1	1	1	1	0	1	0	1	0	0x3EA	O7_L

Example 9-4 Other Addresses Decoded

For the address decoder in Figure 9-14b, show how to connect the first decoder to the second and state which output will be asserted for the address 0x142. Assume that the address bits are input as shown.

Solution

The O2_L output from the first decoder will be connected to the E2_L input of the second. The O1_L output of the second decoder will be asserted for the required address.

Incomplete Address Decoding

When a system does not need all of the I/O address space, a designer can reduce hardware costs by not fully decoding the addresses. There are two methods used, *reduced* address decoding and *linear select* decoding.

Reduced address decoding results in less complex decoders, but the decoded signal is asserted for more than one address.

1. **Reduced address decoding:** In reduced address decoding, the higher order address bits are decoded and the lower order bits are treated as don't cares. Figure 9-16 shows a 74HC138 decoding address bits A9–A4 and a 74HC30 8-input NAND gate decoding bits A9–A2.

Each of the decoder output lines in Figure 9-16a responds to the addresses shown in Table 9-4. By not decoding the lower four bits of the address, each decoder output line is asserted for 16 I/O addresses.

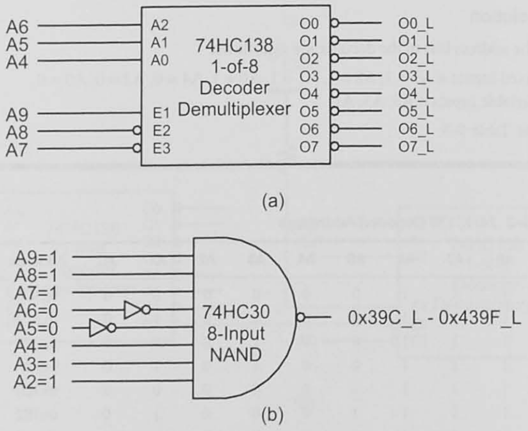


Figure 9-16 Reduced I/O address decoding. (a) 74HC138 decoder. (b) 74HC30 NAND gate decoder.

2. **Linear select decoding:** In very small systems with few I/O devices, each bit in the address bus can select a device. Consider a system where there are only six I/O devices and a 10-bit I/O address, as shown in Figure 9-17. If the I/O select signal is active low, each of the six devices can be chosen by using the addresses given in Table 9-5. You must be careful to not generate addresses that result in more than one device is selected. For the example given in Figure 9-17, an address such as 0x330 would select both devices 2 and 3.

Table 9-4 Reduced Address Decoding for Figure 9-16a

Address Bits									
A9	A8	A7	A6	A5	A4	A3 A2 A1 A0			
Decoder Inputs									
E3	E2_L	E1_L	A2	A1	A0	Not Used by the Decoder	Valid Hex Addresses	Decoder Output	
1	0	0	0	0	0	0000 to 1111	200 to 20F	O0_L	
1	0	0	0	0	1	0000 to 1111	210 to 21F	O1_L	
1	0	0	0	1	0	0000 to 1111	220 to 22F	O2_L	
1	0	0	0	1	1	0000 to 1111	230 to 23F	O3_L	
1	0	0	1	0	0	0000 to 1111	240 to 24F	O4_L	
1	0	0	1	0	1	0000 to 1111	250 to 25F	O5_L	
1	0	0	1	1	0	0000 to 1111	260 to 26F	O6_L	
1	0	0	1	1	1	0000 to 1111	270 to 27F	O7_L	

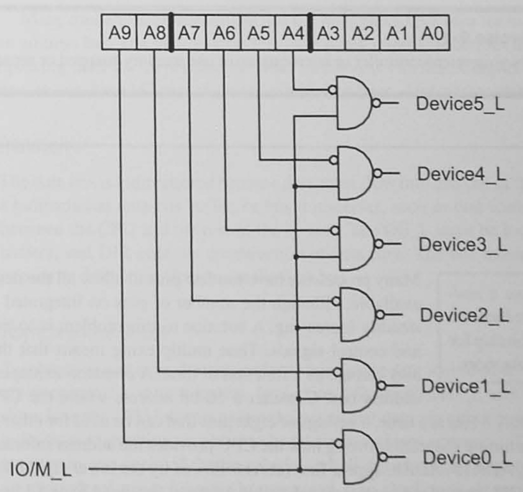


Figure 9-17 Linear select addressing.

Table 9-5 Linear Select Addressing

	Address Bits										
	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	I/O Address
Device 0	0	1	1	1	1	1	X	X	X	X	1F0 – 1FF
Device 1	1	0	1	1	1	1	X	X	X	X	2F0 – 2FF
Device 2	1	1	0	1	1	1	X	X	X	X	3F0 – 3FF
Device 3	1	1	1	0	1	1	X	X	X	X	3B0 – 3BF
Device 4	1	1	1	1	0	1	X	X	X	X	3D0 – 3DF
Device 5	1	1	1	1	1	0	X	X	X	X	3E0 – 3EF

Harvard Architecture Addressing

In memory and I/O addressing, the Harvard architecture (Figure 9-2) is similar to the von Neumann architecture just described. Address decoding is needed to select the I/O device or the memory location, and timing signals are needed to latch the data at the correct time for input and output. The difference in the two architectures is that the Harvard architecture has two completely separate buses for data, address, and control. The data and address buses can be customized to the number of bits needed, depending on the two different memory requirements. Separate control signals are required for reading and writing data and program memory. This allows the performance to be greatly enhanced and is used often in digital signal processors.

Exercise 9-1
Does your microcontroller or microprocessor use memory-mapped or separate I/O?

9.4 More Bus Ideas

Multiplexed Bus

Some microcontrollers have a *multiplexed external bus* to reduce the number of pins needed on the chip for accessing external I/O and memory.

Many processors have too few pins to allow all the desired signals to be available, although the number of pins on integrated circuits has been steadily increasing. A solution to this problem is to time-multiplex bus and control signals. Time multiplexing means that the use of any pin may change as a function of time. A common example is a multiplexed address bus. Consider a 16-bit address where the CPU is designed to provide only 8 bits at a time, a savings of eight pins that can be used for other functions. Figure 9-18a is a timing diagram showing how the CPU provides the address information. The higher eight bits, ADR15–ADR8, appear first (at A) followed by the lower eight bits (ADR7–ADR0) at B. The CPU provides a control signal, called Address_Strobe (AS) or Address_Latch_Enable (ALE), to latch the upper eight bits of the address as shown in Figure 9-18b.

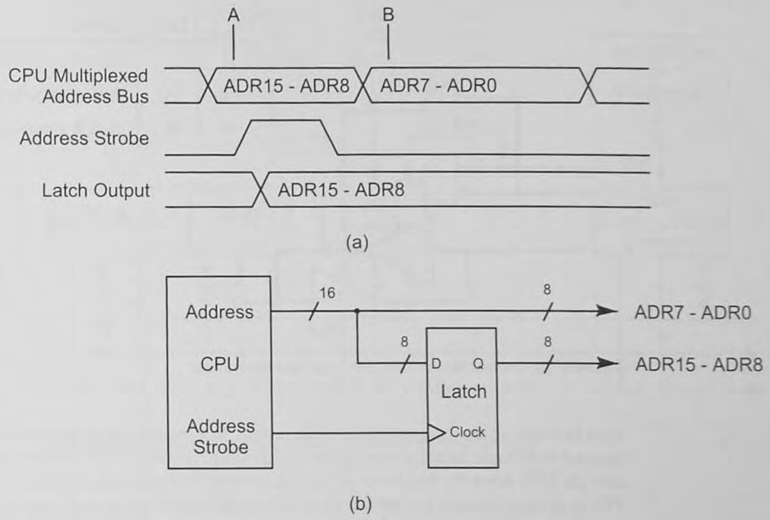


Figure 9-18 Multiplexed address bus. (a) Bus timing. (b) Bus demultiplexer.

Many modern microcontrollers use the multiplexed pin idea for more than just multiplexing an address bus. The microcontrollers have many internal functions that share I/O pins by multiplexing their use depending on what function is enabled. Chapter 15 (Section 15-4) shows how to use these I/O pins to create external address, data, and control buses.

Bidirectional Bus Transceiver

The data bus is bidirectional because data must flow into and out of the CPU. In many systems, a bidirectional data bus buffer, or bus transceiver, such as that shown in Figure 9-19, is used between the CPU and the rest of the system. The OE_L must be low to enable the three-state buffers, and DIR controls the direction of data flow. The bus transceiver provides additional current to drive more devices on the data bus.

Synchronous and Asynchronous Buses

Synchronous and *asynchronous* refer to bus timing protocols that define how and when devices are to respond to data transfers. In Section 9.2, the READ_L and WRITE_L control signals show how the CPU informs external devices that data are either now available on the data bus, in an output operation, or about to be taken, in an input operation. The CPU completes the bus transfer in one cycle of the bus clock, and all devices must respond within this time. This is an example of a *synchronous* bus protocol. The clock may or may not be part of the signals included in the control bus. The problem with the synchronous bus is that the clock frequency

OE_L	DIR	Output
L	L	Bus B data to Bus A
L	H	Bus A data to Bus B
H	X	Bus A isolated from Bus B

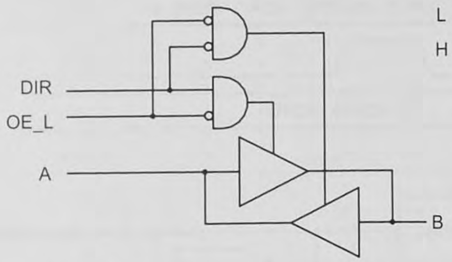


Figure 9-19 One section of 74AC245 octal bus transceiver.

must be based on the slowest device in the system. In a system with a mixture of devices that respond in different times, we would like the design to respond to fast devices quickly and to slow the CPU down for the slower devices. A solution to this problem is the handshaking I/O. This is an *asynchronous* bus protocol and additional control signals are required. We discuss these in more detail in Section 9.7.

Bus Masters and Slaves

Control in buses is designated by reference to bus masters and slaves. In the computer system shown in Figure 9-1, the CPU is the only bus master. All memory, I/O interfaces, and other devices on the buses are slaves. They take their orders from the CPU master. Figure 9-20 shows a scheme called *direct memory access* (*DMA*). Suppose we must retrieve a block of data from a fast I/O device. In the normal, programmed I/O sequence, the data must be input and then stored in memory, using several program steps and clock cycles. If the system includes a DMA controller, the block of data can be transferred directly to memory, bypassing the CPU altogether. Control signals, such as *Hold* and *Hold_Acknowledge* in Intel systems and *Bus_Request* and *Bus_Grant* in Freescale systems, allow the second bus master, the DMA controller, to suspend the operation of the CPU. The DMA controller generates addresses and control signals to transfer the data from the I/O device to the memory.

Bus Arbitration

Bus arbitration is needed if more than one bus master requests the bus at the same time.

Bus arbitration is required in systems with multiple (more than two) bus masters when more than one master wants to control the bus at the same time. In advanced and powerful processors such as the Freescale ColdFire family and the Intel Pentium processors, control signals added to the control bus allow multiple processors to share bus resources. Serial bus interfaces such as the inter-integrated circuit (I^2C) and controller area network (CAN) buses use other bus arbitration schemes. See Chapter 12 for a description of these bus arbitration schemes.

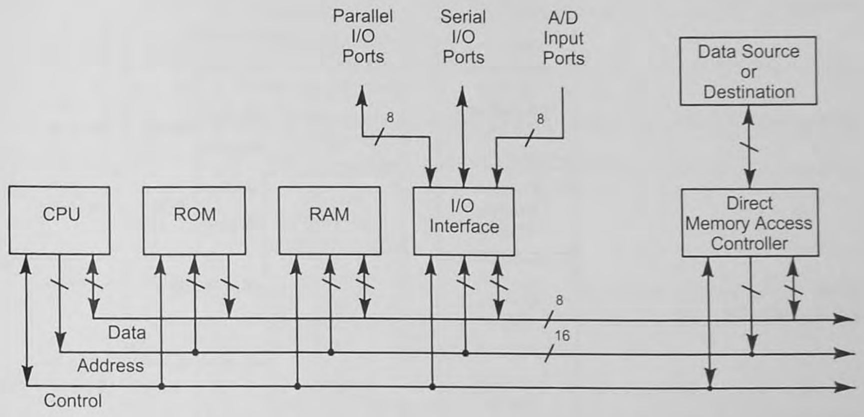


Figure 9-20 Using multiple bus masters for direct memory access.

Additional Bus Control Signals

We have discussed many of the control signals generated by the CPU to help with the timing of reading and writing data. You may find some of the following signals as well.

Clock: Some systems provide the CPU clock as part of the control bus.

Interrupt_Request and Interrupt_Acknowledge: These signals activate the interrupt processing of the CPU. Interrupts will be discussed in detail in Chapter 10.

Reset: The CPU RESET is sometimes provided as a bus signal to reset switching circuits in I/O devices.

9.5 Microcontroller I/O

Modern microcontrollers package into a single chip a variety of I/O devices, such as analog-to-digital converters, timers, and parallel and serial input and output interfaces. Figure 9-21 shows a Flexis microcontroller from Freescale Semiconductor. Like many microcontrollers, the Flexis contains I/O devices that must connect to the outside world. In this case, there are 178 I/O functions that need a pin, far too many for reasonable-size IC packages (the largest Flexis package has 80 pins). The Freescale designers have chosen to multiplex I/O functions onto the port I/O pins (Table 9-6).

Because parallel and serial I/O are built into the microcontroller itself, embedded system designers do not have to design the input and output interfaces covered in the preceding sections, and connecting the microcontroller to external devices is much easier. In Chapter 15 we will learn more about interfacing single-chip microcontrollers to external devices such as switches, LEDs, and expanding parallel I/O techniques.

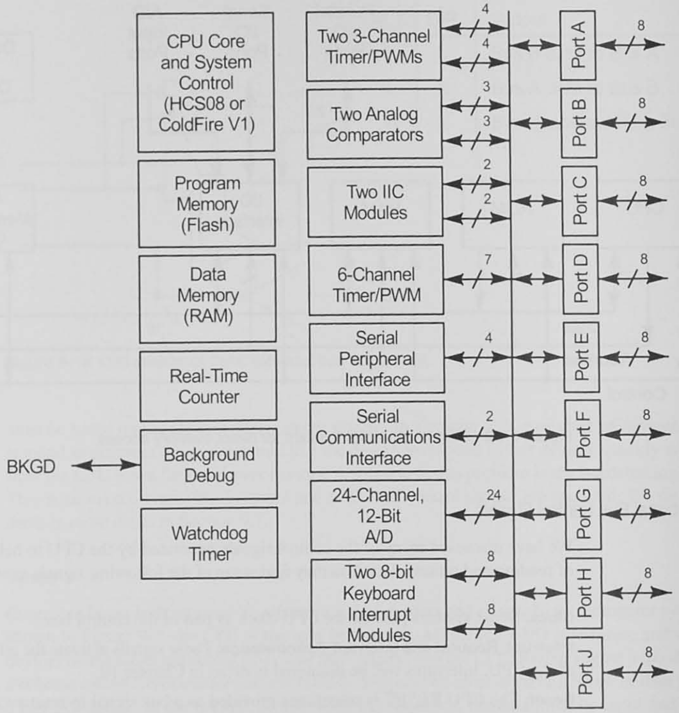


Figure 9-21 The Freescale Flexis microcontroller.

Table 9-6 I/O Functions Multiplexed on Port A

Pin	Multiplexed Pin Functions				
0	Port A Bit-0	Keyboard Interrupt 1, Bit-0	Timer 1, Ch 0	A/D Ch 0	Analog Comparator 1+
1	Port A Bit-1	Keyboard Interrupt 1, Bit-1	Timer 2, Ch 0	A/D Ch 1	Analog Comparator 1–
2	Port A Bit-2	Keyboard Interrupt 1, Bit-2	IIC 1, SDA	A/D Ch 2	Analog Comparator 1 Out
3	Port A Bit-3	Keyboard Interrupt 1, Bit-3	IIC 1, SCL	A/D Ch 3	
4	Port A Bit-4	Background Debug	Mode Select	Reset_L	
5	Port A Bit-5	Interrupt Request	Timer 1, clock	A/D Ch 8	
6	Port A Bit-6		Timer 1, Ch 2		
7	Port A Bit-7		Timer 2, Ch 2	A/D Ch 9	

Exercise 9-2

Make a list of the bidirectional data ports and any internal I/O features present in your microcontroller.

9.6 More I/O Ideas

Buffered I/O

The term *I/O buffering* refers both to the temporary storage of data between the I/O device and the CPU (*data buffering*) and to the conversion between different electrical characteristics found in CPUs, data buses, and I/O devices (*electronic buffering*).

Data Buffering

Data buffering allows a mismatch in the operating speeds of I/O and the CPU.

Data buffering is the storage of data by the I/O device either within the I/O interface or in memory. Figure 9-22 shows a universal asynchronous receiver/transmitter (UART) that is used in serial communications. Serial data bytes are sent by the microcontroller by writing them to the transmit data buffer. If the transmit data parallel in/serial out shift register has completed sending the last byte, the next byte is transferred in parallel from the buffer register to the shift register. The microcontroller may then write another byte to the transmit data buffer. A status bit, called *Transmit Data Register Empty (TDRE)* may be monitored by the microcontroller program to determine when it is safe to output new data.

On the receiving side, as soon as all serial data in bits have been received and shifted into the serial in/parallel out shift register, they are transferred in parallel to the received data buffer. Another serial data byte may then start. This gives the microcontroller time to process the last data byte while a new one is being shifted in. A *Received Data Register Full (RDRF)* status bit may be used by the microcontroller to tell when another byte has been received. We will learn more about the serial I/O interface in Chapter 12.

Electronic Buffering

Electronic buffering provides voltages and currents appropriate for the devices in use. For example, the logic levels for CMOS and TTL devices are different, and TTL/CMOS and CMOS/TTL buffers provide an electronic translation between the two different levels. We will see another example of electronic buffering when we discuss analog-to-digital conversion in Chapters 13 and 15.

9.7 I/O Software

I/O software has an *initialization* part, a data *input/output* part, and must be *synchronized* with the I/O device.

There are three major parts in your I/O software. First is an *initialization* part to set up the function of the ports and the direction of data flow. Second, there are *data input and output* sections that simply read from or write to the appropriate I/O register. There is a third element, namely *software synchronization*. I/O software must synchronize the reading

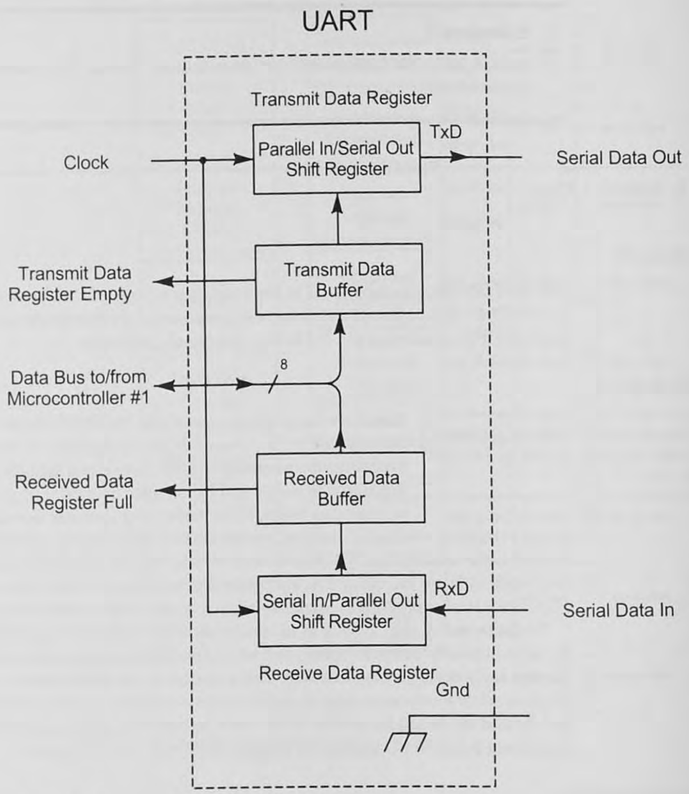

UART
Transmit Data Register
TxD
Clock
Parallel In/Serial Out Shift Register
Serial Data Out
Transmit Data Register Empty
Transmit Data Buffer
Data Bus to/from Microcontroller #1
8
Received Data Register Full
Received Data Buffer
Serial In/Parallel Out Shift Register
RxD
Serial Data In
Receive Data Register
Gnd

Figure 9-22 Input and output data buffering.

and writing of the data with the timing requirements of the I/O device. Typically, microcontrollers are much faster than the I/O devices they serve and must be synchronized by using software and/or hardware techniques. Hardware handshaking techniques for the microcontroller are discussed next, and there are two software I/O synchronization methods. In Chapter 10 we will discuss the interrupts that are used for I/O synchronization.

I/O Synchronization

I/O operations must be *synchronized* with the CPU.

The external I/O device and the CPU must be synchronized in three situations. First, and most common, the CPU may be faster than the I/O device. The design of the interface in Section 9.3 assumes that the output device

(for a CPU write cycle) is ready to receive the data from the CPU by point D in Figure 9-7. For a read cycle, the input device must have placed the data on the bus by point D in Figure 9-8. When these timing requirements are not met, software or hardware must synchronize the fast CPU and the slow I/O. Another synchronization problem occurs when the I/O device needs to transfer data at unpredictable intervals. For example, a CPU that has started an analog-to-digital conversion may have to wait some time for the conversion to be completed before reading the data. A third problem occurs when the I/O device is faster than the CPU, although this is rare. The software and hardware methods shown here can solve the first two problems, although unpredictable transfer times are better synchronized by using interrupts, which we will cover in Chapter 10. The third problem is often solved with a technique called *direct memory access* (DMA), which avoids the CPU altogether. DMA data transfer was shown in Figure 9-20.

Real-Time Synchronization

Real-time synchronization uses a software timing loop.

Real-time synchronization uses a software delay to match the timing requirements of the software and hardware. For example, consider outputting characters to a parallel port at a rate no faster than 1000 characters per second. If we assume that negligible time is spent in getting and outputting each character, a delay of one millisecond is required after each output operation. This could be done with a pair of subroutines: one gets and outputs the characters, and one delays a millisecond before returning.

Real-time synchronization has its problems. It is dependent on the CPU's clock frequency, and it usually has some overhead cycles that cause errors, with the result that the timing is not exact. Thus, depending on the requirements of the application, software timing loops may not be accurate enough. In Chapter 14 we will see how to use the microcontroller timer system to generate highly accurate timing delays. These are far better than software delays, although they also depend on the clock frequency.

Polled I/O

Polled I/O allows the CPU to do other things while it is waiting for the I/O device to become ready.

Polled I/O software uses additional I/O bits as *status bits* for I/O devices. A device receiving data from the microcontroller via Port A could use Port C, bit-0 as a status bit. PTCD-0 will be asserted by the external device when it is ready for new data and deasserted when it is not. Figure 9-23a diagrams the hardware used. Obviously, hardware logic is required in the external device to assert and deassert this bit. The polling software monitors the status bit and outputs data only when the external device is ready. Example 9-5 shows a program segment that polls Port C, bit-0 to determine when it is safe to output more data to Port A.

Polled input software (and hardware) is similar to polled output software. When the software is ready to input new data from the external input device, it checks the RDY_IN signal on PTCD-1 and waits until it is asserted by the external device before inputting the data. The polling software can be doing other things while it is waiting for the external device to supply new data. Figure 9-23b shows the signals needed for input and output polling, and Example 9-5 gives a sample of program code.

At this point you might reasonably ask, "In the polled input scenario, how does the external device know when the CPU has taken its data?" The RDY_IN bit is information *from* the external device *to* the microcontroller. There is no corresponding timing information going in the other direction to let the external device know that the microcontroller has taken the current data and that it is safe to supply new. The solution to this problem can take two forms. First, RDY_IN could activate an *interrupt* bit in Port C and generate an interrupt to ensure the CPU takes the data in a timely fashion. We will discuss this procedure more completely in Chapter 10. Second, *handshaking I/O* can be used as discussed in the next section.

Example 9-5 Using a Status Bit for Output and Input Polling

```
/*************************************************************
 * Program example showing how to use a status bit
 * to determine when an output device is ready to
 * accept data and when an input device has data available
 ************************************************************/

/*************************************************************
 * Define the microcontroller I/O ports used.
 ************************************************************/
#define PTAD (*(volatile unsigned char *) 0x0000) /* Port A */
#define PTBD (*(volatile unsigned char *) 0x0001) /* Port B */
#define PTCD (*(volatile unsigned char *) 0x0004) /* Port C */
/* Define the status bits on Port C */
#define RDY_OUT 1   /* bit 0 */
#define RDY_IN 2    /* bit 1 */
/*************************************************************/
void main(void) {
  unsigned char out_data, in_data;
  /* Initialize the microcontroller's I/O */
  /* . . . */
  /* Output data to the external output device */
  /* Wait until status bit Port J, bit 0 is a 1 */
    while ( (PTCD & RDY_OUT) == 0){
    }
  /* Now output the data */
    PTAD = out_data;

  /* . . . */

  /* Wait for data to be ready from the external
   * input device */
    while ( (PTCD & RDY_IN) == 0){
    }
  /* Data is there, read it */
    in_data = PTBD;
}
```

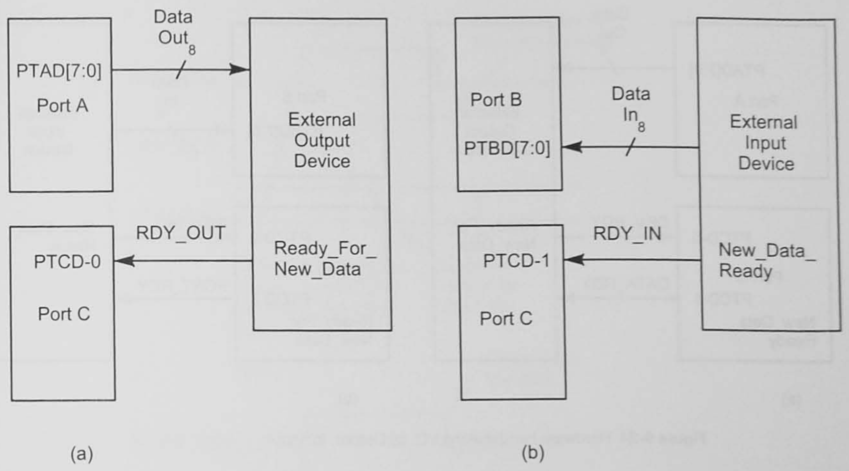


Figure 9-23 (a) Output polling. (b) Input polling.

Microcontroller Internal Polled I/O

Most microcontroller internal I/O devices have status bits, called *flags*, that allow polling or interrupt I/O synchronization. For example, when the UART transmitter shown in Figure 9-22 is being used, one must not output any new data before the last data byte has been sent. The UART sets a *transmit data register empty flag (TDRE)* when all bits have cleared the transmit data register. Polling software should monitor this flag to tell when it can output new data.

Handshaking I/O

Handshaking or *flow control* allows the source device to send data only when the destination device is ready for it.

Handshaking I/O can solve the problem of the source device not knowing when the destination device is ready to receive data. Handshaking is also called *flow control*. Figure 9-24 shows the hardware picture for output and input handshaking. There are a variety of schemes to accomplish handshaking depending on the timing requirements of each device and the microcontroller. One such scheme is shown Figure 9-25.

The I/O handshaking software in Example 9-6 consists of three parts: the initialization, output handshaking, and input handshaking.

1. **I/O initialization:** The initialization code must initialize all registers you are going to use. For a bidirectional port, the data direction registers to control the direction of the bits in the register must be set.

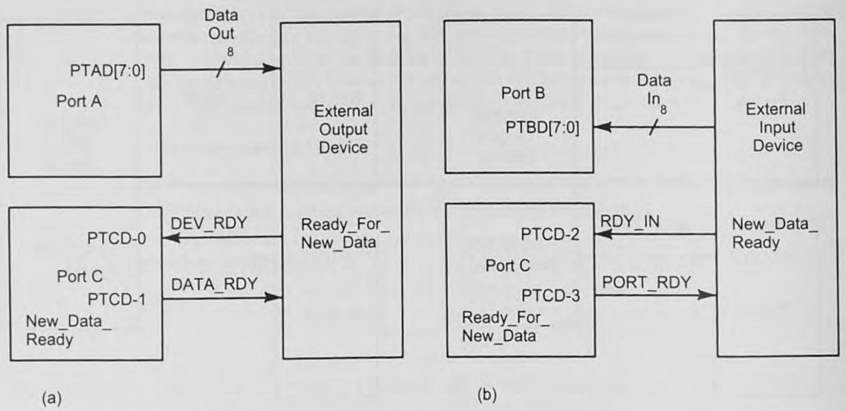


Figure 9-24 Hardware handshaking I/O. (a) Output. (b) Input.

2. **Output handshaking:** When the program is ready to output new data, it starts the handshaking process by asserting the DATA_RDY signal on Port C, bit-1. It then waits until the external device is ready for new data by polling the DEV_RDY bit on Port C, bit-0. When DEV_RDY is asserted (there must be hardware in the external output device that asserts this signal when it is ready to receive data), the program knows that the external device has processed the last data and is ready for new. It then outputs the new data to Port A[7:0] and deasserts the DATA_RDY signal. Often the external device can use this DATA_RDY negative edge to latch the output data. The external device then deasserts DEV_RDY until it is ready for new data again. Figure 9-25a shows the timing diagram associated with this output handshaking transfer of data.

3. **Input handshaking:** Input handshaking (Figure 9-24b) is very similar to output handshaking. When the microcontroller is ready to receive new data, it asserts the PORT_RDY control signal on Port C, bit-3, and begins polling RDY_IN on Port C, bit-2. When the external device has new data ready, it places it on Port B[7:0] and asserts RDY_IN. The microcontroller takes the data and deasserts its handshaking signal PORT_RDY. The external device then deasserts RDY_IN, and the cycle repeats. Figure 9-25b shows a timing diagram for this process.

A common variation on both input and output handshaking themes is that the DEV_RDY and RDY_IN status bits generate interrupts in the microcontroller. This is a good scheme to implement if there are timing issues to be resolved, and it allows the program to go about other business while waiting for the data to be ready.

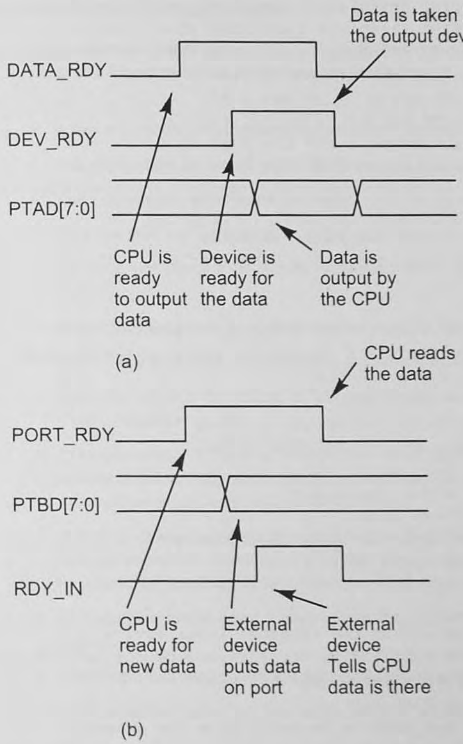

Data is taken by the output device
DATA_RDY
DEV_RDY
PTAD[7:0]
CPU is ready to output data
Device is ready for the data
Data is output by the CPU
(a)
CPU reads the data
PORT_RDY
PTBD[7:0]
RDY_IN
CPU is ready for new data
External device puts data on port
External device Tells CPU data is there
(b)

Figure 9-25 Handshaking timing. (a) Output. (b) Input.

Example 9-6 Handshaking I/O Software

```
/*************************************************************
 * Program example showing how to use status bits for
 * handshaking I/O synchronization.
 ************************************************************/

/*************************************************************
 * Define the microcontroller I/O ports used.
 ************************************************************/
```

```
#define PTAD (*(volatile unsigned char *) 0x0000) /* Port A */
#define PTBD (*(volatile unsigned char *) 0x0001) /* Port B */
#define PTCD (*(volatile unsigned char *) 0x0004) /* Port C */
/* Define the status bits on Port C */
#define DEV_RDY 1      /* bit 0 */
#define DATA_RDY 2     /* bit 1 */
#define RDY_IN 4       /* bit 2 */
#define PORT_RDY 8     /* bit 3 */
/*****************************************************/
void main(void) {
  unsigned char out_data, in_data;
  /* Initialize the microcontroller's I/O */

  /* . . . */

  /* Output data to the external output device */
  /* Assert DATA_RDY to let the external device that
   * data are available */
    PTCD |= DATA_RDY;
  /* Wait until the device is ready */
    while ( (PTCD & DEV_RDY) == 0){
    }
  /* Now output the data */
    PTAD = out_data;
  /* Lower DATA_RDY to latch the data */
    PTCD &= ~DATA_RDY;

  /* . . . */

  /* Get data from the external input device */
  /* Assert PORT_RDY to let external device know we are
   * ready to receive data */
    PTCD |= PORT_RDY;
  /* Wait for data to be ready from the external
   * input device */
    while ( (PTCD & RDY_IN) == 0){
    }
  /* Data is there, read it */
    in_data = PTBD;
  /* De-assert PORT_RDY */
    PTCD &= ~PORT_RDY;
}
```

I/O Synchronization with Interrupts

Another way to synchronize I/O and the CPU is to use interrupts. Interrupts allow an I/O device to signal the CPU that it is ready to be serviced. Interrupts will be covered in detail in Chapter 10.

9.8 Conclusion and Chapter Summary Points

In this chapter we have discovered how computer buses work and how to interface I/O devices to a bus. The key elements of the chapter are the following.

- A bus is a parallel, bidirectional information pathway.
- Sources transfer information to a destination over a bus.
- Three-state gates allow multiple sources to be on a common bus line.
- No more than one source can be active at a time.
- An input interface is a set of three-state gates between an information source and a data bus.
- An output interface is a set of latches between the bus and the destination device.
- One-of-many sources or destinations are chosen by addressing and address decoding.
- The CPU controls the timing of data transfers by generating READ and WRITE control signals.
- I/O addressing may be done with memory-mapped I/O, in which case any memory reference instruction can access I/O, or separate I/O, for which special input and output instructions are included in the instruction set.
- You may choose to decode the entire address bus (full address decoding, which leads to more expensive decoders) or a subset of the address bus (reduced addressing, less expensive but resulting in redundant use of addresses).
- I/O synchronization often is necessary to synchronize a fast CPU with a slow I/O device.
- If multiple bus masters require the bus simultaneously, bus arbitration is required.
- Modern microcontrollers have extensive I/O capabilities integrated into a single chip.

9.9 Problems

Explore

9.1 List parallel I/O devices used with computers you are familiar with, either in the laboratory or in a personal computer.

9.2 Which type of I/O addressing, separate I/O or memory mapped, uses memory reference instructions to access I/O devices? [a]

9.3 Which type of I/O addressing, separate I/O or memory mapped, requires a control signal called "I/O request" to access I/O devices? [a]

9.4 Show the schematic symbol for an 8-bit data bus. [a]

Stimulate

9.5 Describe the advantages of the three-state gate over the open-collector gate when the application entails multiple sources on a data bus. [a, g]

9.6 Describe the advantages of the open-collector gate over the three-state gate when the application entails multiple sources on a control signal. [a, g]

9.7 Why are three-state gates used in an input interface? [a]

9.8 The following control signals are associated with a CPU in a microcontroller.

MEMRQ: asserted when a memory operation is ongoing
IORQ: asserted when an I/O operation is ongoing
WR: asserted when a write operation is ongoing
RD: asserted when a read operation is ongoing
ADROK: asserted output from an address decoder
Write logic equations for the following: [c]

a. A correctly timed latch signal for an output port.
b. A correctly timed three-state control signal for an input port.
c. A correctly timed signal to select memory for reading or writing.

9.9 In a parallel output operation, how is the synchronization of the data transfer between CPU and a data latch consisting of eight, D-type flip-flops accomplished? [a]

9.10 Briefly explain the difference between separate and memory-mapped I/O. [a]

Challenge

9.11 Discuss the consequences of a CPU designer's decision to implement memory-mapped I/O instead of separate I/O. What does it mean to the CPU designer, and what does it mean to you, the system designer using the CPU? [e]

9.12 Design a decoder using a 74HC138 decoder to produce BLOCK_SELECT_L signals to enable memory and I/O devices to the following specifications: [c]

The microcontroller has a 16-bit address bus
The memory is to be addressed in eight, 8K byte blocks in which
1 block is to be used for I/O
1 block is to be used for future I/O expansion
1 block is to be used for ROM in the highest memory addresses
1 block is to be used for future ROM expansion
1 block is to be used for RAM in the lowest memory addresses
1 block is to be used for future RAM expansion
2 blocks will never be used

9.13 A 74HC138 decoder has the following address bits assigned to its inputs:

ADR	74HC138 Pin
A7	A2
A6	A1
A5	A0
A4	E1
A3	E2_L
A2	E3_L

A1 and A0 are don't cares.

Assume an 8-bit address and make a table similar to Table 9-4 showing what address each output responds to. [b]

9.14 Given the reduced address decoder shown in Figure 9-16a, what decoder output should be chosen for address 599_{10}? [b]

9.15 Example 9-5 shows a segment of code that waits until a status bit asserted by the output device is ready to accept new data before writing the data to the port. What can go wrong with this arrangement, and what might you do to make the system better? [b, c]

9.16 Adapt Figure 9-24 and Example 9-6 to allow two microcontrollers to transfer data back and forth. Assume Microcontroller #1 outputs data on Port A, inputs data on Port B, and uses Port C for its I/O handshaking bits. Assume Microcontroller #2 inputs data on Port A, outputs data on Port B, and uses Port C for its handshaking bits. Show your hardware design, and write programs for each of the microcontrollers to transfer the data. Your programs should not hang up waiting for data from one or the other. [c]

Reflect on Learning

9.17 Compare software polling with hardware handshaking I/O synchronizing.

9.18 Discuss the relative merits of software and hardware switch debouncing.

9.19 List five things that you learned while studying this chapter.

9.20 Why are timing signals necessary for the input and output of data?

10 Interrupts and Real-Time Events

Objectives

This chapter shows how an important external or internal event can interrupt the normal flow of a program. We will discuss how the CPU finds out which of several interrupting devices needs service. When an interrupt occurs, an interrupt service routine is executed. We will discuss how interrupt routines work and give guidelines for writing them.

10.1 Introduction

An *interrupt* is an important asynchronous event that requires immediate attention.

An *exception* is an event even more important than an interrupt.

An IRQ is the *interrupt request* signal from a device needing some special action to be taken.

An interrupt is a way for an *important asynchronous event* to be recognized and taken care of (*serviced*) by the CPU executing instructions in a normal program. Consider, for example, a computer system controlling an oil refinery. It has sensors that measure the chemical composition of the product being refined and outputs controlling the process. Figure 10-1a shows a typical process control software loop to do this. The time taken to go around the loop depends on the complexity of the control algorithms and the speed of the processor. Now consider an important external asynchronous event: a fire breaks out in the oil refinery! If the control computer is responsible for activating fire suppression measures, the program should respond immediately instead of waiting for the software to come around the loop to check on the fire detection sensors. On the other hand, we do not want to write a program that is checking the fire sensors all the time, or even frequently, because too much checking would take time away from the control calculations. This is an ideal application for an interrupt. The interrupt is caused by an external device, the fire sensor, generating a signal called *interrupt request*, or IRQ. The interrupt request is asynchronous. That is, it can happen at any time, not necessarily corresponding to any particular time in the instruction execution sequence of the CPU. The IRQ requests the program to take immediate action by executing an *interrupt service routine (ISR)* or *interrupt handler.* Figure 10-1b shows an interrupt service routine added

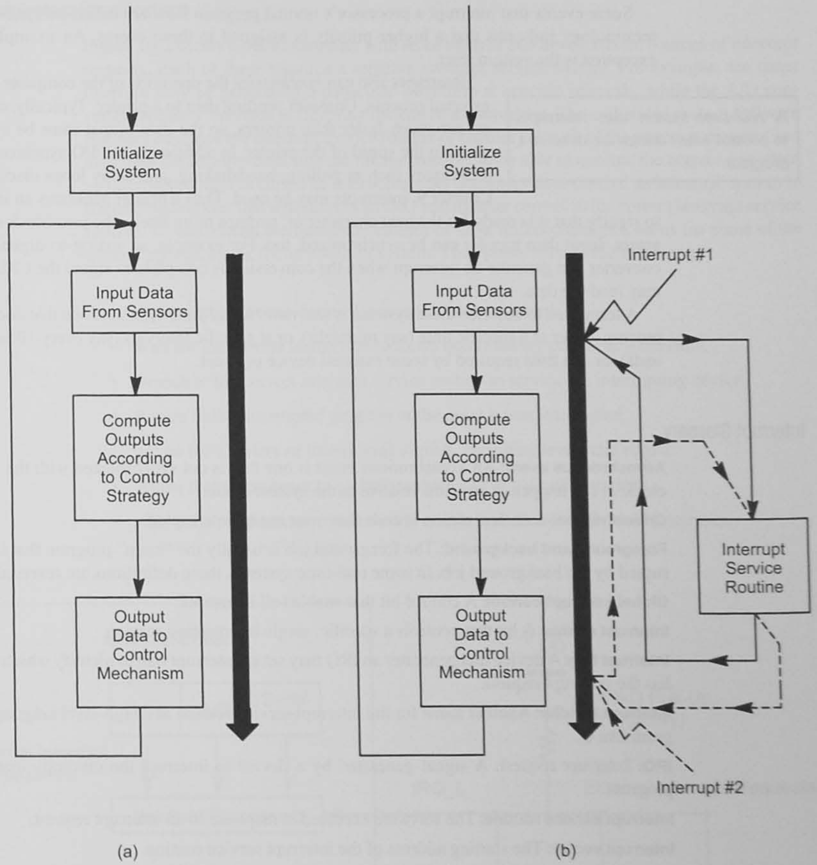

Initialize System
Input Data From Sensors
Compute Outputs According to Control Strategy
Output Data to Control Mechanism
(a)
Initialize System
Interrupt #1
Input Data From Sensors
Compute Outputs According to Control Strategy
Interrupt Service Routine
Output Data to Control Mechanism
Interrupt #2
(b)

Figure 10-1 Process control software. (a) Typical flow without interrupts. (b) Process control software with interrupts.

to the process control software of Figure 10-1a. When Interrupt #1 occurs, the program branches to and executes the interrupt service routine and then returns to the main program at the point of interruption. The same sequence occurs when Interrupt #2 occurs; but notice that the return is to the place where the interrupt occurs. This makes interrupt service routines similar to subroutines or functions. Nevertheless, as we will see, there is more to the interrupt service routine.

Some events that interrupt a processor's normal program flow are called *exceptions*. This terminology indicates that a higher priority is assigned to these events. An example of an exception is the system reset.

A *real-time system* uses interrupts to control *when* things are done in a program.

Interrupts also can *synchronize* the operation of the computer with an external process. Consider sending data to a printer. Typically, computers are much faster than printers, so the data output must be synchronized to the speed of the printer. In addition to the I/O synchronization techniques such as polling, handshaking, and delay loops discussed in Chapter 9, interrupts may be used. Thus a printer generates an interrupt to signify that it is ready for the next character or, perhaps more likely, the next block of characters. Input data transfer can be synchronized, too. For example, an analog-to-digital (A/D) converter can generate an interrupt when the conversion is complete to signal the CPU that it may read the data.

A term used to describe these systems is *real time*. A real-time system is one that does some process, either at a specific time (say midnight), or at specific intervals (say every 10 milliseconds), or at a time required by some external device or event.

Interrupt Glossary

Asynchronous event: An asynchronous event is one that is not synchronized with the system clock. It can happen at any time relative to the system clock.

Critical region: A critical region is code that must not be interrupted.

Foreground and background: The foreground job is usually the "main" program that is interrupted by the background job. In some real-time systems, these definitions are reversed.

Global interrupt control: A control bit that enables all interrupts.

Interrupt enable: A bit that controls a specific, single interrupting service.

Interrupt flag: A device that generates an IRQ may set an interrupt flag to identify which device has the interrupt request.

Interrupt handler: Another name for the interrupt service routine in a high-level language program like C.

IRQ: Interrupt request. A signal generated by a device to interrupt the currently executing program.

Interrupt service routine: The software executed in response to an interrupt request.

Interrupt vector: The starting address of the interrupt service routine.

Interrupt vector table: A table in memory that contains the interrupt vectors for all interrupt service routines.

Latency: Interrupt latency is the time delay from the initiation of the interrupt request by the hardware to the start of the interrupt service routine. Elements contributing to interrupt latency are the time to complete the current instruction, the time to save the machine context and return address on the stack, and the time to find the correct interrupt service routine.

Machine state or context: The state of the registers and the condition code or status bits at any time.

Pending interrupt: An interrupt that has occurred but has not yet been serviced.

Interrupt System Specifications

Figure 10-2 shows a microcontroller with three internal and three external sources of interrupt requests. Each of these requires a separate interrupt service routine. For example, the timer interrupt may be generating a particular waveform at specific intervals, while the A/D converter module's interrupt is signaling the end of a conversion. The embedded system software engineer develops the interrupt service routine or handler for each of the interrupt requests and then uses the features built into the microcontroller to be able to execute the correct one when an interrupt request occurs. The microcontroller designers have created an interrupt system in the microcontroller that will react to an IRQ and transfer control to the correct interrupt service routine. To understand the hardware features of most microcontrollers, let us list some of the general specifications for an interrupt system. The system is to do the following:

- Allow asynchronous events to occur and be recognized.
- Wait for the current instruction to finish before taking care of any interrupt.
- Branch to the correct interrupt service routine to service the interrupting device.
- Return to the interrupted program at the point it was interrupted.
- Allow for a variety of interrupting signals, including levels and edges.
- Allow the programmer to globally enable and disable all interrupts.
- Allow the programmer to selectively enable and disable individual interrupts.

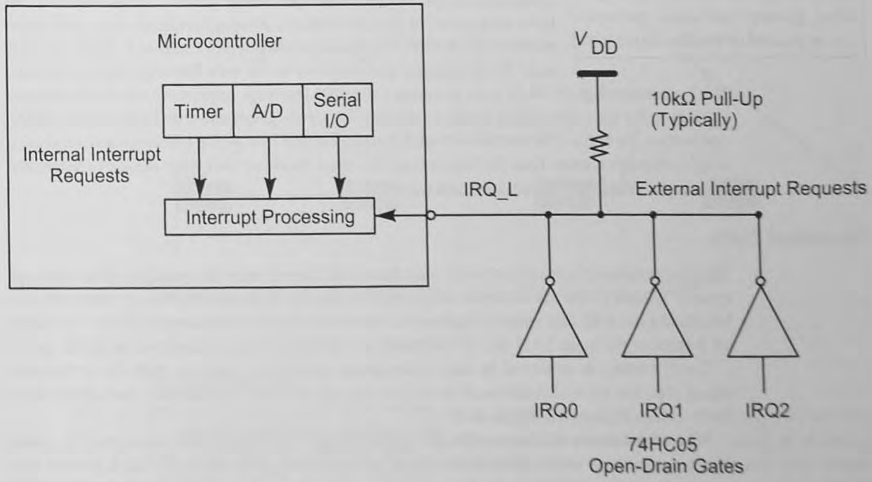


Figure 10-2 Internal and external interrupt requests.

- Disable further interrupts while the first is being serviced.
- Deal with multiple sources of interrupts.
- Deal with multiple, simultaneous interrupts by enacting a prioritization system.

Asynchronous Events and Internal Processor Timing

The current instruction must be finished before an interrupt request is acted upon.

Figure 10-3a shows a program execution time line. The ticks along the line represent the start of each instruction that a normal program executes in sequence. The normal program does not specify *when*, in a real-time sense, an instruction is to be executed, just the *sequence* of instructions. Asynchronous events, the IRQs, can occur at any time.

Figure 10-3b shows an expanded time line. Chapter 2 showed that an instruction execution cycle consists of the instruction fetch and instruction execution parts. The sequence controller can be modified to check for an interrupt request before it fetches the next instruction. More states are added to sample the interrupt request and generate more control signals. This change allows the CPU to finish the current instruction and then to service the interrupt by entering a special interrupt processing sequence; otherwise, it fetches the next instruction.

10.2 The Interrupt Process

The Interrupt Request

All interrupts can be enabled or disabled globally. Individual interrupts can be enabled or disabled separately.

Microcontrollers have both internal and external sources of interrupts. The internal requests come from the internal systems, such as a timer, and from exceptions or error conditions. External requests may come from external I/O devices like those investigated in Chapter 9. Each interrupt must be recognized and serviced by its own interrupt service routine. We have seen (Figure 10-2) how to connect external interrupt request signals from multiple devices to the microcontroller. Multiple interrupting devices may use wired-OR (wired-AND), open-drain, or open-collector gates to pull the request line low. When multiple devices share a single interrupt request line, the microcontroller must check, or poll, each device to determine which one generated a given interrupt request.

The Interrupt Enable

The programmer of a microcontroller must have total control over the operation of the interrupt system. *Global control* is achieved with a bit that either *enables* or *disables* all interrupts or a bit called a *mask bit* that *masks* (disallows) or *unmasks* (allows) all interrupts. When the enable bit is reset or the mask bit is set, all interrupts are disabled or masked and are not acted upon.

Local control is achieved in each interrupting subsystem, such as each timer channel, which also has an *enable bit* used to *enable* (allow) or *disable* (disallow) *that device* from interrupting (Figures 10-4 and 10-5).

When the microcontroller is reset, the global control bit disallows all interrupts. This gives your program time to initialize all hardware and software, particularly the stack pointer register, before interrupts occur. As Figure 10-4 shows, interrupts must be enabled (unmasked) globally AND enabled locally for the interrupt request to be generated.

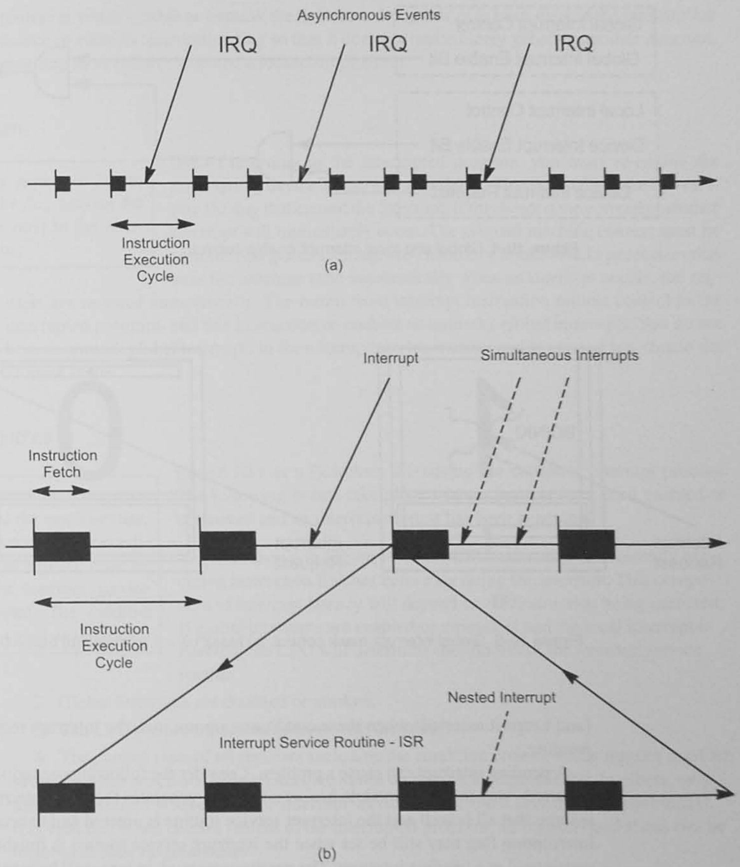


Figure 10-3 Interrupts. (a) Instruction flow. (b) Expanded instruction timing.

Pending Interrupts

A *pending* interrupt is one that will be taken care of after the current interrupt is done or after interrupts have been unmasked and enabled.

For a CPU to act upon a device interrupt, we see from Figures 10-4 and 10-5 that the interrupt enable or interrupt mask bits must be high and low, respectively. In addition, the device interrupt enable bit must be set. If the device asserts the interrupt request

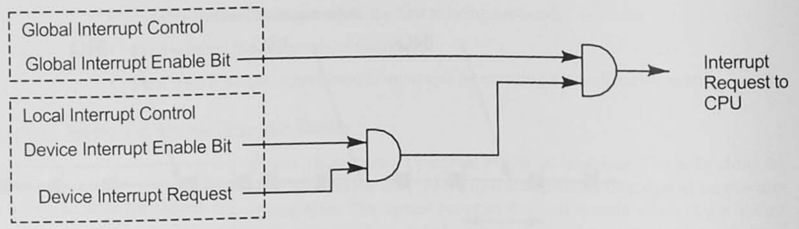


Figure 10-4 Global and local interrupt enable control.

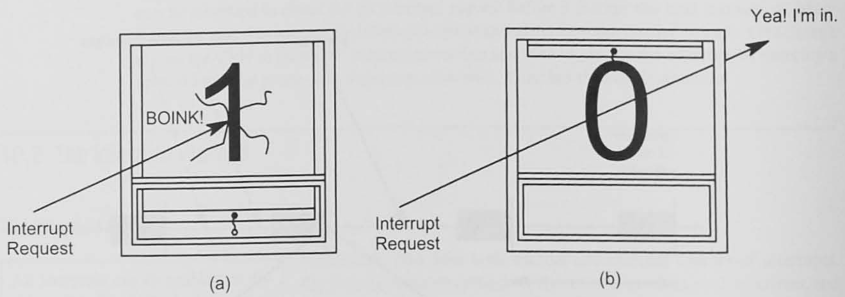


Figure 10-5 Global interrupt mask control. (a) Mask bit = 1 to mask. (b) I-bit = 0 to unmask.

(and keeps it asserted) when these conditions are not met, the interrupt request is said to be *pending*.

A pending interrupt can cause a problem. Consider the following scenario. An interrupt has occurred—say the timer module has set a flag that generates the local interrupt request. Let's assume that all is well and the interrupt service routine is entered and executed. Even so, the interrupting flag may still be set when the interrupt service routine is finished and if so, it is understood as a pending interrupt. The pending interrupt, in turn, will be asserted immediately when interrupts are re-enabled at the end of the interrupt service routine, and the ISR will be executed continuously. Your software, therefore, must reset the interrupting flag in the interrupt service routine, (if it is not reset by some other hardware mechanism) before returning to the interrupted program.

The Interrupt Disable

When an interrupt occurs, global interrupts are disabled so that further interrupts cannot occur. Although you should avoid nested interrupts, they can be allowed in the interrupt service routine if you re-enable or unmask them. Before doing this, you must disable the interrupting source or clear its interrupting flag so that it does not immediately generate another interrupt, resulting in an infinite loop and a locked-up program.

The Interrupt Return

If an interrupt is generated by an internal source, the *flag* causing the interrupt must be *reset* in the interrupt service routine.

Before returning to the interrupted program, you must *re-enable the interrupting device's interrupt capability*. This is usually done by *resetting the flag* that caused the interrupt. If this is not done correctly, another interrupt will immediately occur. The original machine context must be restored and global interrupts re-enabled or unmasked. In processors that save the machine state automatically when an interrupt occurs, the registers are restored automatically. The return from interrupt instruction returns control to the interrupted program, and this instruction re-enables or unmasks global interrupts. You do not have to unmask global interrupts in the interrupt service routine, and in general you should not unmask them.

The Interrupt Sequence

The current state of the microcontroller, also called the *machine* state, including all registers and the condition code or status register, must be saved before the interrupt service routine is executed. The machine state is normally saved on the stack.

Figure 10-6 is a flowchart illustrating the complete interrupt process. The following events take place when interrupts have been enabled or unmasked and an interrupt request has been generated.

1. As shown in Figure 10-3, the CPU waits until the currently executing instruction finishes before servicing the interrupt. This component of interrupt latency will depend on the instruction being executed. If global interrupts are enabled or unmasked and the local interrupt is enabled, the CPU will determine the address of the interrupt service routine.
2. Global interrupts are disabled or masked.
3. The CPU pushes the return address onto the stack.
4. The current state of all registers including the condition code or status register must be saved. In some processors all registers are saved on the stack automatically. In others, only the return address is saved and the interrupt service routine must save the machine context. It is vital that when control returns to the interrupted program, all registers and status bits be restored to their original state.
5. The CPU branches to the interrupt service routine. The interrupt service routine deals with the specific requirements for the interrupt, resets the interrupting flag, and then executes a return-from-interrupt instruction.

Exercise 10-1

Determine if your microcontroller saves all registers on the stack when an interrupt occurs.

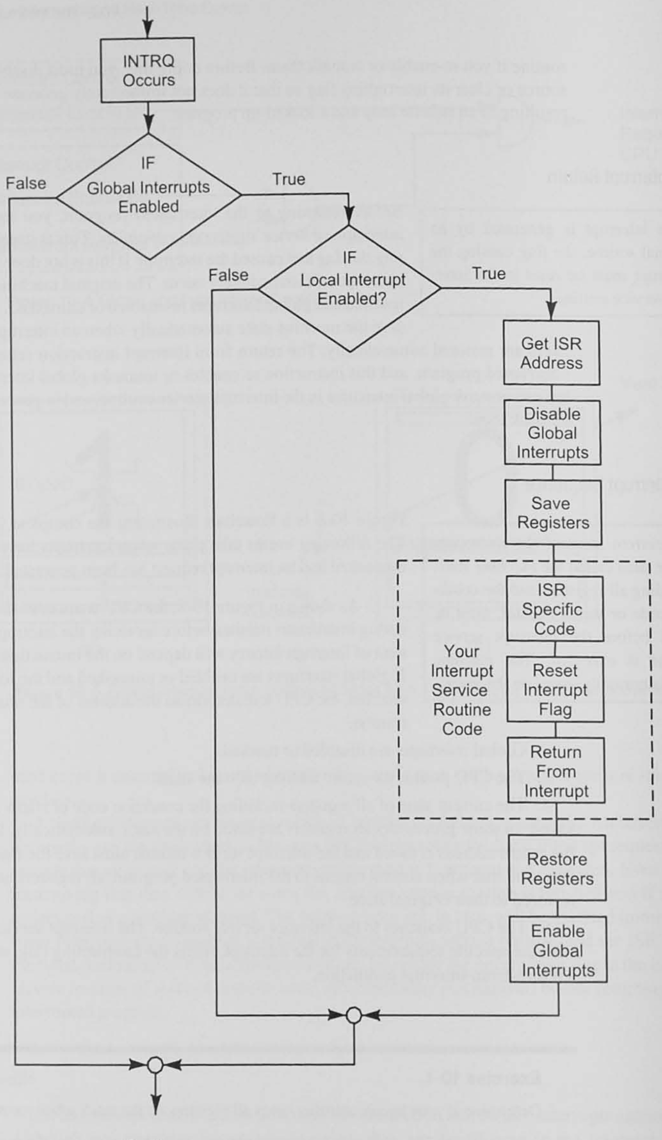


Figure 10-6 Interrupt process flowchart.

10.3 Multiple Sources of Interrupts

The interrupt system must deal with multiple devices generating interrupts. Allowing for these requires the system to do the following:

- Determine which of the multiple devices has generated the IRQ to be able to execute the correct interrupt service routine.
- Resolve simultaneous requests for interrupts with a prioritization scheme.

When there are multiple interrupting devices, the CPU must resolve each interrupt event by determining which device has generated the interrupt request.

A system with three external and three internal interrupting devices is shown in Figure 10-7. All internal and external I/O devices are connected to the CPU with standard input or output interfaces like those designed in Chapter 9. The interrupt request signals generated by each of the interrupt sources are input to the CPU. The CPU responds to the interrupt request by transferring program control to the interrupt service routine. There are two methods of finding out which of many devices may have generated the interrupt request: vectoring and polling.

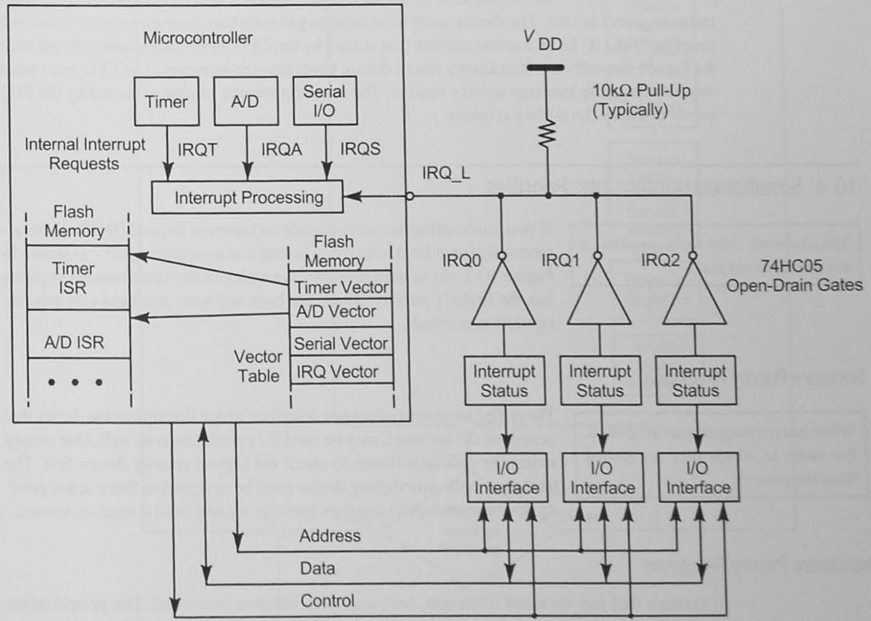


Figure 10-7 Multiple interrupts.

Vectored Interrupts

The interrupting processing hardware detects which device interrupted and then uses a *vector* to transfer to the interrupt service routine in *vectored interrupts.*

A *vectored interrupt* is the most common way to resolve which of several interrupting devices has generated the interrupt request. A vector is simply an address; in this case, it is the starting address of the interrupt service routine. In modern microcontrollers, a specific area of non volatile memory is dedicated to the vectors (addresses) for all possible interrupting devices (timer, analog-to-digital converter, external interrupting devices, etc). This memory is called the *interrupt* vector table. As Figure 10-7 shows, the vectors point to the correct interrupt service routine. The interrupt processing hardware in Figure 10-7 detects which of the interrupt request lines, IQRT, IRQA, IRQS, or IRQ_L, is being asserted and, provided everything is enabled properly, the CPU fetches the address of the interrupt service routine from the vector location and branches to that address to start executing the interrupt service routine.

Polled Interrupts

The interrupting device is found with *software* in a *polled interrupt system.*

Whenever multiple devices share an interrupt request line, like the three external devices shown in Figure 10-7, you may use a polling strategy to determine which device needs servicing. Polling is a software process in which the CPU reads each of the potential interrupting device's interrupt status registers in turn. The device must have logic to generate the interrupt request signal and to set an "I did it" bit in a status register that is read by the CPU. When a register is found with the bit set, the software then knows which device generated the interrupt. The CPU must reset this bit during the interrupt service routine. The interrupt service routine accessed by the IRQ vector contains the polling routines.

10.4 Simultaneous Interrupts: Priorities

Simultaneous interrupts require a prioritization scheme.

If two interrupting devices generate an interrupt request (INTRQ) simultaneously (or at least within one instruction execution cycle), as shown in Figure 10-3, the system must resolve which of the simultaneous requests has the highest priority. There are both software and hardware priority resolution methods.

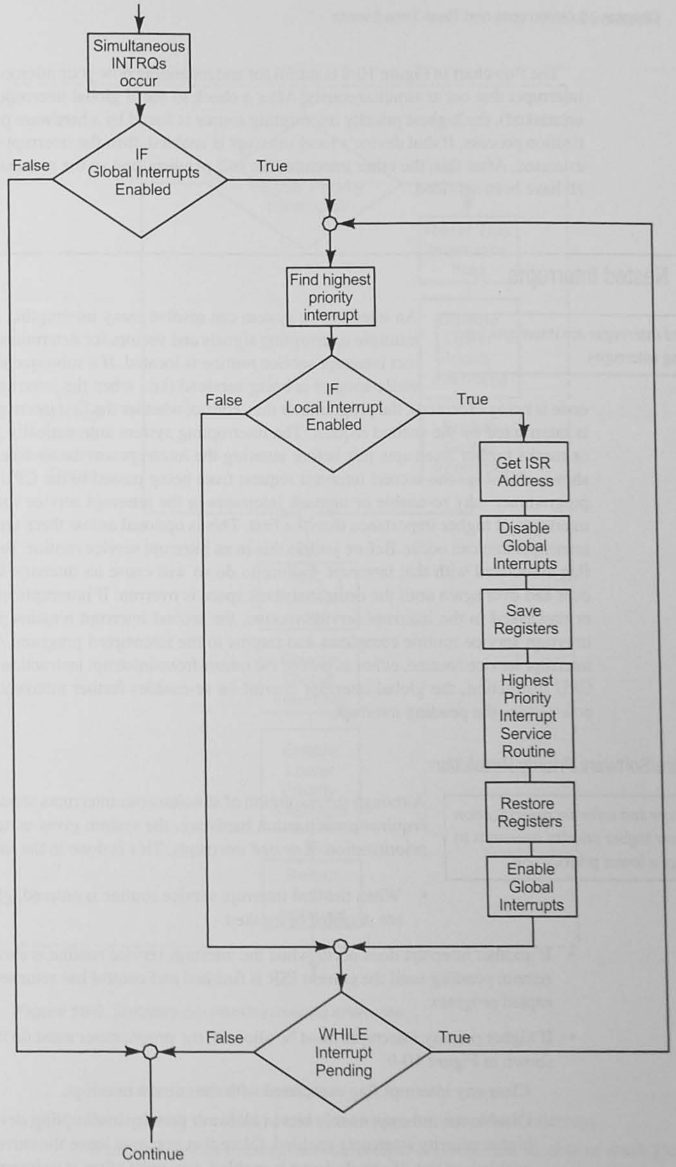
Software Priority Resolution

When interrupting devices are polled, the order in which they are polled fixes the priority.

The polled interrupt system just described which determines the device that generated the interrupt, may be used for prioritization as well. One simply writes the polling software to check the highest priority device first. The hardware in the interrupting device must be designed so that a lower priority device continues to assert its interrupt request until it receives service.

Hardware Priority Resolution

In systems that use vectored interrupts, hardware prioritization is needed. The prioritization may be fixed by the design of the CPU, or you may have some limited capability to define the priority levels in your software.



Simultaneous INTRQs occur
IF Global Interrupts Enabled
False
True
Find highest priority interrupt
IF Local Interrupt Enabled
False
True
Get ISR Address
Disable Global Interrupts
Save Registers
Highest Priority Interrupt Service Routine
Restore Registers
Enable Global Interrupts
WHILE Interrupt Pending
False
True
Continue

Figure 10-8 Simultaneous interrupts.

The flowchart in Figure 10-8 is useful for understanding how your microcontroller resolves interrupts that occur simultaneously. After a check to see if global interrupts are enabled (or unmasked), the highest priority interrupting source is found by a hardware or software prioritization process. If that device's local interrupt is enabled, then the interrupt service routine is executed. After this, the other interrupts are still pending, and so the process is repeated until all have been serviced.

10.5 Nested Interrupts

Nested interrupts are interrupts interrupting interrupts.

An interrupting system can resolve many interrupting sources by using multiple interrupting signals and vectors for determining where the correct interrupt service routine is located. If a subsequent interrupt occurs while another is being serviced (i.e., when the interrupt service routine code is being executed), the programmer may control whether the first interrupt service routine is interrupted by the second request. The interrupting system automatically globally disables or masks further interrupts just before entering the interrupt service routine. As Figure 10-4 shows, this stops the second interrupt request from being passed to the CPU for service. The programmer may re-enable or unmask interrupts in the interrupt service routine if there are interrupts of higher importance than the first. This is optional unless there are more important interrupts that can occur. Before you do this in an interrupt service routine, be sure to clear the flag associated with that interrupt. Failure to do so will cause an interrupt to interrupt itself over and over again until the dedicated stack space is overrun. If interrupts are not re-enabled or unmasked in the interrupt service routine, the second interrupt remains pending until the interrupt service routine completes and returns to the interrupted program. At the end of the interrupt service routine, either as part of the return-from-interrupt instruction or by an explicit CPU instruction, the global interrupt control bit re-enables further interrupts. The CPU can now service the pending interrupt.

Hardware/Software Priority Resolution

Hardware and *software prioritization* can allow higher priority interrupts to interrupt a lower priority one.

Although the resolution of *simultaneous* interrupts shown in Figure 10-3 requires prioritization hardware, the system gives us total control over prioritization of *nested* interrupts. This is done in the following way.

- When the first interrupt service routine is entered, global interrupts are disabled or masked.
- If another interrupt does occur while the interrupt service routine is executing, it will remain pending until the current ISR is finished and control has returned to the interrupted program.
- If higher priority interrupts must be allowed, the programmer must do the following, as shown in Figure 10-9.

  Clear any interrupt flag associated with the current interrupt.

  Disable the interrupt enable bits in all lower priority interrupting devices, leaving higher priority interrupts enabled. (Note that you may leave the current interrupt enabled, or not. If you do leave it enabled, you must allow it to interrupt itself.)

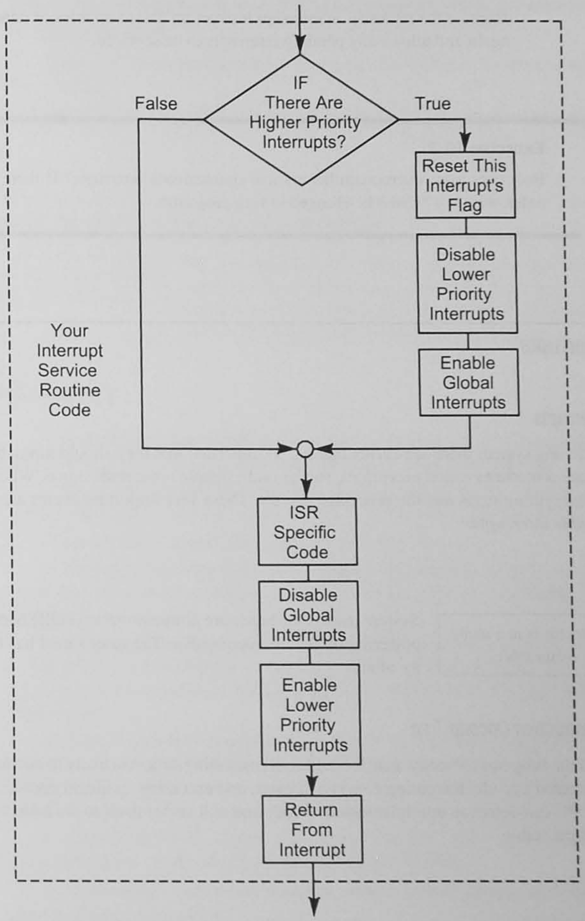

IF There Are Higher Priority Interrupts?
False
True
Reset This Interrupt's Flag
Disable Lower Priority Interrupts
Enable Global Interrupts
Your Interrupt Service Routine Code
ISR Specific Code
Disable Global Interrupts
Enable Lower Priority Interrupts
Return From Interrupt

Figure 10-9 Software prioritization nested interrupts.

  Re-enable or unmask interrupts.

  Proceed with the interrupt service routine for the current interrupt.

  When the current interrupt service routine is completed, disable or mask global interrupts and re-enable the lower priority interrupts that were disabled.

Execute the return from interrupt instruction that re-enables or unmasks interrupts again and allows any pending interrupts to be serviced.

Exercise 10-2

How does your microcontroller resolve simultaneous interrupts? If there is a prioritization order, what is it? Can it be changed in your program?

10.6 Other Interrupts

Nonmaskable Interrupts

In any system there are events that are so important that they should never be masked. These are sometimes called *exceptions,* and a good example is the reset signal. When this is asserted, everything stops and the processor is reset. These very important events are called *nonmaskable interrupts*.

System Reset

The system reset vector is in a memory location in the vector table.

System reset is the hardware *power-on reset* (POR) normally done when powering up the microcontroller. The reset signal has the highest priority of all.

Unimplemented Instruction Opcode Trap

If the program somehow gets lost and starts executing data, it is likely to encounter an unimplemented opcode. Executing data is a disaster, and executing an illegal opcode even worse. The CPU can detect an unimplemented opcode and will vector itself to the address specified in the vector table.

Software Interrupt

The software interrupt is, in effect, a one-byte, indirect branch to a subroutine whose address is in the vector table. Because it operates like the rest of the interrupt system, it is often used to implement debugging breakpoints.

Nonmaskable Interrupt Request

To detect important external events, such as loss of power, an external, nonmaskable interrupt input may be used. Once this interrupt source has been enabled, it cannot be disabled or masked.

Exercise 10-3

Does your microcontroller have nonmaskable interrupts like those just listed? If so, what are they?

10.7 The Interrupt Service Routine or Interrupt Handler

The interrupt service routine is called an *ISR* in assembly and an *interrupt handler* in C.

The interrupt service routine, or interrupt handler, is executed when the vector has been initialized properly, interrupts have been unmasked and enabled, an interrupt has occurred, and the vector has been fetched. Here are some hints for your interrupt service routines.

Interrupt Service Routine Hints

1. **Save the machine context:** If your microcontroller does not automatically save the machine context, you must do it in the ISR before doing anything else.
2. **Re-enable interrupts in the ISR only if you need to:** You must re-enable or unmask global interrupts if there are higher priority interrupts that must be serviced.
3. **Do not allow nested interrupts:** Unless you have to.
4. **Reset any interrupt generating flags in I/O devices:** All devices are different, and each requires somewhat different procedures. If you do not reset the flag, interrupts will be generated continuously.
5. **Do not assume any register contents:** Never assume that the registers contain a value needed in the interrupt service routine unless you have full control over the whole program and can guarantee that the contents of a register never change in the program that is interrupted.
6. **Keep it simple to start:** Learning how to use an interrupt can be frustrating if you try to do too much in the ISR. The first step should be to see if the interrupts are occurring and if the interrupt service routine is being entered properly. After you have found affirmative answers in both cases, you can make the ISR do what it is supposed to do.
7. **Keep it short:** Do as little as possible in the ISR. This reduces the latency in servicing other interrupts should they occur during the current ISR.
8. **If necessary, restore the machine context before returning:** Some processors do this automatically, some do not.

Interprocess Communication

Frequently, an interrupt service routine and another part of the program must exchange information. For example, an ISR may be incrementing a counter each time a product goes by on an assembly line. Another part of the program may be monitoring this counter to package the product when the counter reaches a certain value. The only interprocess data exchange technique appropriate for interrupt service routines uses a global data element.

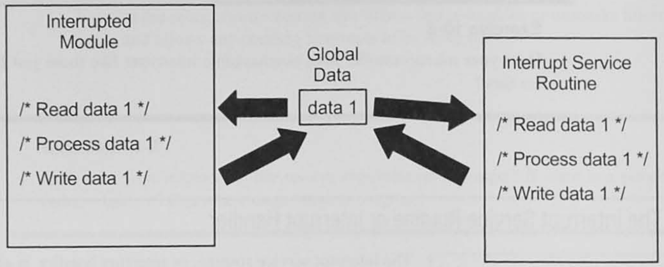


Figure 10-10 Using global data in Information transfer.

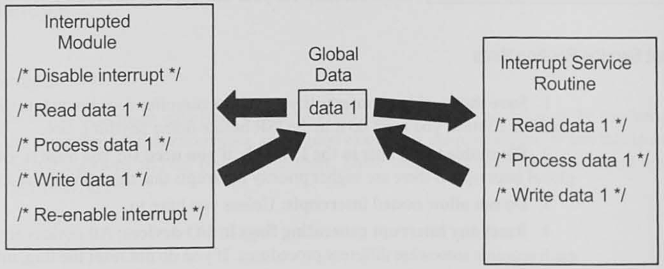


Figure 10-11 Protecting critical code.

Clearly, except for the very simplest programs, registers cannot pass information back and forth. See Figure 10-10.

In addition to the method of data transfer, we must be concerned about the timing of the data exchange. In normal program flow, we have some control over when we write data elements. With interrupts, however, the interrupt can occur at any time, and we must ensure that data are not changed while being used. Consider the situation in Figure 10-10, where both the main program and the interrupt service routine must read, modify, and write the data. There is no problem if the interrupt does not occur while the main program is reading, modifying, and writing the data. If it does occur at these times, the data modification that the ISR produced may be lost. A *critical region* in a program is one in which the interrupted program takes more than one instruction to read, modify, and write data. Figure 10-11 shows that a solution to this problem is to disable the interrupt just before the critical region and re-enable it just after.

10.8 An Interrupt Program Template

All interrupt programs can start with the same basic format. You can use the template given in Example 10-1 for your interrupt programs.

Example 10-1 Interrupt Template

```
/*******************************************************
 * Interrupt Template
 *******************************************************/
/**/
main(){
  /*******************************************************/
  /* 1. Initialize all I/O */
  /*******************************************************/
  /* 2. Clear any flags or interrupt status bits that could
       cause an interrupt */
  /*******************************************************/
  /* 3. Enable the specific interrupt source */
  /*******************************************************/
  /* 4. Enable or unmask global interrupts */
  /*******************************************************/
  /* 5. Do the foreground job */
  for (;;){
  } /* Forever */
}
/*******************************************************/
/* 6. Here is the interrupt service routine. */
/*******************************************************/
void interrupt isr( void ) {
  /*******************************************************/
  /* 6A. Save the machine context if it is not saved
        automatically. */
  /* 6B. Reset the interrupt flag */
  /*******************************************************/
  /* 6C. Enable higher priority interrupts if needed by
        disabling lower priority ones and then re-enabling
        or unmasking global interrupts. */
  /*******************************************************/
  /* 6D. Do the interrupt's specific task */
  /*******************************************************/
  /* 6E. Disable or mask global interrupts */
  /*******************************************************/
  /* 6F. Re-enable lower priority interrupts disabled in 6B. */
/*******************************************************/
  /* 6G. Restore the machine context if not done automatically
        by the return from interrupt */
  /*******************************************************/
  /* 6H. Return from interrupt
  /*******************************************************/
}
```

10.9 Advanced Interrupts

Selecting Edge or Level Triggering

The external interrupt request is normally a low-level-sensitive input. Low sensitivity is suitable for systems with several devices whose interrupt request lines may be tied in a wired-OR configuration, as shown in Figure 10-7. The reason for this is that if more than one device interrupts, each device after the first can keep its interrupt request asserted and thus will be a pending interrupt when another is finished. You may be able to choose to have a negative-edge-sensitive interrupt request. An edge-sensitive interrupt is appropriate only if there is only one interrupt source connected to the interrupt request line.

What to Do While Waiting for an Interrupt

There are three ways to make your microcontroller spin its wheels while waiting for an interrupt to occur. These are *spin loops*, a *wait-for-interrupt* instruction, and a *stop-clocks* instruction.

1. **Spin loop:** The simplest way to make the CPU wait is the spin loop. You use the code shown in Example 10-2 to make the processor branch to itself.

When an interrupt occurs, the CPU will finish executing the instruction, which is a branch to the same instruction. Before the instruction is executed again, the interrupt will be acknowledged and the interrupt service routine executed. The program will fall back into the spin loop when it returns. When the spin loop is used, the interrupt service routines do all the processing.

It is useful to use a spin loop when you are working on the interrupt service routine during the debugging phase of your software development.

Example 10-2 Using a Spin Loop to Wait for an Interrupt

```
/*****************************************************************/
/* The main program does nothing. All processing is done in the
   interrupt service routine */
/* Wait for the interrupt to occur */
  for ( ; ; ){} /* Wait forever */
```

2. **Wait-for-interrupt:** A wait-for-interrupt instruction, if your microcontroller has one, performs two functions. First, it prepares for a subsequent interrupt by saving the machine context.[1] This in turn reduces the delay (the *latency*) in executing the interrupt service routine. This could be important in time-critical applications. Second, it can place the CPU into a reduced power consumption, standby state, which may be important in battery-powered applications.

3. **Stop-clocks:** A stop-clocks instruction operates like the wait-for-interrupt instruction, but it goes even further in reducing the power consumption by stopping all clocks not necessary for the microcontroller's standby operation.

[1] If your processor automatically saves the machine context at the start of an interrupt.

Initializing Unused Interrupt Vectors

Whenever you enable or unmask interrupts, you risk getting interrupts from any interrupting source. To prevent unfortunate things from happening, should an unexpected interrupt occur, always initialize all interrupt vectors to point to a dummy interrupt service routine. This routine should at least reset the flag that caused the interrupt, and it should ensure that the interrupt is indeed disabled. You may wish to add some diagnostics to your software by turning on an LED to indicate that an unexpected interrupt has occurred. See Example 10-3.

Example 10-3 Default Interrupt Service Routine for an Unused Interrupt Source

```
/*****************************************************************
 * This is the default ISR for an unused timer channel.
 *****************************************************************/
/* Reset the timer interrupt flag */
/* Disable the timer interrupt enable bit */
/* Set the interrupt fault LED on if there is one available */
/* Return from interrupt
/*****************************************************************/
```

Exercise 10-4

Does your microcontroller have wait-for-interrupt and stop-clocks instructions?

10.10 Watchdog Timer or Computer Operating Properly (COP)

The *computer operating properly* function is a *watchdog timer*. It can reset the microcontroller if the program gets lost.

A watchdog timer, or COP, is a vital part of computers used in embedded applications. The system must have some way to recover from errors that occur unexpectedly. Power surges or programming errors may cause the program "to get lost" and to lose control of the system. This could be disastrous, and so the watchdog timer is included to help the program recover. When in operation, the program is responsible for pulsing the watchdog at specific intervals. This is accomplished by choosing a place in the program to pulse the watchdog timer regularly. Then, if the program fails to do this, the watchdog

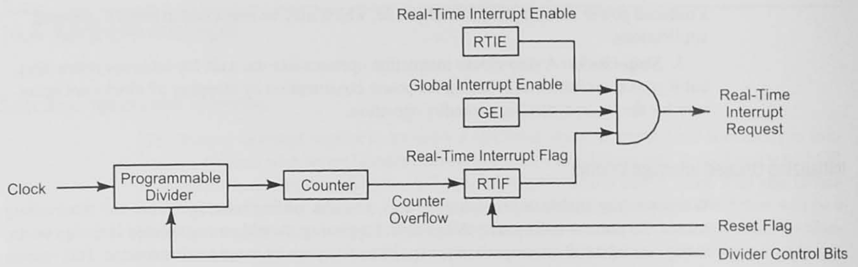


Figure 10-12 Real-time interrupt hardware.

automatically provides a hardware reset to begin the processing again.[2] It is not a good idea to put the watchdog "tickle" in an interrupt routine accessed by a timer interrupt. If the entire program were to be corrupted with the timer interrupt still running, the watchdog would not time out. Place the watchdog timer reset in the main process loop.

10.11 Real-Time Interrupt

The expression "real time" in a microcontroller embedded application generally does not mean "real" time in the sense of the hours, minutes, and seconds of a clock. Instead, it refers to an interval of time whose length is accurately specified and generated by hardware in the microcontroller. In a system using a real-time operating system (RTOS), interrupts are generated at intervals, sometimes called *ticks*, to control the operation of the RTOS.

Figure 10-12 shows a real-time interrupt generator. The clock is divided by a programmable divider, which then drives a counter. When the counter overflows, the real-time interrupt flag is set, and if the global interrupt enable and the real-time interrupt enable bits are set, a real-time interrupt request is generated. The program can control the intervals between interrupts by controlling the programmable divider and sometimes the number of bits in the counter. The interrupt service routine for the RTI must reset the flag.

10.12 Conclusion and Chapter Summary Points

The interrupt capabilities of a processor are important and should be considered very early in your evaluation of a microcontroller for an embedded application. Interrupts can synchronize the operation of the program with real-time events. Interrupts can allow the CPU to continue processing while waiting for I/O devices to become ready for data transfer. Interrupts can also be internally generated when errors or exceptions occur. Most processors also have a software

[2] A particularly good article on watchdog timers can be found at http://www.ganssle.com/watchdogs.htm.

interrupt instruction that is useful for debugging. The important points in this chapter are as follows.

- Interrupts are important asynchronous events that require immediate attention.
- Interrupts are disabled when the CPU is reset.
- Interrupts may be enabled and disabled by the programmer.
- Interrupts are globally enabled/disabled, masked/ unmasked by a control bit.
- Masking means to disallow interrupts, unmasking means allowing them.
- Interrupts may be selectively enabled and disabled by individual bits in control registers.
- The routine that is executed when an interrupt occurs is called an interrupt service routine or interrupt handler.
- A watchdog timer, or COP (for computer operating properly), can reset the CPU if the program misbehaves and runs away.
- There are a variety of interrupting sources, including externally generated ones on I/O ports and internally generated ones such as from the timer subsystem.
- A wait-for-interrupt instruction and a stop-clocks instruction can put the microcontroller into a power-saving mode until an interrupt occurs to wake the processor up.
- Interrupts are asynchronous; they may occur at any time.
- Some interrupt systems use polling and some use vectors to resolve the question of which of many devices has generated an interrupt request.
- Simultaneous interrupt priorities may be resolved by software in polling systems and with hardware in vectored systems.
- Systems with multiple interrupting devices have enable bits to control individually each one.
- The machine context must be saved before you enter the interrupt service routine.
- The machine context must be restored before you return to the interrupted program.
- Interrupts are disabled when the interrupt service routine is entered.
- Interrupt service routines should be kept as simple and short as possible.
- Avoid nested interrupts if possible.
- Use global data elements for interprocess data communications.
- Re-enable interrupts before leaving the interrupt service routine, if the CPU does not do this automatically.
- Disable interrupts in the main program in critical areas.
- "Real time" does not mean hours, minutes and seconds. It means real time intervals.

10.13 Problems

Explore

10.1 List five possible applications for interrupts. [a]

10.2 Describe the actions your microcontroller takes between the time an interrupt request occurs and when the interrupt service routine is entered. [a]

10.3 Why, in most processors with interrupts, are further interrupts disabled when the processor reaches the interrupt service routine? [a, k]

10.4 What is a pending interrupt? [a]

10.5 Name two methods by which a CPU can determine which of several devices has generated an interrupt. [a]

10.6 What are vectored interrupts? [a]

10.7 What are polled interrupts? [a]

10.8 Which type of interrupt, vectored or polled, requires hardware for priority resolution? [a]

10.9 Define "interrupt latency". [a]

10.10 What does interrupt latency depend upon? [a]

10.11 Give at least two components of interrupt latency. [a]

10.12 What is a critical region in a program? [a]

10.13 Does your microcontroller automatically save the machine context when an interrupt occurs, or must that be done in the interrupt service routine by the program?

10.14 Does your microcontroller have a wait-for-interrupt instruction? If so, what does it do (other than wait for the interrupt)?

10.15 Does your microcontroller have a stop-clocks instruction?

Stimulate

10.16 Compare the polling and vectored methods for determining which of many interrupt sources has generated the interrupt request. [a]

10.17 For a processor with 10 interrupting devices, which type of architecture, polled or vectored, provides the faster transfer of control to the interrupt service routine for a specific interrupt? [a]

10.18 What is an advantage of polled interrupts over vectored interrupts? [a]

10.19 What must be done to solve the problem of two devices generating simultaneous interrupts in a system with polled interrupts? [a]

10.20 What must be done to solve the problem of two devices generating simultaneous interrupts in a system with vectored interrupts? [a]

10.21 A real-time interrupt generator as shown in Figure 10-12 is driven by an 8 MHz clock. A programmable divider is followed by a 10-bit counter to generate overflow interrupts. [b, c].

a. How should the programmable divider be set to generate interrupts approximately once every millisecond?

b. How close to 1 ms can you get?

Challenge

10.22 "An interrupt system must allow asynchronous events to interrupt an ongoing process." Give five more hardware and software attributes of an interrupt system. [a]

10.23 Design the hardware for an input interrupting device in a polled interrupt system. Assume an 8-bit switch register for data, a one-bit status register for an "I did it" bit, and a push-button switch to generate a wired-OR IRQ_L signal. The status register and switch register are each to occupy an address in the 8-bit I/O address space. Assume separate I/O with control signals READ_L and WRITE_L. [c]

10.24 Discuss the differences and similarities between a subroutine and an interrupt service routine. [a]

10.25 Write a complete program for your microcontroller for an interrupt occurring on an external interrupt source. When the interrupt occurs, the ISR is to increment an 8-bit memory location "COUNT" starting from 0x00. The foreground job is to be a "spin loop," doing nothing else. [c, k]

10.26 Describe how you could measure interrupt latency in the lab using a lab processor board and other lab instrumentation. [c]

Reflect on Learning

10.27 Summarize what you learned about interrupts in this chapter that you did not know before.

11 Memory

Objectives

This chapter covers the basic principles of memory elements and memory architectures. We explain the different types of memory and discuss the interaction of memory with the CPU.

11.1 Introduction

All computers have both RAM and ROM.

As discussed in Chapter 5, every computer system has two types of memory, RAM and ROM, and the choice of how much and the location in the memory map of each type depends on the computer system being designed. Desktop systems have copious amounts of RAM to load programs into, with little ROM used for the BIOS. On the other hand, an embedded system often has a limited amount of RAM available for variable data storage but significant amounts of ROM for the embedded application's program.

Definitions

Random access: This term applies to memory that can be accessed in any order by supplying it with the address of the memory location and other control signals. Both today's RAM (memory able to be read and written) and ROM (read-only memory) are random access types.

Serial access: In serial access memory, data are stored in sequential locations but must be accessed by starting at the beginning and reading until the required data location has been reached. Disk drive systems, are serial access systems, as were the magnetic tape systems of the olden days.

11.2 A Short History of Random Access Memory

The stored program computer in use today relies on random access memory for program and data storage. This has not always been the case in computer memories. One of the earliest

"computer" memories can be seen in the abacus, which dates back two millennia BC. The Babbage analytical engine of the 1850s was designed to use punched cards based on those developed by Joseph Jacquard in 1801 to control weaving loom. Unfortunately, the Babbage machine was never built; but Herman Hollerith later adopted Jacquard's punched card system to speed up the computation needed for the U.S. census of 1890. The Hollerith punched card remained a staple for program storage until the 1970s, when magnetic tape and disk storage became more affordable.

Many early computers used serial access memories, where a rotating drum storage medium stored the data, similar to the operation of today's hard disk storage system. Another serial access memory was an acoustic delay line, where pulses propagating though a dense medium formed the serial memory storage element. The EDSAC computer, developed in England in the late 1940s, stored 1024 eighteen-bit words in 32 mercury-filled tubes. Another type of serial access memory, based on the operating principle of a storage oscilloscope, was called the Williams-Kilburn tube. This device, developed in the late 1940s in England, could store 500 to 1000 bits. It shared a trait with today's dynamic random access memory in that it needed to be refreshed because the information was an electronic charge stored at a location on the oscilloscope tube.

In 1951 the magnetic core memory, based on work by An Wang at Harvard University in 1949, was used in the Whirlwind computer developed at MIT. The core memory was the first widely successful random access memory, and unlike the semiconductor RAM used in systems today, it was nonvolatile. The memory could retain its data when the power was removed. Figure 11-1 shows two bits of a magnetic core memory. The donut-shaped core, on the order of 1 mm in diameter, had three wires threaded though it. The magnetic flux in the core was set in one direction or the other by current flowing in the X and Y lines. A logic one could be stored if the flux was in one direction, say counterclockwise, and a zero when the flux was clockwise. The bits could be addressed individually, and the currents in the X and Y lines were set so that both X and Y currents were needed to change the direction of flux in the core. To write a bit into a core, the bit had to be addressed, which was accomplished by selecting its X and Y lines. Then the current

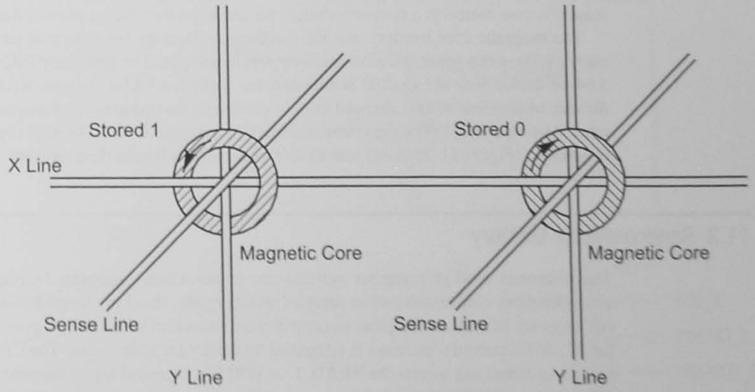


Figure 11-1 Magnetic core memory.

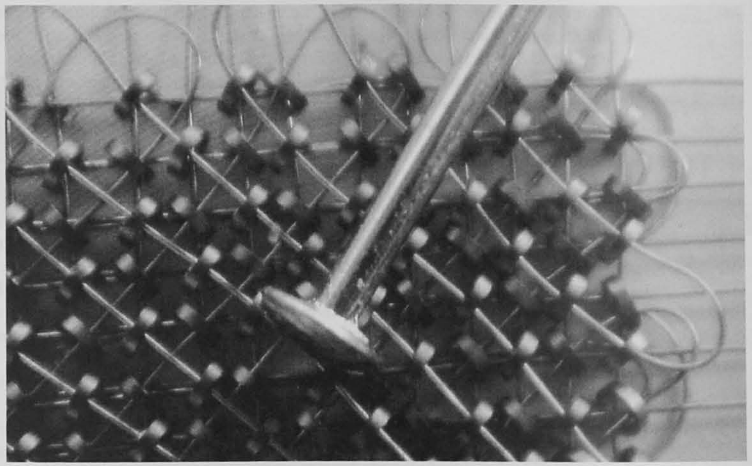


Figure 11-2 Part of a 4096 x 16-bit magnetic core memory.

in the X and Y lines was driven in the direction needed to set the flux in the proper direction for a one or a zero. A read operation was actually a read-write. Again, the bit was addressed and the X and Y lines driven with sufficient current to change the flux direction. If the flux changed, say from the one direction to the zero direction, a current was induced in the sense line that was detected by the memory system. The flux in cores that were in the zero direction did not change and thus induced no current in the sense line. Bits that were changed by the read operation had to be restored, hence the read-followed-by-write operation. Figure 11-2 shows a 16-bit, 4096-word magnetic core memory; a common straight pin indicates the relative sizes of the cores.

The magnetic core memory was the mainstay technology for computer memory until the early 1970s, when semiconductor memory was developed. The first static RAM memory was a 64-bit device done at Fairchild Semiconductor in 1964; a 1 Kbit dynamic RAM with half the die size of previous efforts allowed Intel to challenge the dominance of magnetic core memories starting in 1971. The exponential growth in dynamic RAM (DRAM) capacity since its beginnings (Figure 11-3) shows that Moore's law applies for the development of memory.

11.3 Semiconductor Memory

The memories used in computer systems are semiconductor integrated circuits. A random access memory chip consists of an array of memory cells, decoders for addressing a particular cell or group of cells, and signals to control the direction of data flow (Figure 11-4). Each of the 2^{2N}, M-bit memory locations is addressed by the $2N$-bit address bus. The CPU supplies the required address and asserts the READ_L or WRITE_L control signal for reading or writing. A larger memory can be created by means of the chip enable control signal, CE_L when the memory chip is used with others.

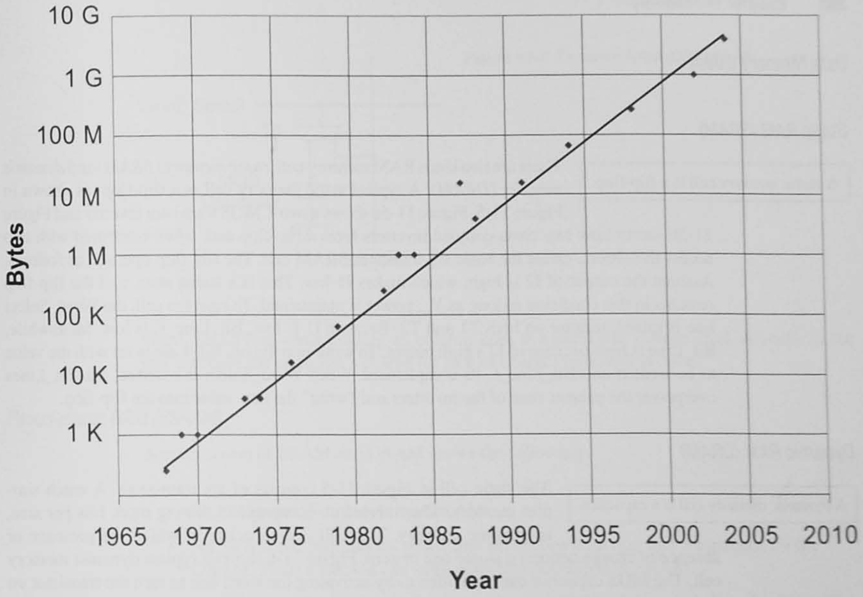


Figure 11-3 DRAM capacity.

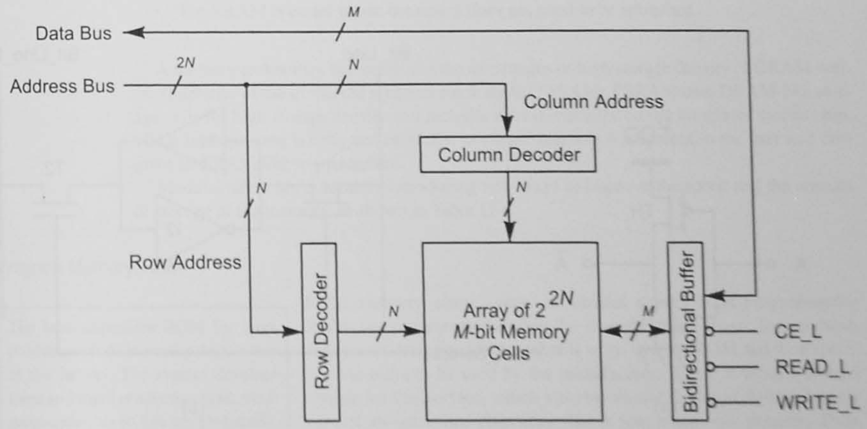


Figure 11-4 Memory chip array.

Data Memory (RAM)

Static RAM (SRAM)

A *static memory* cell is a flip-flop.

There are two kinds RAM memory cell, *static memory* (*SRAM*) and *dynamic memory* (*DRAM*). A typical static memory cell is a flip-flop, as shown in Figure 11-5. Figure 11-5a shows a two-CMOS transistor inverter and Figure 11-5b shows how two cross-coupled inverters form a flip-flop and, when combined with two access transistors, create the basic six-transistor SRAM cell. The flip-flop operates as follows. Assume the output of I2 is high, which makes I1 low. This is a stable state, and the flip-flop remains in this condition as long as V_{DD} power is maintained. To read the cell, the Word_Select line is raised, turning on both T1 and T2. Because I1 is low, Bit_Line_L is low. Meanwhile, Bit_Line is high because of I2's high output. To write into the bit, Bit_Line is set with the value to be written and Bit_Line_L its complement. When Word_Select is asserted, the Bit_Lines overpower the present state of the inverters and "write" the new value into the flip-flop.

Dynamic RAM (DRAM)

A *dynamic* memory cell is a capacitor.

The static cell in Figure 11-5 consists of six transistors. A much simpler memory, which therefore is capable of storing more bits per area, is *dynamic memory*. This cell is a capacitor in which the presence or absence of charge denotes a stored one or zero. Figure 11-6 shows a typical dynamic memory cell. The MOS capacitor can be written to by activating the word line to turn the transistor on and charge the MOS capacitor through the bit line. Turning the transistor on and sensing a voltage on the Bit_Line reads the cell.

A problem with dynamic memory is that the charge stored on the capacitor leaks away to the substrate. Thus, dynamic memory must be refreshed at periodic intervals by activating the Word_Select line while holding all column lines at a particular voltage level. All cells in the row can have the capacitor's charge (or lack of charge) refreshed at once.

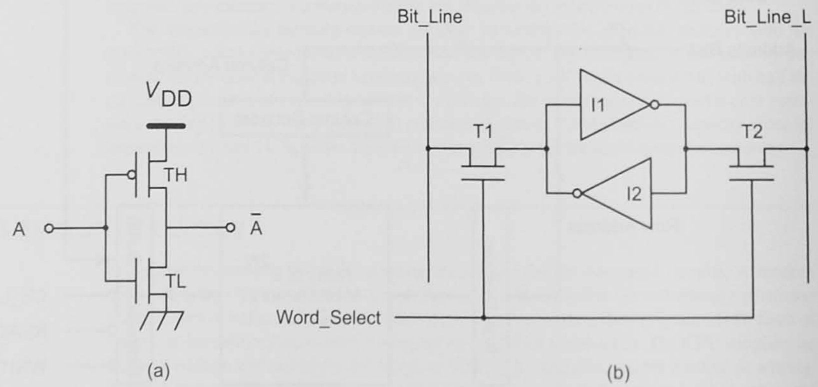


Figure 11-5 Static RAM (SRAM) cell. (a) CMOS inverter. (b) Cross-coupled inverters.

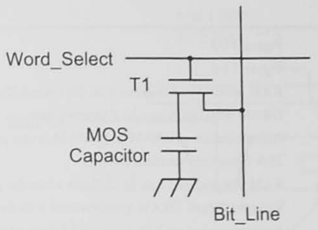


Figure 11-6 Dynamic RAM (DRAM) cell.

Pseudostatic RAM (PSRAM)

A comparison of SRAM and DRAM shows the following:

- SRAM generally is faster than DRAM.
- An SRAM cell requires six transistors versus one transistor and a capacitor for the DRAM. Thus, a DRAM can store more bits per chip than SRAM.
- Because the DRAM storage element is a capacitor, it requires periodic refreshing. The SRAM storage cell is a flip-flop, which does not need refreshing.
- If a DRAM is refreshing, the CPU may have to wait until the process is complete before accessing a storage site.
- The SRAM is easier to use because it does not need to be refreshed.

A memory technology that combines the advantages of high storage density of DRAM with the simplicity of use of SRAM is *pseudostatic RAM* (*PSRAM*). PSRAM uses DRAM-like storage cells for high storage density and includes refresh circuitry on the integrated circuit chip. The refresh process is designed carefully, to ensure that it is transparent to the user and thus gives an SRAM-like user interface.

Manufacturers are constantly introducing new ways to improve the speed and the amount of storage of the memory, as shown in Table 11-1.

Program Memory: ROM

The least expensive ROM for large production runs is *mask programmed* at the factory. For system development and small production runs, *field-programmable* ROMs are preferred.

ROM memory chips come in various types. *Mask-programmable* ROMs are programmed during the manufacturing stage. To use these, the system designer decides what is to go into the ROM and then specifies the *mask* to be used by the manufacturer. There is usually a *mask charge* for this service, which may be several thousand dollars; but the cost of an individual chip after that is low, often only pennies. Thus, mask-programmed devices are suitable for high-volume applications.

Table 11-1 RAM Memory Types

SRAM	Static RAM	Figure 11-5
DRAM	Dynamic RAM	Figure 11-6
BSRAM	Burst or SynchBurst static RAM	RAM access synchronized with the system clock to speed up access
FPM DRAM	Fast page mode DRAM	DRAM with fast access to a memory row
EDRAM	Enhanced DRAM	A combination of SRAM and DRAM in one package
EDO RAM	Extended data output DRAM	25% faster than standard DRAM
NVRAM	Nonvolatile RAM	RAM that does not lose its contents when the power is turned off
SDRAM	Synchronous DRAM	Various types of DRAM synchronized with the processor clock
DDR SDRAM	Double data rate SDRAM	Activates output on both rising and falling edges of the clock
ESDRAM	Enhanced SDRAM	A combination of SRAM and SDRAM
PSRAM	Pseudostatic RAM	DRAM cells with on-chip refresh circuitry

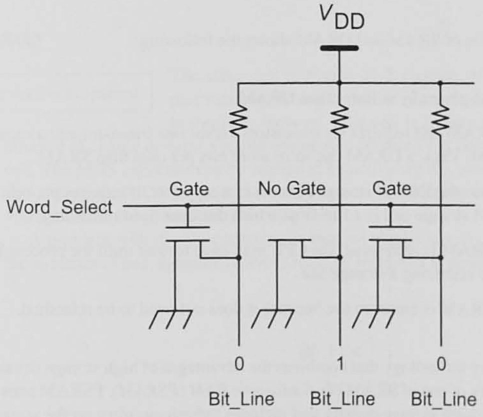


Figure 11-7 ROM cells.

Figure 11-7 shows a mask-programmed ROM. A bit is either 1 or 0, depending on whether the transistor gate is or is not integrated.

EPROM

Field-programmable ROMs are used for system development and in low-volume applications.

Other ROM devices are *field programmable* and may be programmed by the user. These so-called *programmable read-only memories* include *UV-erasable PROMs* (*EPROMs*), and *one-time-programmable* (*OTP*) EPROMs (Figure 11-8). The first EPROM was the Intel 1702, introduced in 1971. This memory was a 256 word, 8-bit chip and could be

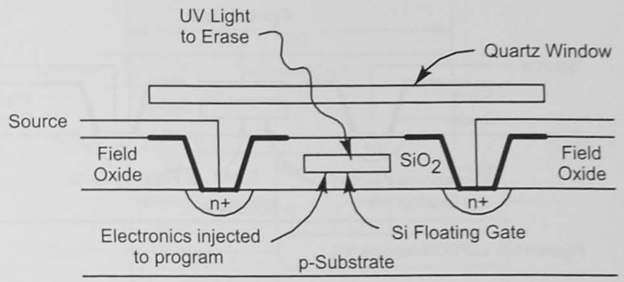


Figure 11-8 EPROM storage cell.

programmed, erased, and reprogrammed during the development cycle. These characteristics greatly speeded up the development process and reduced the cost because the part did not have to be thrown away if the program had to be changed. EPROMs are electrically programmable and erased by irradiating the chip through a quartz window with ultraviolet light. The cell in an EPROM is a MOS transistor without a connection to the gate. This is called a *floating-gate, avalanche-injection, charge storage device*. Figure 11-8 shows a model. To program the EPROM, the silicon chip is placed into a *PROM programmer*, and during the programming cycle, the address and data are sent to the chip and the programming voltage is applied. To change the state of the gate, electrons either are or are not injected by an avalanche mechanism into the silicon floating gate . Thus, after programming, the channel between the source and the drain either conducts or does not. If the chip needs to be erased, it must be removed from its circuit and placed into a *PROM eraser,* where it is irradiated with UV light at a wavelength less than 400 nm (0.4 μm). This disperses back into the substrate any charge stored in the floating gate and erases the memory. Sunlight and fluorescent lamps of some types contain energy in this wavelength region, and manufacturers caution users that an EPROM can become erased by direct exposure to the sun for one week and by exposure to fluorescent lamps for 3 years. In applications where this danger exists, you should place an opaque cover over the quartz window. An OTP EPROM is an EPROM without the window; this means that once programmed, the memory cannot be erased.

EEPROM and Flash Memory

Figure 11-9 shows an *electrically erasable PROM* (*EEPROM*). Note its similarity to Figure 11-8. A second polysilicon gate, called the control gate, is added above the floating gate. A control voltage may be applied to this gate to program and erase the cell by injecting or dispersing electrons in the floating gate.

A *Flash* memory chip is similar to the EEPROM. Although it can be programmed faster than the standard EEPROM (hence the name Flash), it has the drawback that the entire memory or a block of memory must be erased, where as single locations can be erased and reprogrammed in the EEPROM devices.

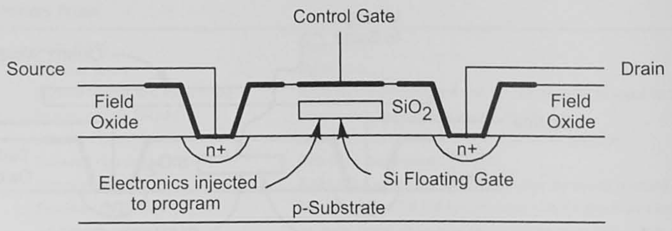


Figure 11-9 EEPROM storage cell.

11.4 Memory Timing Requirements

Remember from Chapter 2 that the CPU is controlling the information transfer in the system. It generates the control signals, such as READ_L and WRITE_L, and takes data from or puts data onto the bus at specific times, as shown Figures 2-17 and 2-18. The CPU clock controls the overall timing.

Let us now look at the memory system timing from the point of view of the memory. It is easiest to start the discussion with the timing of a static memory chip. Figure 11-10a shows typical read cycle timing diagrams for static RAM and defines the basic times listed in the following sections on memory read and write cycles.

The Memory Read Cycle

t_{RC}, **read cycle:** This is the total time for the read cycle.

t_{ACS}, **chip select access:** The maximum time required by the memory for the CS_L to be asserted before the data are available.

t_{AA}, **address access:** This is the maximum time required by the memory for the address to be present before the data are available.

t_{RDHA}, **read data hold after address:** The time the memory may hold the data at the output after the address is changed.

t_{RDHC}, **read data hold after chip select:** The minimum time the chip will hold the data after being deselected.

t_{OE}, **output enable access:** On chips that have an output enable, this parameter gives the maximum time for the chip to respond with the data.

t_{OHZ}, **output enable to output high Z:** On chips that have an output enable, this parameter specifies how long the data will remain valid before going into three-state (high impedance, Z).

Two times for reading data are important to memory system designers. The read cycle time, t_{RC}, is the minimum time that the addresses must be stable (unchanging) at the chip. The address access time, t_{AA}, is the maximum time required by the memory before the data are available. Although most manufacturers draw the timing diagrams showing t_{RC} and t_{AA} looking different, they are usually the same.

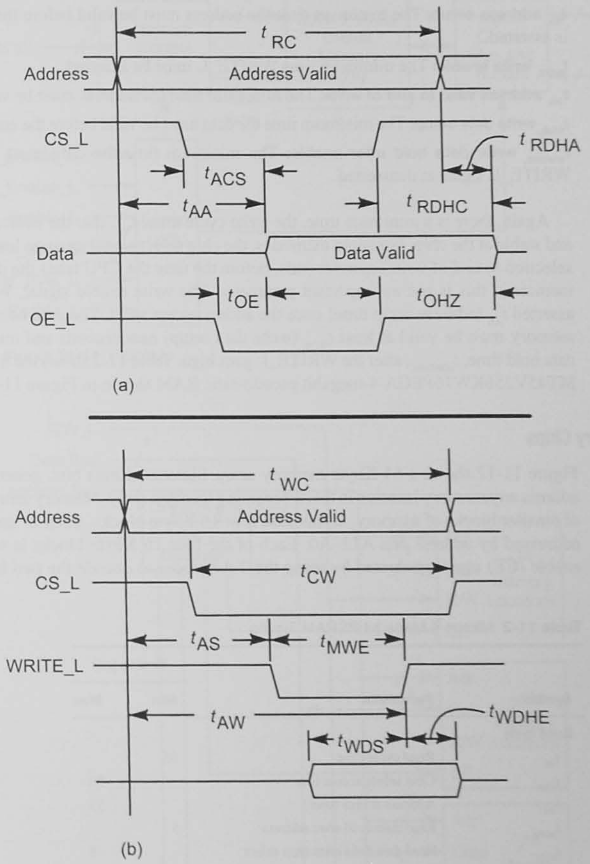


Figure 11-10 (a) Memory read cycle. (b) Memory write cycle.

The Memory Write Cycle

The memory write cycle timing diagram is shown in Figure 11-10b, and we can define the following times:

t_{WC}, **write cycle:** This is the minimum total time required by the memory to complete a write cycle. This may or may not be the same as the read cycle time t_{RC}.

t_{CW}, **chip selection to end of write:** The minimum time the CS_L signal must be asserted.

t_{AS}, **address setup:** The minimum time the address must be valid before the WRITE_L signal is asserted.

t_{MWE}, **write enable:** The minimum time WRITE_L must be asserted.

t_{AW}, **address valid to end of write:** The minimum time the address must be valid.

t_{WDS}, **write data setup:** The minimum time the data must be valid before the end of write enable.

t_{MWDHE}, **write data hold after enable:** The minimum time the data must be valid after the WRITE_L signal is deasserted.

Again, there is a minimum time, the write cycle time, t_{WC}, that the address must be present and stable at the chip. For some memories, the chip select signal must go low, at least t_{CW} (chip selection to end of write) nanoseconds, before the time the CPU takes the data away. In other memories, this is not an important parameter. The write enable signal, WRITE_L, may be asserted t_{AS} (address setup time) once the addresses are valid. The data being written into the memory must be valid at least t_{WDS} (write data setup) nanoseconds and must be held for the data hold time, t_{MWDHE}, after the WRITE_L goes high. Table 11-2 shows the timing for a Micron MT45V256KW16PEGA 4-megabit pseudostatic RAM shown in Figure 11-11.

Arrays of Memory Chips

Figure 11-12 shows a 64 Kbyte memory array. Sixteen address bits, generated by the CPU, address any memory location in this 2^{16} memory location array. Memory arrays are constructed of smaller blocks of memory, in this case four 16 Kbyte blocks. Each 2^{14} memory location is addressed by address bits A13–A0. Each of the four 16 Kbyte blocks is selected by a chip enable (CE) signal generated by using the 2-4 decoder to decode the two highest significant

Table 11-2 Micron 4 Mega bit PSRAM Timing

		Timing (ns)	
Symbol	**Parameter**	**Min**	**Max**
Read cycle			
t_{RC}	Read cycle time	55	
t_{ACS}	Chip select access time		55
t_{AA}	Address access time		55
t_{RDHA}	Read data hold after address	5	
t_{RDHC}	Read data hold after chip select		8
t_{OE}	Output enable access time		20
t_{OHZ}	Output enable to output high Z		8
Write cycle			
t_{WC}	Write cycle time	55	
t_{CW}	Chip selection to end of write	45	
t_{AS}	Address setup time	0	
t_{MWE}	Write enable width	35	
t_{AW}	Address valid to end of write	45	
t_{WDS}	Write data setup time	23	
t_{WDHE}	Write data hold time	0	

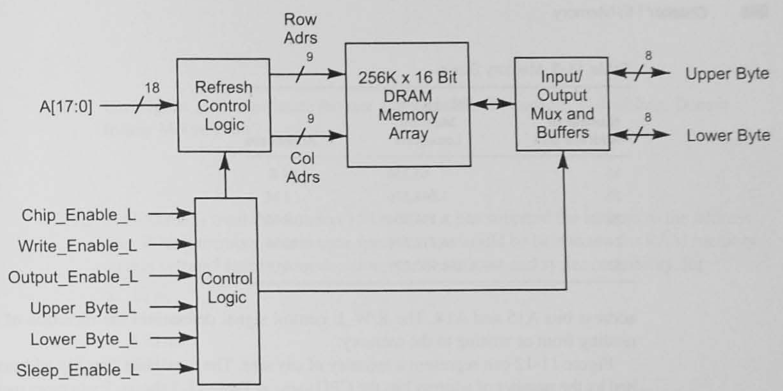


Figure 11-11 PSRAM.

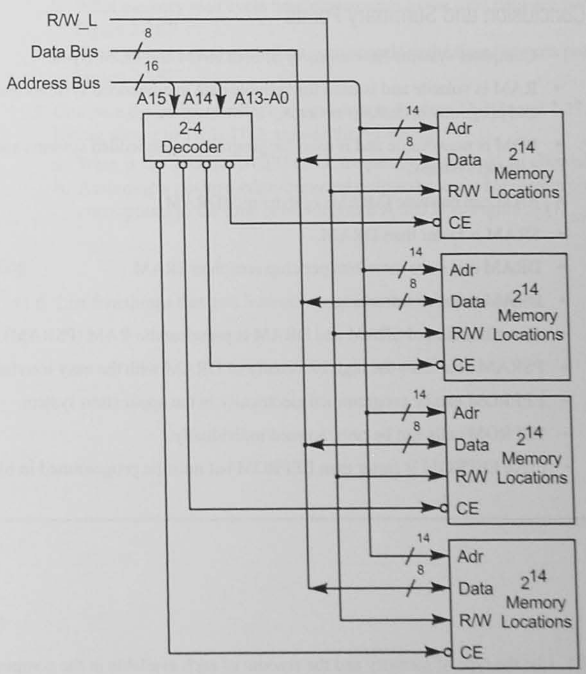


Figure 11-12 64 Kbyte memory.

Table 11-3 Memory Sizes

Number of Address Bits	Number of Memory Locations	Addresses
16	65,536	64 K
20	1,048,576	1 M
22	4,194,304	4 M
24	16,777,216	16 M
32	4,294,967,296	4 G

address bits A15 and A14. The R/W_L control signal determines the direction of data flow, reading from or writing to the memory.

Figure 11-12 can represent a memory of any size. The maximum directly addressed is limited by the number of address bits the CPU uses, as Table 11-3 shows. Even more memory than this can be addressed in processors with expansion memory, as discussed in Chapter 4.

11.5 Chapter Conclusion and Summary Points

- Computer systems have memory of both RAM and ROM types.
- RAM is volatile and is used for variable data in embedded systems and variable data and programs in desktop systems.
- ROM is nonvolatile and is used for programs in embedded systems and the BIOS in desktop systems.
- RAM can be static (SRAM) or dynamic (DRAM).
- SRAM is faster than DRAM.
- DRAM can store more bits per chip area than SRAM.
- DRAM requires refreshing.
- A combination of SRAM and DRAM is pseudostatic RAM (PSRAM).
- PSRAM combines the high bit density of DRAM with the easy interface of SRAM.
- EEPROM can be programmed electrically in the application system.
- EEPROM cells can be programmed individually.
- Flash EEPROM is faster than EEPROM but must be programmed in blocks.

11.6 Problems

Explore

11.1 List the type of memory and the amount of each available in the computer system you are studying.

Stimulate

11.2 Use Figure 11-3 to estimate the rate at which DRAM capacity is doubling. Does it follow Moore's law?

Challenge

11.3 A CPU reads from the data bus 150 ns after it has supplied the address to the address bus. Which memory access time specification would be best to use for RAM memory in this system? Justify your decision in terms of cost and system reliability. [c]

a. 10 ns
b. 110 ns
c. 150 ns
d. 200 ns

11.4 Compare the memory read cycles shown in Figures 11-10a and 2-18. For the memory timing shown in Table 11-2, answer the following.

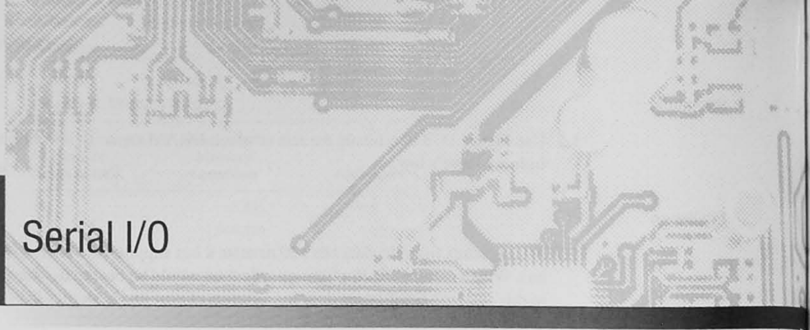
a. What is the maximum CPU clock frequency that would be allowed?
b. What memory read cycle time corresponds to the time between points A and C in Figure 2-18?
c. What memory read cycle time corresponds to the time between points B and C in Figure 2-18?

11.5 Compare the memory write cycles shown in Figures 11-10b and 2-17. For the memory timing shown in Table 11-2, answer the following.

a. What is the maximum CPU clock frequency that would be allowed?
b. Assuming a positive-edge-triggered output device, what memory write cycle time corresponds to the time between points A and D in Figure 2-17?

Reflect on Learning

11.6 List five things that you learned about memories in this chapter.



12 Serial I/O

Objectives

In this chapter we dispel the mysteries and myths of the asynchronous serial interface. Nearly everybody who has connected a serial device to a computer has had trouble of some kind. In the personal computer world, the serial interface is called the *com* port, and many PCs have one or more of these, although the universal serial bus (USB) is taking over many of the jobs the serial com port used to do in desktop computers. Nonetheless, in embedded applications, asynchronous serial I/O is a useful method of transporting data over long distances using only three wires (at a minimum). We will see that interfacing serial devices is not difficult once we understand the basics of serial data transmission and how to use the handshaking signals defined for the RS-232-C interface. In this chapter we will also describe the synchronous serial peripheral interface (SPI), and, briefly, the inter-integrated circuit (I²C) and controller area network (CAN) buses.

12.1 Introduction

Chapter 9 discussed parallel I/O interfaces to input and output data. A disadvantage of parallel I/O is that a wire is needed for each bit, and a parallel cable can be bulky and expensive when source and destination are more than a few feet apart. In addition, long runs of parallel wires can act as a transmission line that is susceptible to reflections and induced noise. Serial I/O techniques can offer a solution to these problems. Data are sent one bit at a time, using fewer wires. By defining appropriate standards for the logic levels, we can both reduce the effects of long transmission lines and combat noise problems.

12.2 The Asynchronous Serial Communication System

Figure 12-1 shows a serial communication system connecting two microcontrollers. In many microcontrollers the serial interface is called the *serial communications interface*, or *SCI*. The interface's job is to convert the parallel data transfer within each microcontroller to a serial data transfer between them. Handshaking signals are defined for the serial interface operation

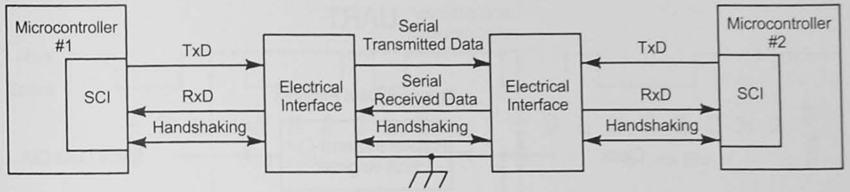


Figure 12-1 Serial communication system.

to accomplish the I/O synchronization described in Chapter, Section 9.7. An electrical interface is required to convert the CMOS or TTL logic levels of the microcontroller to other signal levels more suited to the external environment. The design of this system must consider the following questions:

- How do you encode the data?
- If the data are sent in serial, which bit is sent first?
- How is the receiver synchronized with the transmitter?
- What is the data rate?
- How are the electrical signals for logic values defined?
- How does the system provide for handshaking?

The Serial Communications Interface (SCI)

A *UART* is a parallel-to-serial plus a serial-to-parallel data converter.

The serial interface in the microcontroller is called a *universal asynchronous receiver/transmitter*, or *UART* (Figure 12-2). The microcontroller sends data through its internal parallel I/O interface to the *transmit data buffer*. These data are transferred to the *parallel in/serial out shift register*, and the clock shifts the data out on the *transmitted data (TxD)* signal line. Serial data bits are received on the *received data (RxD)* signal line and shifted into the *serial in/parallel out shift register*. After all data bits have been shifted, they are transferred to a *received data buffer*, where the microcontroller can use an input operation to read them. Although you can buy UARTs as individual chips, these days most microcontrollers have them as an integral part. Besides the data bus and clock signals shown in Figure 12-2, there are other signals for handshaking and control, such as *transmitted data register empty* and *received data register full*. We discuss the need for these in the next sections.

Data Coding and Transmission

Any data code can be used for serial data transfer.

Any binary code that both ends agree upon can be used. Serial data transfer is frequently used to send data between a terminal and a computer. In this case, the information is the alphanumeric key pressed on the

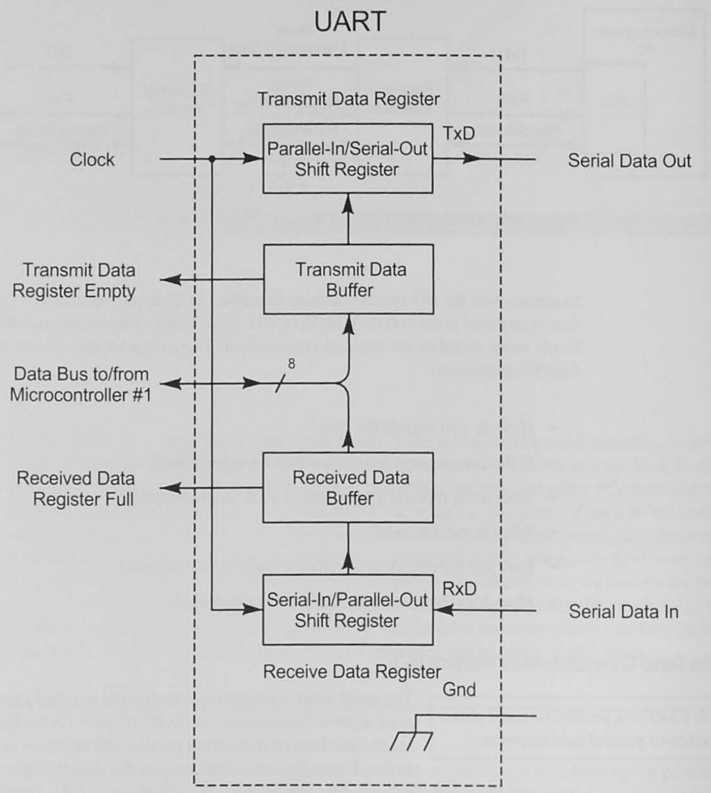

UART
Transmit Data Register
Clock
Parallel-In/Serial-Out Shift Register
TxD
Serial Data Out
Transmit Data Register Empty
Transmit Data Buffer
Data Bus to/from Microcontroller #1
8
Received Data Register Full
Received Data Buffer
Serial-In/Parallel-Out Shift Register
RxD
Serial Data In
Receive Data Register
Gnd

Figure 12-2 Serial communication UART.

keyboard or the character displayed on the screen. Of the several codes used for alphanumeric information, the most common in microcomputer work is the *American Standard Code for Information Interchange*, or *ASCII*. The ASCII code, shown in Section 12.5, uses 7 bits to encode 96 printable characters and 32 control characters.

Serial data bits are synchronized at the receiver by first sending a *start bit*, then the data, and then a *stop bit.*

We have two choices for the order of data transmission. The designers of the UARTs have chosen to send the least significant bit first. Sending characters in this way is called *asynchronous* serial communications because the characters can be sent at any time and do not need to be synchronized with any process in either the sending or receiving unit. For example, characters typed on a keyboard are sent when you type them. The designers provided

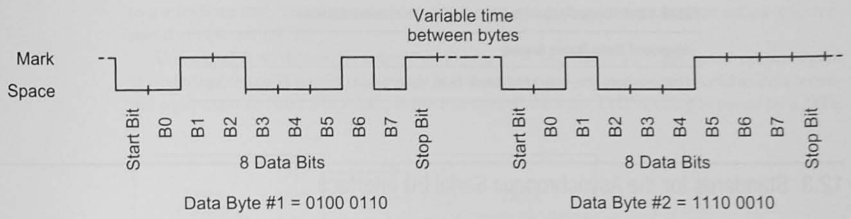


Figure 12-3 Asynchronous serial character transmission.

a way to synchronize the receiver shift register with the transmitter shift register to cope with the asynchronous transmissions. Two other bits, known as the *start bit* and the *stop bit*, encapsulate the data bits. Figure 12-3 shows the format of the data and several terms used in serial data communications. Here are the basic definitions.

1. Mark and space: The logic one and zero levels are called *mark* and *space*. When the transmitter is not sending anything, it holds the line at the mark level (i.e., logic one). This is also called the *idle* level.
2. Start bit: When the transmitter has data to send, it first changes the line from the mark to the space level for one bit time. This synchronizes the receiver with the transmitter. When the receiver detects the start bit, it knows to start clocking in the serial data bits.
3. Data bits: Almost any number of data bits can be sent between the start and stop bits, depending on the length of the transmit and receive shift registers. Typically, eight or nine are used.
4. Parity bit: Only 7 bits are needed to encode ASCII characters. Most UARTs allow up to 8 (and sometimes 9) bits to be sent between the start and stop bits, and so a parity bit may be included. The parity bit is added to the data to make the total number of ones odd (odd parity) or even (even parity). The parity bit may be used to detect errors in the data. A parity bit is used frequently when 7-bit ASCII codes are being transferred.
5. Stop bit: The stop bit is added at the end of the data bits. This gives at least one-bit time between successive characters. Some systems require more than one stop bit.

Data Transmission Rate

The *baud rate* is the number of bits per second.

The rate at which bits are sent is often called the *baud rate*. This is a misused term because a *baud* is a unit of signaling speed and signifies the number of times per second the state of the line is changed. It is the reciprocal of the length of the shortest element in the code and is given in bits per second. Baud is a contraction of the surname of an early pioneer in serial data communications, J. M. E. Baudot.[1] The data rate can be any value, and standard data rates are shown in Table 12-1.

[1] J. M. E. Baudot (1845–1903) invented a 5-bit code for sending data in a telegraph system. It was adopted by the French telegraph system in 1877 and became one of the standards used for international telegraph communication.

Table 12-1 Data Rates Used in Serial Communications

Standard Data Rates (baud)
110, 150, 300, 600, 900, 1200, 2400, 4800, 9600, 14,400, 19,200, 38,400, 57,800

12.3 Standards for the Asynchronous Serial I/O Interface

Several standards have been developed to define the interface between two SCIs in a serial communication system. In interface standards, which are necessary to allow different manufacturers' equipment to be interconnected, the following elements must be defined:

- Handshaking signals
- Direction of signal flow
- Types of communication devices
- Connectors and interface mechanical considerations
- Electrical signal levels

The RS-232-C standard of the Electronic Industries Association[2] is used in most asynchronous serial interfaces. For signals that must be transmitted farther than 50 feet or at greater than 20 Kbit/s, however, another electrical interface standard, such as RS-422, RS-423, or RS-485, should be chosen. For each of these, handshaking, direction of signal flow, and types of communication device are based on the RS-232-C standard.

Handshaking Signals

Serial data transfer requires handshaking signals for synchronization and control of the transmitter and receiver. All signals in the RS-232-C interface other than the transmitted and received data are for handshaking. To understand these, we must first look at communication system types and at modems.

Data Terminal Equipment and Data Communication Equipment

The EIA standard defines two kinds of device serving as the electrical interface shown in Figure 12-1. Modems[3], also called *data communications equipment* (*DCE*), connect the SCI to a telephone line. The terminals or computers to which they are attached are called *data terminal equipment* (*DTE*).

The signal flow directions defined in Figure 12-4 and Table 12-2 are based on the signal flow defined for a DTE device. You will find that computers often are configured as data terminal equipment devices. For example the transmitted data pin TxD is being *sourced* by a DTE

[2] The Electronic Industries Association (EIA) publishes engineering standards to serve the public interest by eliminating misunderstandings between manufacturers and purchasers. EIA standards can be purchased from the organization:
EIA Engineering Department
Standards Sales
2001 I Street, NW
Washington, DC 20006
(200) 457–4966

[3] A modem, or "MOdulator/DEModulator," converts binary signals (logic levels) to and from the tones sent over the telephone line.

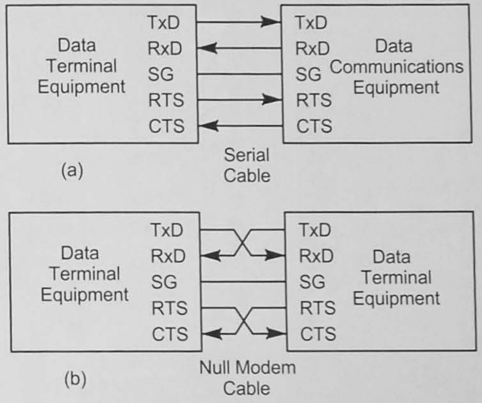


Figure 12-4 Serial communications. (a) DTE-DCE; (b) DTE-DTE.

Table 12-2 RS-232-C Signal Definitions

DE9	DB25	Signal	Purpose
	1	PG	*Protective ground.* This is usually the shield in a shielded cable. It is designed to be connected to the equipment frame and may be connected to external grounds.
3	2	TxD	*Transmitted data.* **Sourced** by DTE and **received** by DCE. Data terminal equipment cannot send unless RTS, CTS, DSR, and DTR are asserted.
2	3	RxD	*Received data.* **Received** by the DTE, **sourced** by DCE.
7	4	RTS	*Request to send.* **Sourced** by DTE, **received** by DCE. RTS is asserted by the DTE when it wants to send data. The DCE responds by asserting CTS.
8	5	CTS	*Clear to send.* **Sourced** by DCE, **received** by DTE. CTS must be asserted before the DTE can transmit data.
6	6	DSR	*Data set ready.* **Sourced** by DCE, **received** by DTE. Indicates that the DCE has made a connection on the telephone line and is ready to receive data from the terminal. The DTE must see this asserted before it can transmit data.
5	7	SG	*Signal ground.* Ground reference for the signal is separate from pin 1, protective ground.
1	8	DCD	*Data carrier detect.* **Sourced** by DCE, **received** by DTE. Indicates that a DCE has detected the carrier on the telephone line. Originally it was used in half-duplex systems but can be used in full-duplex systems too.
4	20	DTR	*Data terminal ready.* **Sourced** by DTE, **received** by DCE. Indicates the DTE is ready for sending or receiving.
9	22	RI	*Ring indicator.* **Sourced** by DCE, **received** by DTE. Indicates that a ringing signal is detected.

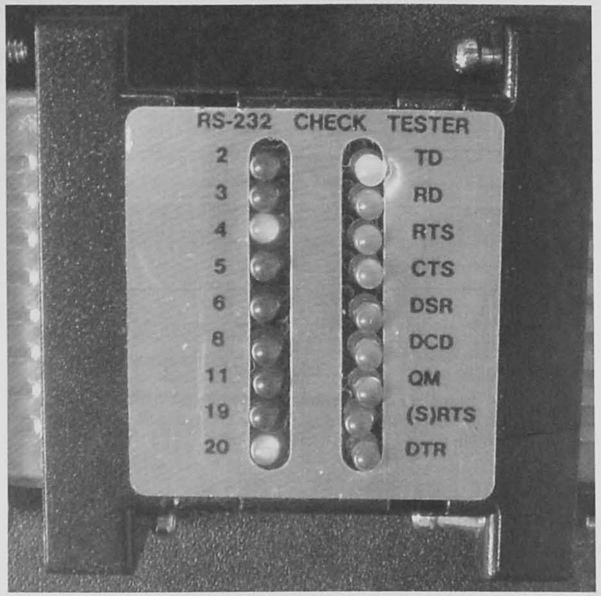


Figure 12-5 RS-232-C tester.

device. Data communication equipment (DCE) devices include modems and some printers. The signal flow and signal names used for DCE devices are often incorrectly specified. For example, the TxD signal is actually *received* by the DCE device. It is incorrect to call this signal RxD. When connecting one device to another, we must be sure what kinds of device are being used, and we must select the proper cable.

A very useful tool to have when working with RS-232-C interface devices is the RS-232-C tester (Figure 12-5). This tester shows what serial lines are active and allows us to determine easily if we are connecting to a DTE or DCE device.

Modem Handshaking Signals

The principal signals used in modem handshaking are as follows.

1. **Ring indicator (RI):** The telephone company transmits a special tone that rings the phone. The modem can detect this and assert the RI signal. The terminal or computer can use RI to start some special process, such as notifying the user that the other end is calling or to answer the telephone in an answer modem.

2. **Data set ready (DSR):** This signal tells the DTE that the modem (also called a *data set*) has established a connection over the telephone line to the far end.

3. **Data terminal ready (DTR):** This signal from the DTE informs the modem that it is ready to operate. This is usually just an indication that the power is turned on in the terminal, but the signal could be controlled by a computer. An intelligent answer modem can use it to answer a call automatically only when the computer or terminal is ready.

4. **Data carrier detect (DCD):** The DCD signal is asserted when the carrier (the tone defined for a mark) is being generated by the modem on the other end. DCD was used originally in systems where data could be sent in only one direction at a time; these are called half-duplex systems. When one end wanted to transmit, it first asserted the RTS line. The modem then checked the DCD bit. If it found it asserted, it knew the other end was sending. When DCD was deasserted, CTS was asserted allowing transmission from the requesting terminal.

The complete RS-232-C standard defines all signals and signal directions for DTE and DCE devices. There are three schemes for labeling the signals: mnemonic acronyms, alphabetic circuit codes, and CCITT (International Telegraph and Telephone Consultative Committee[4]) numeric codes. The most descriptive and most frequently used are the signal acronyms listed in Table 12-2. Also shown are the RS-232-C standard pin numbers for the DB25 connector and the pins that have been defined for the DE9 connector used on IBM personal computers and compatibles. The signals given in Table 12.2 are the main ones used in serial interfaces. The RS-232-C standard also defines another set of signals that are used for secondary data transmission. These are very rarely used.

12.4 Asynchronous Serial Hardware Interfaces

RS-232-C Interconnections

A *null modem cable* is used to connect two DTE computers together.

When two serial ports are connected, the data rate, the number of data bits, whether parity is used, the type of parity, and the number of stop bits must be set properly and identically on each UART. You must also have the proper cables; depending on the devices to be interconnected, there are four kinds of cable from which to choose. These are the full DTE-DCE cable (Figure 12-6), a DTE-DTE *null modem* cable (Figure 12-7), and a minimal DTE-DCE cable that works in many applications (Figure 12-8). A minimal null modem cable for DTE-DTE connections also may be constructed (Figure 12-9).

When first encountering the RS-232-C interface, many users have trouble reconciling the *direction* of data flow with the *signal name*. Look at the directions shown for the signals on the DTE device in Figure 12-6. Notice that pin 2, transmit data (TxD), is a data output. On the other side, TxD for a DCE device is an input! Unfortunately, many manufacturers do not provide enough details in their documentation to let us know if a signal is an input or output. You cannot tell by the name alone. You must also know if you have a DTE or DCE device. To provide the proper cable, you may have to resort to inspecting the schematic diagram, measuring voltages, or using the RS-232-C tester shown in Figure 12-5 to find out which pin is an output.

[4] This organization is known by its French acronym, CCITT.

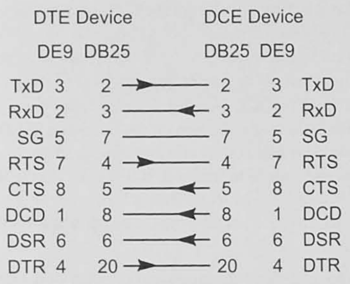


Figure 12-6 Full DTE-DCE cable (straight serial cable).

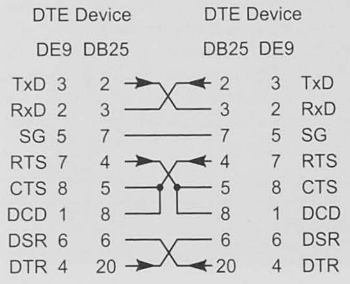


Figure 12-7 DTE–DTE null modem cable.

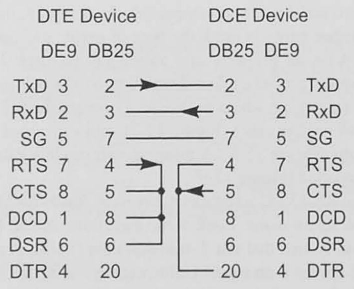


Figure 12-8 Minimal three-wire serial cable.

Standard Electrical Signal Levels

RS-232-C Standard

The signal levels for RS-232-C mark and space are shown in Table 12-3. Notice that the signal level for a mark is low.

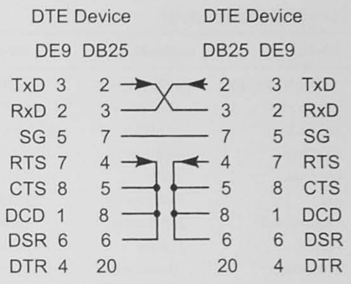


Figure 12-9 Minimal null modem cable.

Table 12-3 RS-232-C Logic Levels

RS-232-C Signal	Voltage	Logic State	Logic Level
Mark	–25 to –3 V	1	Low
Space	+3 to +25 V	0	High

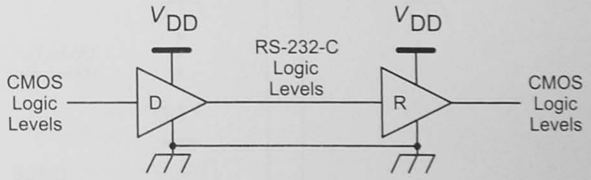


Figure 12-10 RS-232-C interface.

RS-232-C signal levels have been defined to give a large noise margin.

The RS-232-C interface driver (D) and receiver (R) pair is shown in Figure 12-10. The driver and receiver are called *single ended* because the signal line is referenced to the ground. The driver and receiver convert CMOS or TTL logic levels to the RS-232-C levels, which provide much greater noise margin. RS-232-C drivers can be used effectively if the distance does not exceed 50 feet and the data rate is not higher than 20 Kbit/s. As the line distances get longer or the data rate higher, another signaling standard should be chosen. The electrical characteristics of the RS-232-C standard will be given shortly, in Table 12-4.

RS-423 Standard

The RS-423 interface can transmit at higher data rates and over longer distances than RS-232-C.

The RS-423 interface is shown in Figure 12-11. It, too, is a single-ended system, but the drivers are especially matched and tuned to one another to allow the longer distances and higher data rates shown later in Table 12-4, along with the electrical specifications for RS-423 signaling. RS-423 also allows a driver to broadcast data to 10 receivers.

Table 12-4 Summary of RS-232-C, RS-423, RS-422, and RS-485 Standards

Specification	RS-232-C	RS-423	RS-422	RS-485
Receiver input voltage	±3 to ±15 V	±200 mV to ±12 V	±200 mV to ±7 V	±200 mV to −7 to +12 V
Driver output signal	±5 to ±15 V	±3.6 to ±6 V	±2 to ±5 V	±1.5 to ±5 V
Maximum data rate	20 Kbit/s	100 Kbit/s	10 Mbit/s	10 Mbit/s
Maximum cable length	50 feet	4000 feet	4000 feet	4000 feet
Driver source impedance	3–7 kΩ	450 Ω min	100 Ω	54 Ω
Receiver input resistance	3 kΩ	4 kΩ	4 kΩ min	12 kΩ
Mode	Single-ended	Single-ended	Differential	Differential
Number of drivers and receivers allowed on one line	1 driver 1 receiver	1 driver 10 receivers	1 driver 10 receivers	32 drivers 32 receivers

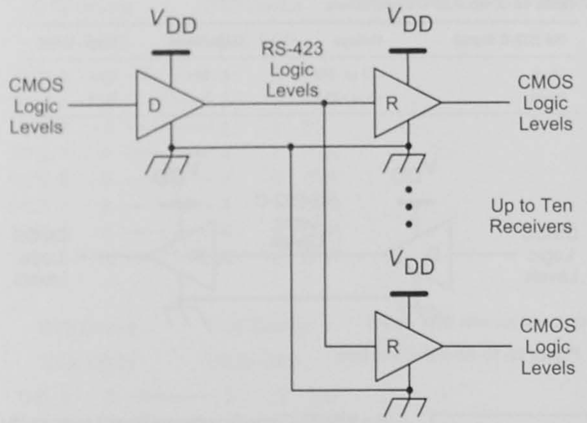


Figure 12-11 RS-423 interface.

RS-422 Standard

A problem experienced with the single-ended drivers and receivers of RS-232-C and RS-423 is that for long line lengths, noise and ground shifts can cause errors in the received data. Noise and ground shifts appear as common-mode signals; that is, they affect each line equally. The RS-422 line drivers and receivers operate with differential amplifiers as shown in Figure 12-12. These drivers eliminate much of the common-mode noise experienced with long transmission lines[5]; the line lengths and data rates that can be achieved will be shown in Table 12-4, along with the RS-422 electrical specifications.

[5] Approximately 100 Ω.

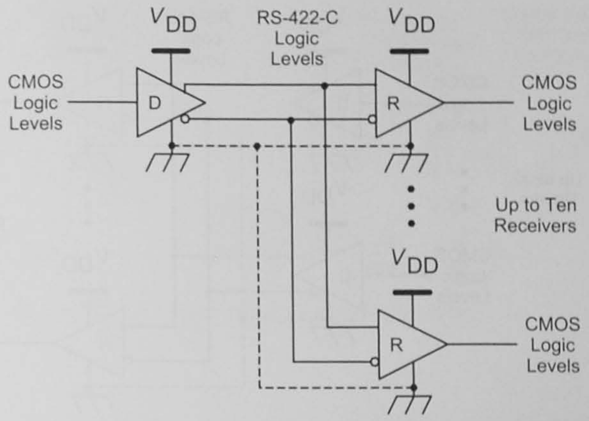


Figure 12-12 RS-422 interface.

RS-485 Standard

The RS-485 standard allows a bus architecture with multiple sources and receivers.

The RS-485 standard is similar to RS-422 in that it uses differential line drivers and receivers. However, as shown in Figure 12-13, the standard provides for multiple drivers and receivers in a bussed environment. Up to 32 driver/receiver pairs can be used together. For the RS-485 specifications, see Table 12-4.

Serial Interface Electrical Specifications

There are four electrical specifications in use for interconnecting serial interfaces. These are shown in Table 12-4. The two most widely used are RS-232-C and RS-485. The latter, which offers much higher data rates over longer distances than the RS-232-C standard, is less widespread, however.

Each of the electrical standards shown in Table 12-4 requires a level converter to translate the TTL or CMOS logic levels of the microcontroller's serial data input and output lines to the voltages specified by the standard. Figure 12-14 shows the SCI connected to a MAX3232 CMOS-to-RS-232 level converter.

Low-Voltage Differential Signaling (LVDS)

Another electrical interface being used for higher speed serial data networks in both onboard and offboard applications is *low-voltage differential signaling (LVDS)*. This interface is similar to RS-485 because differential transmitters and receivers are used. Differential line drivers can operate at much higher speeds because the differential line pair is relatively immune to common-mode noise. Data rates up to 2 gigabits per second are possible with this technology.

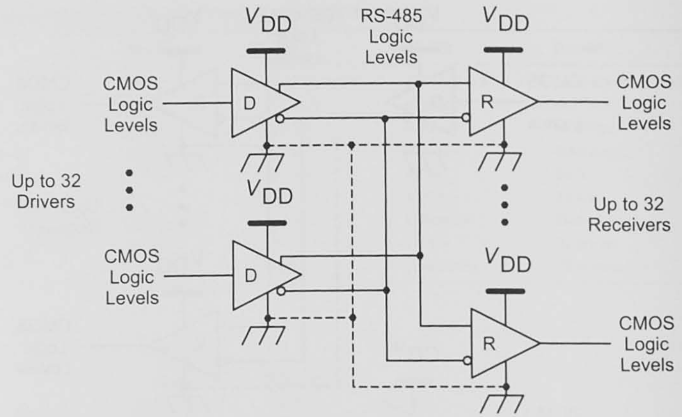


Figure 12-13 RS-485 interface.

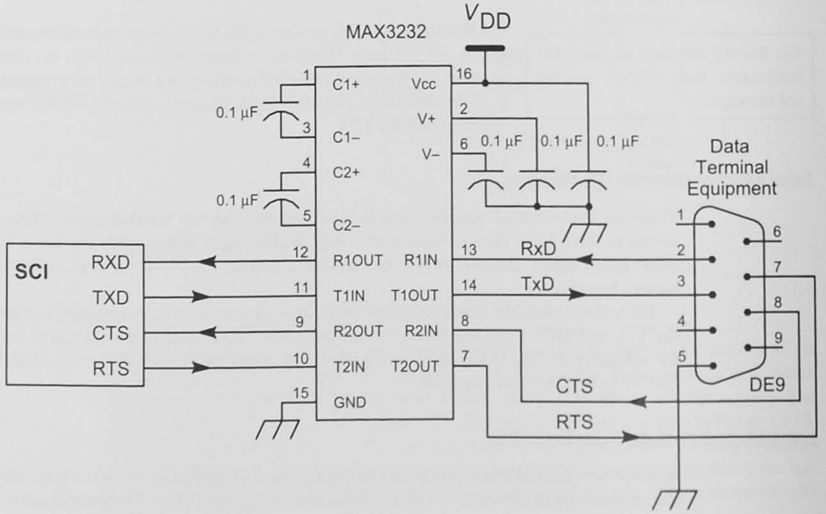
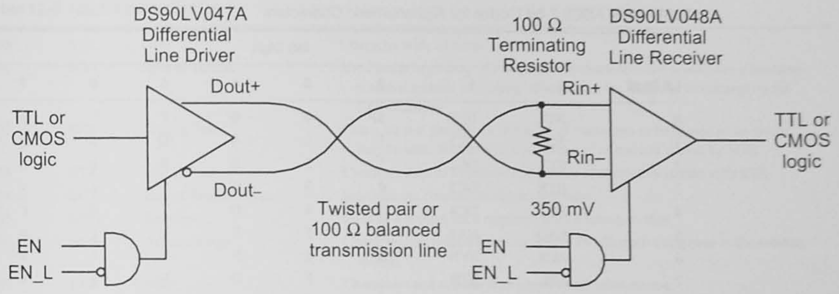


Figure 12-14 SCI with RS-232-C interface.



DS90LV047A Differential Line Driver
100 Ω Terminating Resistor
DS90LV048A Differential Line Receiver
TTL or CMOS logic
Dout+
Dout–
Rin+
Rin–
Twisted pair or 100 Ω balanced transmission line
350 mV
EN
EN_L

Figure 12-15 Low-voltage differential signaling (LVDS) interface.

The LVDS is a standard promoted as ANSI/TIA/EIA-644-A[6]. Unlike RS-232-C, the standard does not include functional specifications, protocol, or cable characteristics. It does specify a differential line driver and receiver configuration, as shown in Figure 12-15.

12.5 ASCII Data and Control Codes

The most commonly used code for sending data between a terminal device (a keyboard and a display) and a computer is the *American Standard Code for Information Interchange, or ASCII*. The ASCII code uses 7 bits to encode 96 printable and 32 *control* characters, as shown in Table 12-5; the printable characters, in the right-most six columns, have the codes 0x20–0x7E. The control codes (columns 0 and 1) are used by serial devices to provide some control of what is being transferred. For example, the CR code (0x0D) is sent to cause the printing terminal or display to perform a carriage return. The definitions for the other control codes are given in Table 12-6.

A *control code* may be sent by holding down the terminal's control key and typing another printable key.

Control codes are often used by software to provide special functions. For example, in some systems you can stop and start the output to a terminal by typing the DC3 (0x13) and DC1 (0x11), respectively. The control key on the keyboard of your terminal or PC allows you to send control codes. When the control key is pressed and held while another, printable character is typed, the effect is to map columns 4 and 6 into column 0 and 5 and 7 into column 1. For example, to send the DC3 character, one would press and hold the control key and type either S or s. This control-key/printable-key combination is known as control-S. See Examples 12-1 through 12-3.

[6] ANSI = American National Standards Institute, TIA = Telecommunications Industry Association, EIA = Electronic Industries Association.

Table 12-5 ASCII 7-bit Codes for Alphanumeric Characters

	MS Digit							
LS Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Example 12-1 Finding the ASCII Code for a Character

Use Table 12-5 to find the hexadecimal ASCII codes for the characters A, a, and].

Solution

A = 0x41, a = 0x61,] = 0x5D

Example 12-2 Finding the ASCII Code for a Character

Use Table 12-5 to find the hexadecimal ASCII codes for the control characters CR, BEL, and LF.

Solution

CR = 0x0D, BEL = 0x07, LF = 0x0A

Table 12-6 ASCII Control Codes

00	NUL	Null	Character with all zeros
01	SOH	Start of Header	Used at the beginning of a sequence of characters that constitutes a machine-readable address of routing information; the header is terminated by the STX character
02	STX	Start of Text	Character that precedes a sequence of characters to be treated as an entity; may be used to terminate a sequence of characters started by SOH
03	ETX	End of Text	Character used to terminate a sequence of characters started with STX
04	EOT	End of Transmission	Indicates the conclusion of a transmission
05	ENQ	Enquiry	Used as a request for a response from a remote station
06	ACK	Acknowledge	Character transmitted by a receiver as an affirmative response to the sending station
07	BEL	Bell	Character used to control an alarm or attention device
08	BS	Back Space	Controls the movement of the printing mechanism back one space
09	HT	Horizontal Tab	Controls the movement of the printing mechanism to the next predefined tab position
0A	LF	Line Feed	Moves the printing mechanism to the next line; in some systems, this may be interpreted as a "new line" (NL), where the print mechanism moves to the beginning of the next line
0B	VT	Vertical Tab	Controls the movement of the printing mechanism to the next predefined printing line position
0C	FF	Form Feed	Moves the printing mechanism to the start of the next page
0D	CR	Carriage Return	Moves the printing mechanism to the start of the line
0E	SO	Shift Out	Indicates that the code combinations following are outside the character set of the standard ASCII table until a Shift In character is received
0F	SI	Shift In	Indicates that the code characters following are to be interpreted according to the standard ASCII table
10	DLE	Data Link Escape	Changes the meaning of a limited number of following characters; DLE is usually terminated by a Shift In character
11	DC1	Device Controls	Characters used to control ancillary devices associated with data processing
12	DC2		
13	DC3		
14	DC4		
15	NAK	Negative Acknowledge	Transmitted by a receiver as a negative response to the sender
16	SYN	Synchronous Idle	Character used by a synchronous transmission system in the absence of any other characters to maintain synchronism between the transmitter and receiver
17	ETB	End of Transmission Block	Used to indicate the end of a block of data
18	CAN	Cancel	Indicates that the data with which it is sent is in error or is to be disregarded
19	EM	End of Medium	Sent with data to represent the physical end of the medium
1A	SUB	Substitute	Character that may be substituted for a character that is invalid or in error
1B	ESC	Escape	Control character intended to provide code extension; usually a prefix affecting the interpretation of a limited number of contiguously following characters
1C	FS	File Separator	Information separators that may be used within data
1D	GS	Group Separator	
1E	RS	Record Separator	
1F	US	Unit Separator	

Example 12-3 Finding the ASCII Code for a Character

How would you send the BEL character from a terminal keyboard?

Solution

BEL is a control character that can be sent by holding down the "Ctrl" key on the keyboard while pressing the "G" key.

12.6 Asynchronous Data Flow Control

Flow control refers to a higher level of handshaking needed to control the software transferring data via serial ports. For example, in the transfer of data from one computer to another, if the receiving computer cannot deal with the incoming data fast enough, data may be lost. If this happens, the receiving computer must send a message to the other computer to stop sending data until it is ready to receive some more. There are two ways to achieve flow control.

Hardware flow control: The request-to-send (RTS) and clear-to-send (CTS) handshaking signals are used in hardware flow control. The sending and receiving computers must control and sense these bits in the communication software.

Software flow control: Software flow control is called the *XON/XOFF* protocol. The XOFF character (ASCII DC3, 0x14, Ctrl-S) is sent by the receiving station to turn the transmission off. The XON character (ASCII DC1, 0x11, Ctrl-Q) turns it on again. The communication software must detect these characters being sent.

12.7 Debugging and Trouble Shooting

The serial interface has caused problems for many computer users. The major problems stem from a lack of documentation about what hardware has been implemented and from failure to set up the UART data transmission parameters correctly. The following procedure is suggested to help solve your serial interfacing problems.

Choose the Correct Cable

The cable to be used depends on the types of interface to be interconnected. You must find out if the devices are DTE or DCE. If the documentation does not show this, disconnect all cables and check for a negative voltage at pin DB25-2 or DE9-3. If a negative voltage exists when no characters are being sent, the interface is a DTE; otherwise, it is a DCE. When one device is a DTE and the other DCE, a DTE-DCE cable is required. If both are DTE or both DCE (unlikely), a null modem cable is required.

The number of wires in the cable depends on the handshaking and flow control used in the system. Hardware handshaking and flow control require a full DTE-DCE or null modem cable as shown, respectively, in Figures 12-6 and 12-7. If software flow control or no flow control is used, a minimal cable (Figure 12-8 or 12-9) can be used.

A very useful tool to have when working with RS-232-C interface devices is the RS-232-C tester shown earlier (Figure 12-5).

Table 12-7 Serial I/O drivers

C Function	Function Purpose
`Void init_sci( void );`	Initialize the microcontroller SCI to 8 data bits, 1 stop bit, no parity, and 9600 baud; enable the SCI transmitter and receiver
`void put_char( char output );`	Wait until the transmit data register is empty and output the character
`char get_char( void );`	Wait until a character is received and return it
`int char_ready( void );`	Check if a character has been received; return TRUE if so; otherwise return FALSE

Choose the Correct Communication Parameters

After you have connected the two interfaces with the correct cable, make sure that the software at each end is using the same parameters. The data rate (baud rate), number of data bits, type of parity, and the number of stop bits must be specified. In some communication systems, the type of flow control can be chosen.

12.8 Asynchronous Serial I/O Software

It is useful to write serial I/O software in the form of general-purpose I/O drivers that can be used by any application program. The needed drivers include an initialization routine to set up the microcontroller's communications interface and routines to input and output characters. See Table 12-7 and Example 12-4.

Example 12-4 Serial I/O Drivers

```
/***********************************************************
 * Serial I/O Driver Software Design
 * You must insert code for your own microcontroller
 ***********************************************************/
/***********************************************************
 * Initialize the SCI to 8 data bits, 1 stop bit, no parity
 * and 9600 baud (9600, 8, N, 1)
 ***********************************************************/
void init_sci( void ) {
  /* Set control registers for 8 data, 1 stop and no parity */
  /* Enable Transmitter and Receiver */
  /* Set the baud rate */
  return;
/***********************************************************
 * Check to see if a character has been received.
 * Return TRUE if so, otherwise return FALSE
 ***********************************************************/
int char_ready( void ) {
  /* If the Receive Data Register Full Flag (RDRF) is set */
  if ( RDRF == 1 ) {
    /* Then a character is there */
    return( TRUE) ;
  else
    /* Else no character has been received */
    return( FALSE );
}
/*************************************************************
 * Put a character
 ************************************************************/
void put_char( char send_data ) {
  /* Wait until the Transmit Data Register Empty flag is set */
  while ( TDRE == 0 );
  /* The last data has gone, now output the new byte */
  TX_DATA = send_data;
  /* Reset the Transmit Data Register Empty flag if needed */
  return;
}
/*************************************************************
 * Get a character
 ************************************************************/
char get_char( void ) {
  char temp_char;
  /* Wait until a character has been received */
  while ( char_ready() == FALSE );
  /* Now a character has been received, read it and return */
  temp_char = RX_DATA;
  return( temp_char );
}
```

12.9 Synchronous Serial Peripheral Interface (SPI)

A synchronous serial peripheral interface includes a clock signal.

A simple *synchronous* serial interface is the serial peripheral interface (SPI). It is synchronous because the device that is sending the data also supplies a clock signal. The receiver uses this as a shift clock to shift the data into its receiving shift register.

SPI Characteristics

The SPI is a simple serial interface. Unlike the inter-integrated circuit (IIC), the controller area network (CAN), and some of the other serial data interfaces, there is no defined data protocol that includes device addressing or error checking. There can be only one device, called the *master*, controlling the data transfer. If there are to be multiple receivers of the information, called *slaves*, they must be selected with hardware, as we will describe.

Figure 12-16 shows that an SPI system consists of a *master device* and a *slave device*. Some systems allow multiple masters with additional control signals, but only one device can be a master at a time. Another thing to notice is that the two shift registers act together, with data being shifted out of each one into the other simultaneously. This means that if the slave has

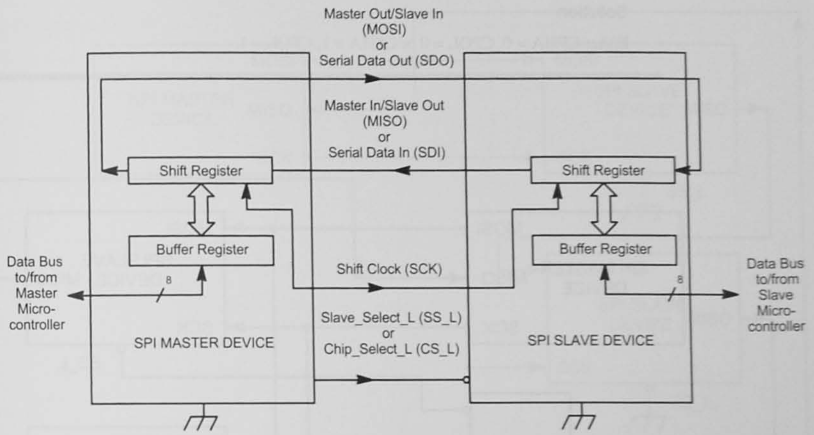


Figure 12-16 Serial peripheral interface.

data to send to the master, the master must control the transfer, a requirement that makes this somewhat simple device more complicated for bidirectional data transfer operations.

Configurations with multiple slave devices can be found, as well. Figure 12-17 shows a single master with multiple slaves. Since only one slave may be active at a time, a decoder circuit allows the master to choose the slave that is to be active. In this case, the slave device must have an open drain or three-state output.

Figure 12-18 shows a single master with a daisy-chain connection of the multiple slaves. Software in the master will control how many shift clocks are to be asserted to shift the data to the proper destination. Notice that all slave data shift registers will be shifting their data; your SPI control software must consider this.

Clocking the SPI Data

As Figure 12-16 shows, the shift clock is used to shift the data in and out of the SPI's data registers. Because no universal standard for SPI devices specifies the precise time that a shift clock edge must be relative to valid data, most SPI master devices provide a user-selectable clocking signal. Figure 12-19 shows the clocks available in a typical microcontroller such as the Freescale HCS12. Two bits control one of four clocking schemes. Table 12-8 shows that you may choose odd, even, rising, or falling edges to determine when the levels on the serial-in or serial-out lines are sampled. See Example 12-5.

Example 12-5 Choosing the SPI Clock

The 74HC595 8-bit shift register shown later (Figure 12-20) requires a positive-going-edge clock to shift the serial data into the register. Which clock phase (CPHA) and clock polarity (CPOL) values could you use?

Solution

Either CPHA = 0, CPOL = 0 or CPHA = 1, CPOL = 1.

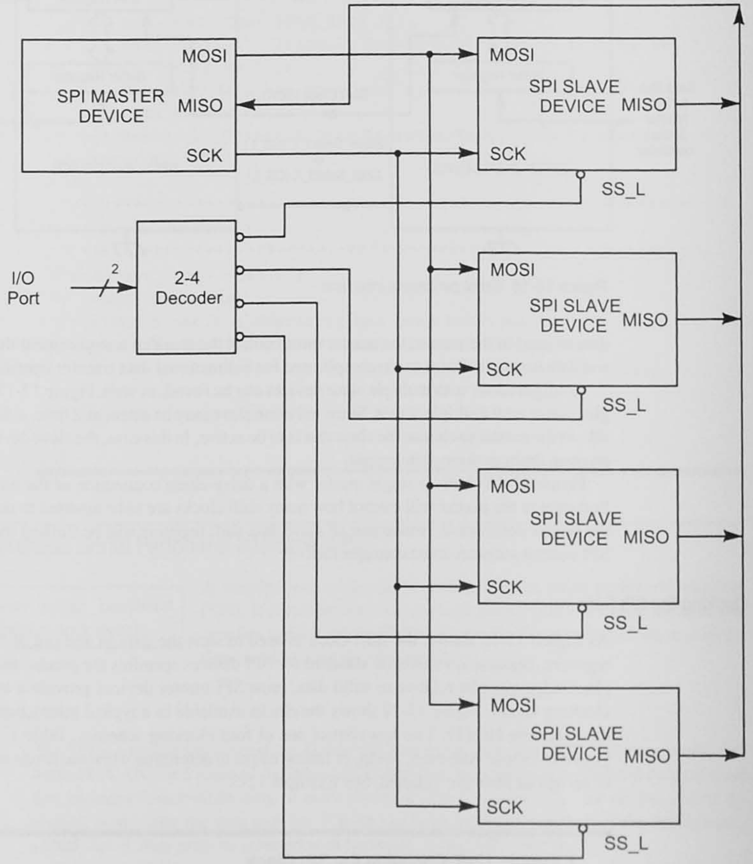


Figure 12-17 Single master, multiple slaves.

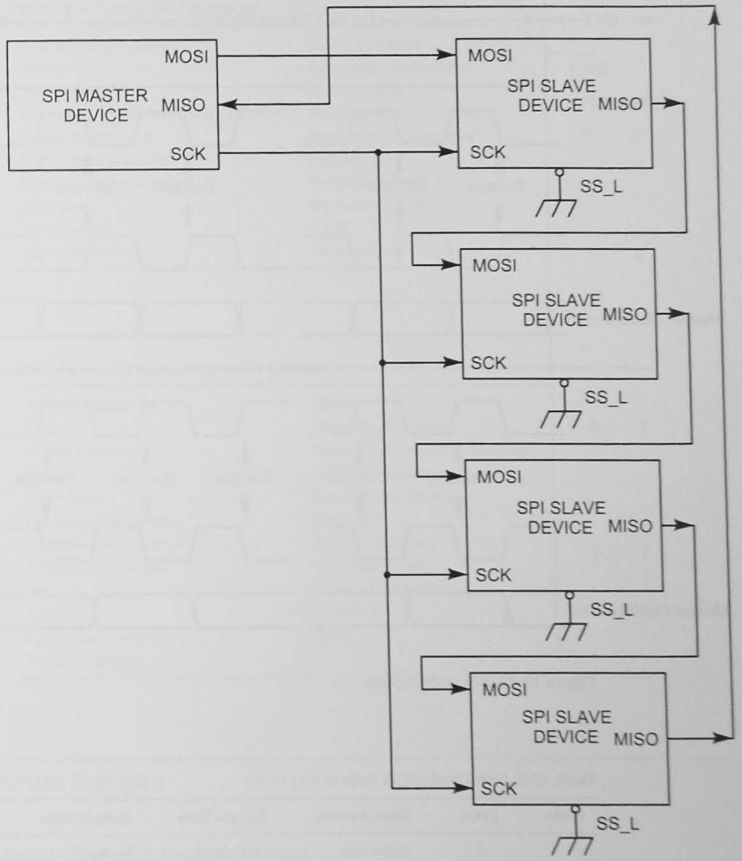


Figure 12-18 Single master, daisy-chain slaves.

The Software SPI

If your microcontroller does not have an integrated SPI, you can simulate device operation by controlling bits on a parallel I/O port. This process, called *bit banging*, can be used to generate serial I/O. Four bits will be needed to simulate the MOSI, MISO, SCK, and SS_L signals. Precise timing is not needed as long as you ensure that the serial data out is stable when you clock the data into the serial-in shift register. The approach does require software overhead that would not be needed if the microcontroller had an SPI.

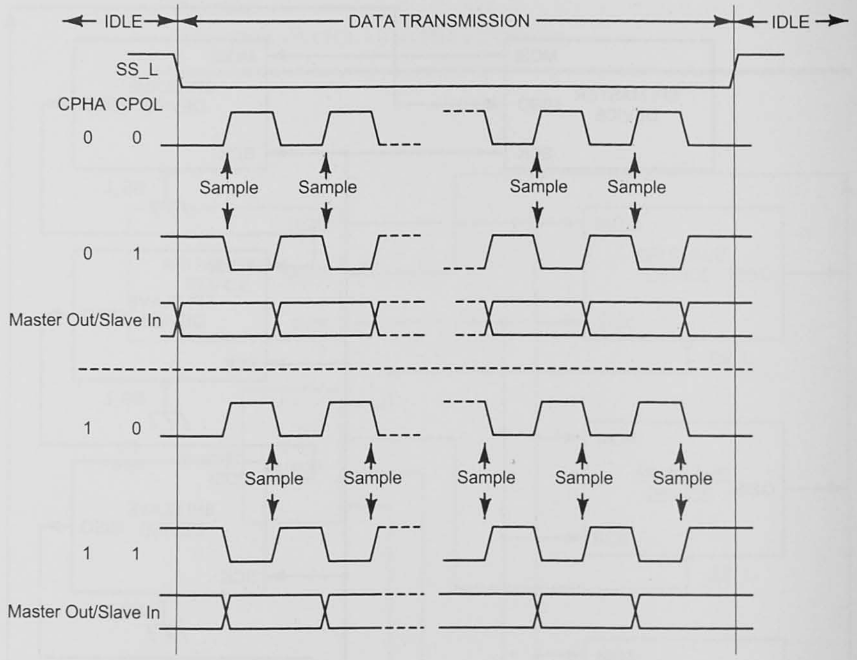


Figure 12-19 SPI clock signals.

Table 12-8 Shift Clock (SCK) Polarity and Phase

CPHA	CPOL	Clock Polarity	Sample Time	Sample Edge	SCK Idle State
0	0	Active high	Odd edges	Rising	Low
0	1	Active low	Odd edges	Falling	High
1	0	Active low	Even edges	Falling	Low
1	1	Active high	Even edges	Rising	High

SPI Typical Devices and Manufacturers

SPI devices first appeared in Freescale (Motorola) microcontrollers. The Microwire devices of National Semiconductor are similar. Table 12-9 shows the range of SPI devices available, and Table 12-10 lists some of the manufacturers offering these devices.

Table 12-9 Typical SPI Peripherals

Analog-to-digital converter	LED display driver
Analog switch	High voltage display driver
Audio mixer	Microcontroller
Controller area network (CAN) controller	Multimedia card
Digital potentiometer	Multiplexer pressure sensor
Digital signal processor	Real-time clock
Digital-to-analog converter	Temperature sensor
EEPROM	Touch screen controller
Flash memory	UART
LCD controller	USB controller

Table 12-10 Manufacturers of SPI Devices

AKM Semiconductor	Maxim
Altera	Microchip Technology
Analog Devices	National Semiconductor (Microwire)
Atmel	ON Semiconductor
Cirrus Logic	Ramtron International
Fairchild Semiconductor	SanDisk
Freescale Semiconductor	STMicroelectronics
Infineon Technologies	Texas Instruments
Intel	Winbond Electronics
Intersil	Xilinx
Lattice Semiconductor	Zilog
Linear Technology	

12.10 SPI Interface Examples

Expanding Parallel I/O with the SPI and Shift Registers

You do not have to use SPI devices to take advantage of a microcontroller's SPI. Figures 12-20 and 12-21 show how to use the SPI to add parallel input and output to your microcontroller. In Figure 12-20 a 74HC595 8-bit serial-in/serial-or-parallel-out shift register is used for additional output lines. Although this example shows only 8 bits, the serial-out pin (Q7') can be used as the serial input for another, cascaded 8-bit port. In another configuration, you can expand multiple 8-bit output ports in parallel by using a decoder for the SLAVE SELECT_L signal to select which of the 74HC595s are to receive the data.

Figure 12-21 shows the adding of 8 input bits with a 74HC165 parallel-in/serial-out 8-bit shift register. It too can be expanded to provide more input bits by cascading additional chips.

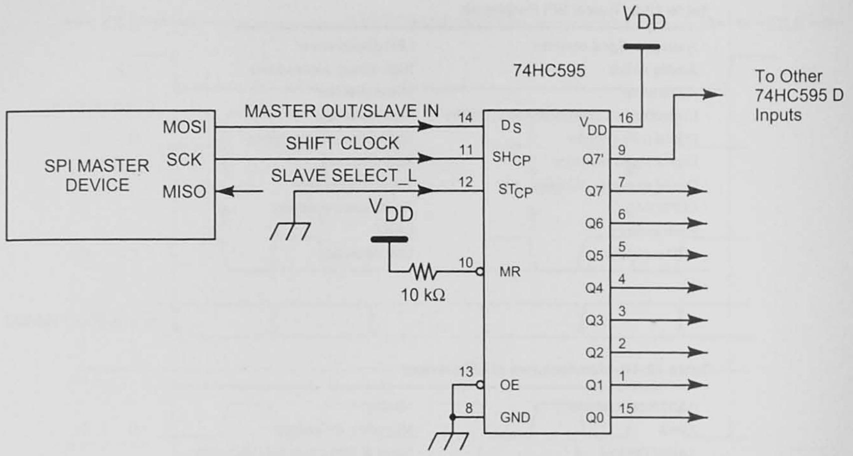


Figure 12-20 Adding parallel output with the SPI.

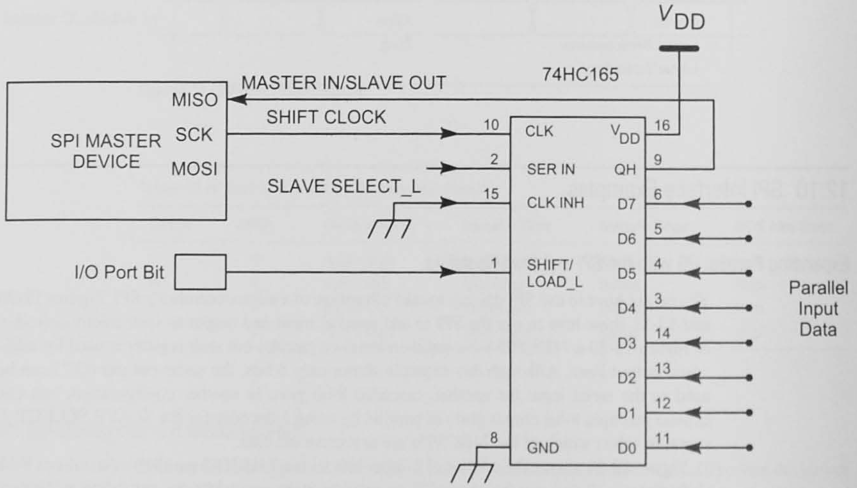


Figure 12-21 Adding parallel input with the SPI.

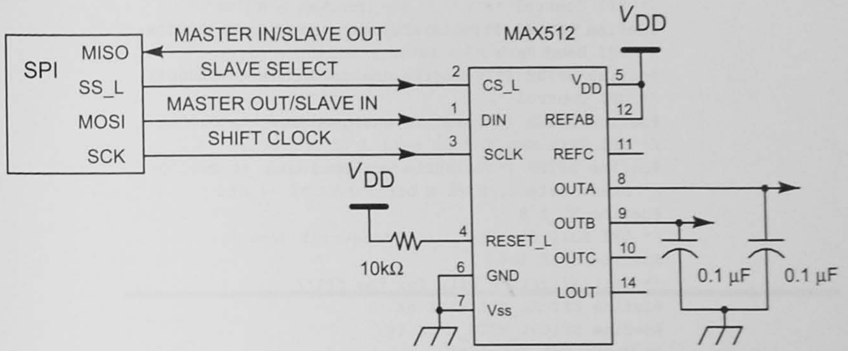


Figure 12-22 SPI with Maxim 512 D/A converter.

Digital-to-Analog Output

While many microcontrollers have an analog-to-digital converter input port, often they do not have a corresponding digital-to-analog converter for analog output signals. Figure 12-22 shows a MAX512[7], three-channel, digital-to-analog converter. It interfaces to the SPI and, as shown, outputs two analog channels on OUTA and OUTB. The third channel, OUTC, and the latched digital output, LOUT, are not used in this application. See Example 12-6.

Example 12-6 Digital-to-Analog Converter with C

```
/*************************************************************
 * Sample Serial Peripheral Interface Example
 * This program continuously outputs a sawtooth wave to the SPI
 * port connected to a Maxim MAX512 serial D/A converter
 *************************************************************/
/*************************************************************
 * Define the microcontroller specific I/O ports used on a
 * Freescale MC9S12C32
 *************************************************************/
/* Port M */
#define PTM   (*(volatile unsigned char *) 0x0250)
/* Data Dir Reg */
#define DDRM  (*(volatile unsigned char *) 0x0252)
```

[7] Maxim MAX512/MAX513 Low-Cost, Triple, 8-bit Voltage-Output DACs with Serial Interface. http://www.maxim-ic.com.

```
/* SPI Control */
#define SPICR1 (*(volatile unsigned char *) 0x00D8)
/* SPI Baud Rate */
#define SPIBR (*(volatile unsigned char *) 0x00DA)
/* SPI Control */
#define SPISR (*(volatile unsigned char *) 0x00DB)
/* SPI Data Reg */
#define SPIDR (*(volatile unsigned char *) 0x00DD)
/* Slave Select, Port M bit 3 */
#define SS_L 8
/* SPI Busy flag */
#define SPIF 128
/* Initialization bits for the SPI*/
#define SPICR1_SPE_MASK 64
#define SPICR1_MSTR_MASK 16
/* Maxim D/A Setup */
#define DA_SETUP 0b10110001   /* Output A enabled */
/*******************************************************/
void main(void) {
volatile char temp;
unsigned char sawtooth;
/*******************************************************/
 /*******************************************************
  * Initialize your microcontroller's I/O
  *******************************************************/
 /* Set Port M direction to be able to control SS_L output */
   DDRM |= SS_L;
 /* Enable SPI and set master mode */
   SPICR1 =  (SPICR1_SPE_MASK | SPICR1_MSTR_MASK);
 /* Set the SCK to 4 MHz */
   SPIBR = 0;
 /*******************************************************/

 /* Initialize the sawtooth data */
   sawtooth = 0;
 /* DO */
   for(;;) {
   /* Set SS_L low to select the D/A */
     PTM &= ~SS_L;
   /* Send the first byte to the D/A
      * Write an 8-bit control word to the serial D/A
      * converter before writing the data */
      SPIDR = DA_SETUP;
   /* Wait until the byte is shifted out */
     while ( (SPISR & SPIF) == 0 ) { };  /* Wait for SPIF */
   /* Send the second byte to the D/A and
    * increment the value */
```

```
      SPIDR = sawtooth++;
   /* Wait until the byte is shifted out*/
      while ( (SPISR & SPIF) == 0 ) { };
   /* Clear the SPI flags */
      temp = SPISR;
      temp = SPIDR;
   /* Raise the SS_L line to tell the D/A
    * to output the data */
      PTM |= SS_L;

   } /* wait forever */
}
```

Liquid Crystal Display

Many inexpensive liquid crystal displays make excellent display devices for embedded systems. Most can interface to the microcontroller in at least two ways, including 4-bit and 8-bit parallel connections.

Figure 12-23 shows an LCD module. To reduce the parallel I/O bits needed to drive the LCD, a 74HC595 serial-in/parallel shift register is connected to the SPI port on the microcontroller. Figure 12-24 illustrates the hardware design, and Example 12-7 shows software to display characters on the LCD.

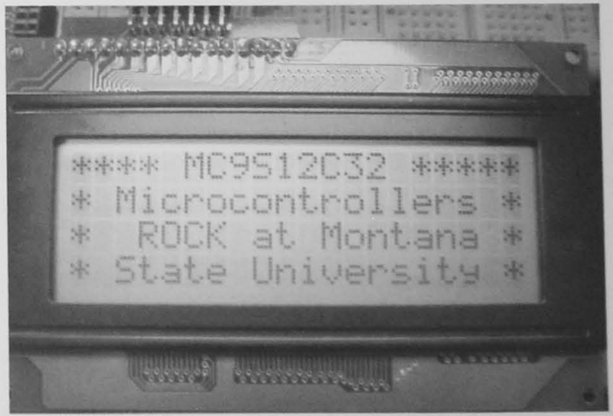


Figure 12-23 LCD module.

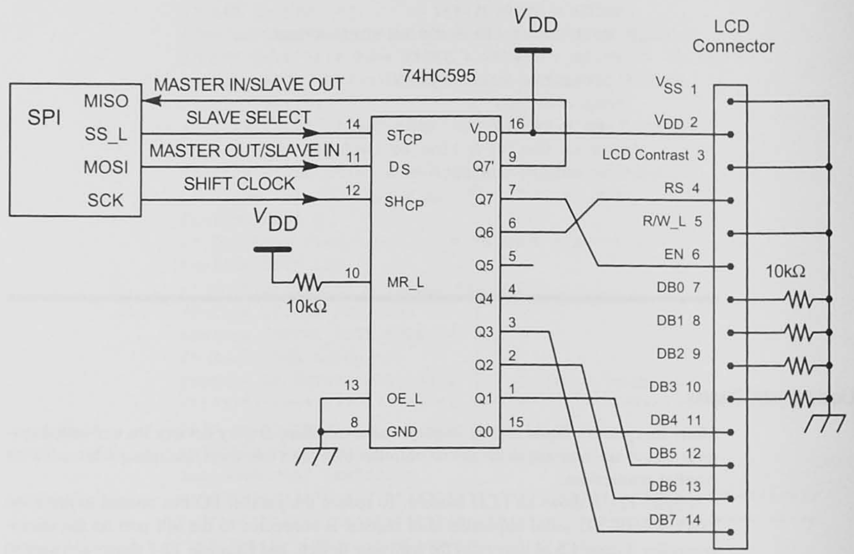


Figure 12-24 SPI and liquid crystal display.

Example 12-7 Liquid Crystal Display Drivers

```
/*********************************************************
 * LCD Display Program
 *********************************************************/
/*********************************************************
 * Define the microcontroller specific I/O ports used on a
 * Freescale MC9S12C32
 *********************************************************/
/* SPI Control 1 */
#define SPICR1 (*(volatile unsigned char *) 0x00D8)
/* SPI Control 2 */
#define SPICR2 (*(volatile unsigned char *) 0x00D9)
/* SPI Baud Rate */
#define SPIBR (*(volatile unsigned char *) 0x00DA)
/* SPI Control */
#define SPISR (*(volatile unsigned char *) 0x00DB)
/* SPI Data Reg */
#define SPIDR (*(volatile unsigned char *) 0x00DD)
/* SPI Busy flag */
```

```
#define SPIF 128
/* Initialization bits for the */
#define SPICR1_SPE_MASK 64
/* SPI */
#define SPICR1_MSTR_MASK 16
#define SPICR2_MODFEN_MASK 16
#define SPICR1_SSOE_MASK 2
/*********************************************************
 * Define the commands to be sent to the LCD
 *********************************************************/
#define EN 0x80
#define RS 0x40
#define DB7 0x80
#define FSET_8_BIT 0x03
#define FSET_4_BIT 0x02
#define FSET_4_LINE  0x28
#define FSET_D_OFF  0x08
#define FSET_CLEAR  0x01
#define FSET_ENTRY  0x06
#define FSET_D_ON   0x0E
#define FSET_CUR_OFF 0x0C
#define HOME  0x02
/*********************************************************
 * Define an array to be used for addressing each line
 *********************************************************/
unsigned char LINE[4][20] = {
 {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19},
 {64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83},
 {20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39},
 {84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,
  103}};
/*********************************************************/
void delay_X_ms( unsigned int );      /* Variable Delay */
void spi_send_byte( unsigned char );  /* Send a byte */
void lcd_put_command( unsigned char );/* Send a command */
void init_spi( void );                /* Initialize the SPI */
void lcd_init( void );                /* Initialize the LCD */
void lcd_print( char *str_pointer );  /* Print a string */
/* Move cursor */
void lcd_move_cursor( unsigned char, unsigned char );
/*********************************************************
 * Test program printing a message on the LCD
 *********************************************************/
void main(void) {
  /* Initialize the SPI and the LCD */
    lcd_init();
  /* Print the message below */
    lcd_move_cursor( 1,1 );              /* Line 1 */
    lcd_print( "**** MC9S12C32 *****\0");
    lcd_move_cursor( 2,1 );               /* Line 2 */
    lcd_print( "* Microcontrollers *\0" );
    lcd_move_cursor( 3,1 );               /* Line 3 */
    lcd_print( "*  ROCK at Montana **\0" );
    lcd_move_cursor( 4,1 );               /* Line 4 */
    lcd_print( "* State University *\0");
  /* Turn cursor off */
    lcd_put_command( FSET_CUR_OFF );
    for(;;) {
      } /* wait forever */
}
/*************************************************************
 * Initialize the SPI
 *************************************************************/
void init_spi( void ) {
/*************************************************************/
  /* Initialize the SPI */
  /* Enable SPI in master mode */
    SPICR1 = SPICR1_SPE_MASK|SPICR1_MSTR_MASK|SPICR1_SSOE_MASK;
    SPICR2 = SPICR2_MODFEN_MASK;
  /* Set the SCK to 4 MHz */
    SPIBR = 0;
}
/*************************************************************
 * Initialize the LCD
 *************************************************************/
void lcd_init( void ){
/*************************************************************/
  /* Initilize the SPI */
    init_spi();
  /* Initialize the LCD */
  /* Delay 15 ms in case the power just came on */
    delay_X_ms( 15 );
  /* Send the first command */
    lcd_put_command( FSET_8_BIT );
  /* Delay 5 ms */
    delay_X_ms( 5 );
  /* Send the first command again */
    lcd_put_command( FSET_8_BIT );
  /* Delay > 100 us */
    delay_X_ms( 1 );
  /* Send the first command again */
    lcd_put_command( FSET_8_BIT );
  /* Set the interface to 4 bits */
    lcd_put_command( FSET_4_BIT );
  /* Set interface to 4 line, 5x7 chars */
```

```
    lcd_put_command( FSET_4_LINE );
  /* Set display off */
    lcd_put_command( FSET_D_OFF );
  /* Clear display */
    lcd_put_command( FSET_CLEAR );
  /* Set entry mode */
    lcd_put_command( FSET_ENTRY );
  /* Turn display on */
    lcd_put_command( FSET_D_ON );
}
/*************************************************************
 * Send a byte to the SPI
 *************************************************************/
void spi_send_byte( unsigned char spidata ){
  volatile unsigned char status, data;
/*************************************************************/
    SPIDR = spidata;
  /* Wait until the data has shifted out */
     while( (SPISR & SPIF) == 0 );
  /* Clear the flag by reading the status register
   * and then the data register */
    status = SPISR;
    data = SPIDR;
}
/*************************************************************
 * Send byte to display to the LCD
 *************************************************************/
void lcd_put_data( unsigned char character ){
  static unsigned char msn, lsn;
/*************************************************************/
  /* Input is an ASCII character to display */
  /* Split into two nibbles and send the ms nibble first */
    msn = (character >> 4) | RS;      /* Setting the RS high */
    lsn = (character & 0x0f) | RS;
  /* Send the most significant nibble */
    spi_send_byte( msn  );
    spi_send_byte( msn | EN );      /* Set the enable high */
    spi_send_byte( msn );
    delay_X_ms( 1 );
  /* Send the least significant nibble */
    spi_send_byte( lsn );
    spi_send_byte( lsn | EN );
    spi_send_byte( lsn );
    delay_X_ms( 1 );
}
/*************************************************************
 * Send command byte to the LCD
 *************************************************************/
void lcd_put_command( unsigned char character ){
  static unsigned char msn, lsn;
/*************************************************************/
  /* Input is an ASCII character to display */
  /* Split into two nibbles and send the ms nibble first */
    msn = (character >> 4);
    lsn = (character & 0x0f);
  /* Send the most significant nibble */
    spi_send_byte( msn  );
    spi_send_byte( msn | EN );     /* Set the enable high */
    spi_send_byte( msn );
    delay_X_ms( 2 );
  /* Send the least significant nibble */
    spi_send_byte( lsn );
    spi_send_byte( lsn | EN );
    spi_send_byte( lsn );
    delay_X_ms( 2 );
}
/*************************************************************
 * Print a null terminated string on the LCD
 *************************************************************/
void lcd_print( char *str_pointer ){
/*************************************************************/
  /* Print a null terminated string on the display */
    while (*str_pointer != 0){
    lcd_put_data( *str_pointer++ );
  }
  return;
}
/*************************************************************
 * Move cursor to a line, column.
 * Input is the line number 1 - 4, column number 1 - 20
 * If line is not 1 - 4, line is set to 1.
 * If column is not 1 - 20, column is set to 1
 *************************************************************/
void lcd_move_cursor(unsigned char line,unsigned char column){
/*************************************************************/
  /* Check for line 1-4 */
    if (line < 1 || line > 4) line = 0;
    else line = line - 1;
  /* Check for column 1 - 20 */
    if (column < 1 || column > 20) column = 0;
    else column = column - 1;
    lcd_put_command(LINE[line][column] | DB7);
}
```

12.11 Inter-Integrated Circuit (IIC or I²C)

This section briefly explains another option for serial data transfer. As in all serial data interfaces, the number of signal lines between the sources and destinations for information is reduced in comparison to parallel data transfer. This comes at the expense of reduced data transfer speed. The two systems described in this chapter; *inter-integrated circuit (IIC or I²C)* bus and the *controller area network (CAN)* bus, allow a network of sources and destinations for information with data synchronization, multiple master/slave organization, error checking, and addressing of devices on the network.

Some Common Terms

Address: A code to specify a device on the serial bus. The address codes used in the I²C and the CAN buses assist in arbitration.

Arbitration: Process that allows only one master to send data if more than one tries to control the bus at the same time.

Master/Slave: A master device is one that controls the transfer of data in a system. It initiates the data transfer and provides the needed timing. A slave device is controlled by the master to receive (and in some cases send) the data.

Multiple-master system: A system in which multiple devices may act as masters.

Receiver: The device that receives the data.

Synchronization: Providing a clock to synchronize the data transfer between two devices.

Transmitter: The device that sends the data.

Inter-Integrated Circuit (IIC or I²C) Serial Bus

The I²C uses two-wires (plus ground) for data and clock signals.

The I²C serial bus can be seen in Figure 12-25. This bus was developed by the Philips Semiconductor Company in the early 1980s. The current specification supports data rates of 100 kbit/s (standard

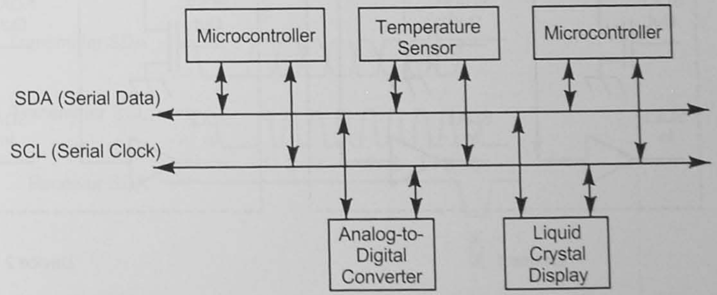


Figure 12-25 I²C serial bus.

mode), 400 kbit/s (fast mode) and 3.4 Mbit/s (high-speed mode). It is a multiple-master bus, and more than one device is capable of controlling the bus and sending data. There are two wires (plus a ground reference). Both SDA and SCK, the *serial data* and the *serial clock*, respectively, are bidirectional lines. Because there may be more than one device trying to transmit data simultaneously, open-drain transistors are used, as shown in Figure 12-26. When either bus line is idle (i.e., where no device is transmitting clock or data), the bus line is pulled high by the pull-up resistors.

I²C Bit Transfer

Figure 12-27 shows three conditions necessary to send data on the I²C bus. The start condition occurs when a master wants to send data. The master first checks to see that the SDA line is idle (high) and then lowers SDA while SCL is high. This is the *start condition*. The master then clocks data out by ensuring that SDA is stable (high or low) and then raises and then lowers SCL. The *stop condition* occurs at the end of the data message by changing SDA from low to high while SCL is high. All masters generate their own clock and data are valid only when the clock is high.

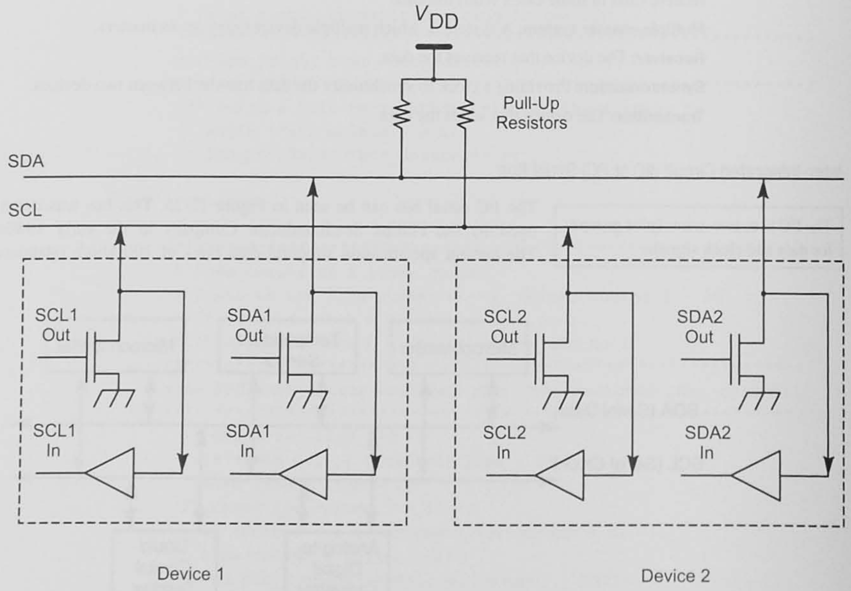


Figure 12-26 I²C serial bus interfaces.

Data Transfer

All data sent on the I²C bus are 8-bit bytes. The number of bytes sent in a message is unrestricted, but each byte must be followed by an acknowledgment bit sent by the slave. The message transfer starts with the start condition and ends with the stop condition (Figure 12-27).

Figure 12-28 shows how a byte is transferred. The master transmitter generates the start condition and then clocks out eight data bits. Following the eighth bit, it releases the SDA line and waits for the receiver to pull its SDA line low; this constitutes acknowledgment that the receiver has received all 8 bits. If the receiver does not generate the ACK bit, the transmitter can generate a stop condition and abort the data transfer.

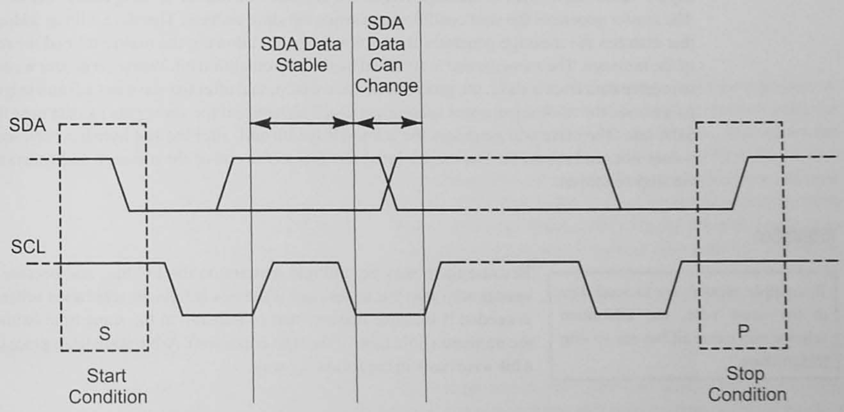


Figure 12-27 I²C start and stop and bit transfer timing.

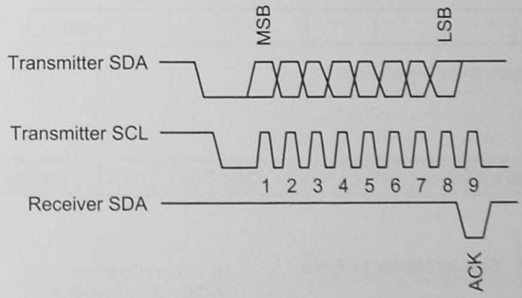


Figure 12-28 I²C data transfer with acknowledge.

Slave Addressing

Slave devices are selected by an address that is sent at the start of a message. The specification allows for 7-bit and 10-bit addresses. We will describe the 7-bit address operation here. The 10-bit address operation is similar.

The 7-bit address occupies the most significant bits of a byte with the least significant bit an R/W_L bit. (Figure 12-29). If R/W_L is low, the master will be transmitting to the slave (the slave is writing data); when the master is to read data from the slave, this LSB will be high.

The I²C Message

Figure 12-30 shows the I²C message format for the case of a master sending data to the slave. The master generates the start condition and sends the slave address. The slave with an address that matches the message generates the acknowledge bit, allowing the master to send the rest of the message. The message end is signified by the stop condition bit. When the master wishes to receive data from a slave, the process is similar except that after the slave has acknowledged its address, the master generates subsequent clock signals and the slave puts its data onto the SDA line. The slave still generates the acknowledge bit and, after the last byte has been sent, it does not send the ACK. The master interprets this as the end of the message and generates the stop condition.

Arbitration

If multiple masters try to send data at the same time, the arbitration scheme will cause all but one to stop transmitting.

Because there may be multiple masters on the I²C bus, and because a master may start a transfer only if the bus is free, an arbitration scheme is needed if multiple masters start to transmit at the same time (within the minimum hold time of the start condition). Arbitration takes place on a bit-wise basis in the following way.

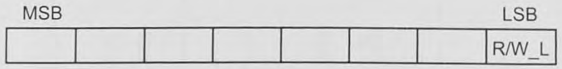


Figure 12-29 Slave address byte.

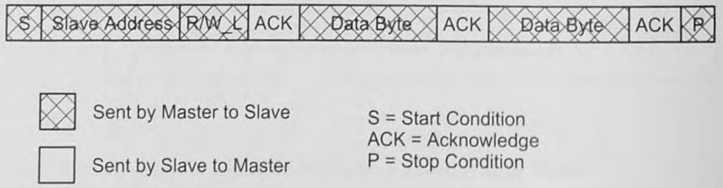


Figure 12-30 I²C message format.

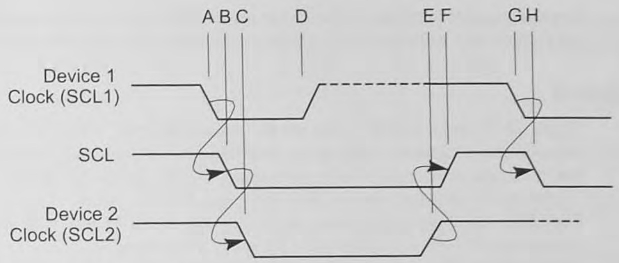


Figure 12-31 I²C clock synchronization.

Notice in Figure 12-26 that the driver for the SDA bus line is an open-collector transistor. A high level is achieved when the transistor is not being driven and the pull-up resistor pulls the line high. Thus, for SDA to be low, a transmitter must actively pull the line low. This allows the arbitration scheme to work. When a transmitter is transmitting data onto the SDA line, it also monitors the line level. If more than one transmitter are transmitting at the same time, the one that pulls the SDA line low will "win" the arbitration battle. If a transmitter sees that the SDA line is different from what it is sending, it will stop sending, allowing the other device to continue. This establishes a priority scheme according to which a lower slave address (with more zeros) has higher priority than a higher address. Even if the multiple masters are addressing the same slave, eventually a data bit will be different and one of the masters will discontinue.

Clock Synchronization

The I²C bus is designed to allow different devices to have different clock rates. Because the arbitration scheme relies on a bit-by-bit comparison of the SDA line, however, all devices must have a clock rate synchronized to the slowest device. The clock synchronization process works like this.

Assume that Device 1 in Figure 12-31 has a shorter clock period than Device 2. The synchronization starts when Device 1 pulls SCL1 low at point A and SCL follows at point B.[8] Device 2 detects this transition and pulls its clock (SCL2) low at point C. Both devices start counting their clock low period, and at point D Device 1 raises SCL1. Because Device 2 is the slower of the two, it does not raise SCL2 until point E. SCL transitions high at point F, and both devices start counting their clock high period. Device 1 is the faster of the two and at point G pulls SCL1 (and SCL) low. In this way, the amount of time SCL is held low is controlled by the slowest device and the time it is high by the fastest device. This clock synchronization takes place during every clock pulse.

Your Microcontroller's I²C Interface

Many microcontrollers today have a built-in I²C interface incorporating arbitration hardware and status bits that allow you to determine when messages have been sent successfully. It

[8] The A-B offset simply illustrates a propagation delay time.

can also generate interrupts to notify your program that an I²C message has been received. In microcontrollers without an I²C interface, clever programmers can bit-bang the I²C signals.

I²C Interface Example

Figure 12-32 shows an LM92 temperature sensor with an I²C microcontroller interface. The chip contains a 12-bit plus sign temperature-to-digital converter, and the microcontroller can read the temperature at any time by interrogating the chip on the I²C bus. The two address bits, A1–A0, select up to four devices. With both grounded, the device will respond to address 00. The LM92 can also be set up to act as a comparator that will generate an interrupt when the temperature exceeds a programmable set value. The amount of hysteresis that temperature changes impose before the alarm condition resets is programmable as well.

12.12 The Controller Area Network (CAN) Bus

Robert Bosch introduced the *controller area network*, or *CAN*, serial bus at the Society of Automotive Engineers congress in February 1986. The CAN bus can handle reliably short messages (up to 8 bytes) with multiple-master access. Although originally developed for automotive markets, this bus is finding uses in many other applications.

CAN Definitions

The CAN serial interface has its own jargon and terms. Here are a few definitions to help you understand some of the CAN descriptions that follow.

Acceptance filter: A digital keyword that incoming messages must match before the receiver accepts them.

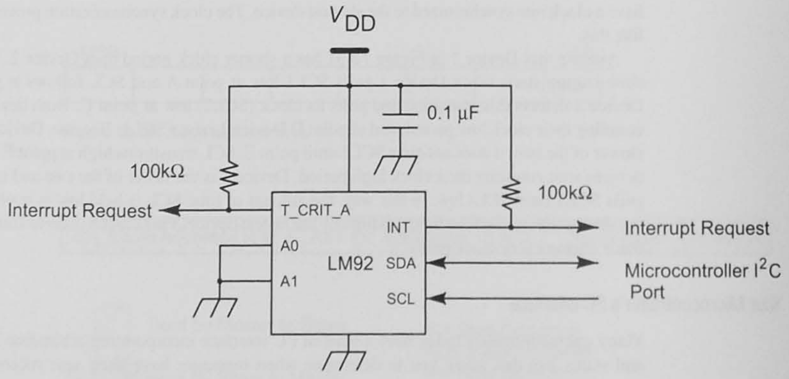


Figure 12-32 LM92 temperature sensor with I²C interface.

Basic CAN: Basic CAN devices implement in hardware only the basic functions of the protocol, such as generation and checking of the bit stream. All message management, such as accepting the message, must be done in software. See **Full CAN**.

Baud rate: Number of bits per second for data transmitted on the CAN bus. See **Time quantum**.

CRC: Cyclic redundancy check. A 15-bit error checking word used to detect bit errors in the preceding data and in itself.

CSMA/CD: Carrier sense, multiple-access collision detection. A method for avoiding or resolving errors when multiple devices try to send messages at the same time.

Data frame: CAN uses data frames when the node wants to send data. Remote frames are a request for information. A frame with the RTR (remote transmission request) bit set means that the transmitting node is requesting information of the type specified by the identifier.

Dominant level: A logic low level.

EOF (end-of-frame): A recessive (logic high) bit, similar to a stop bit in an asynchronous serial interface, that signifies the end of the current message buffer.

Extended frame: A data frame defined by CAN 2.0B with a 29-bit identifier.

Frame: A message consisting of the start-of-frame (SOF), arbitration, control, data, CRC, acknowledge (ACK), and end-of-frame (EOF) fields.

Full CAN: A full CAN device implements the whole bus protocol in hardware, including acceptance filtering and the message management. See **Basic CAN**.

Idle bus: A bus in the recessive mode (logic high) for more than three bit times.

Initialization mode: A mode that allows system initialization to be done because the CAN is disconnected from the CAN bus.

Recessive level: A logic high level.

Remote frame: See **data frame**.

SOF (start-of-frame): A dominant (logic low) bit used like the start bit in an asynchronous serial communications system.

Standard frame: A data frame defined by CAN 2.0A with an 11-bit identifier.

Synchronization jump: An increment of time quanta used to synchronize a receiver's bit sampling time with the incoming data.

Time quantum: A time interval less than the bit time. There may be 8 to 25 time quanta per bit.

CAN Serial Communications

The CAN bus is a serial bus system with each of the CAN devices, called *nodes*, connected to the bus capable of being a *master*. A master device can initiate data transmission to any of the other nodes on the bus, unlike the SPI, which allows only one master at a time with multiple slaves. The bus uses a single wire (actually two) to reduce the amount of wiring needed in its applications. The bus provides clock synchronization based on the data stream. These concepts require the clever design that Robert Bosch introduced in 1986.

CAN Serial Bus Basics

Recessive bits are logic high, and *dominant* bits are logic low.

Figure 12-33 shows a CAN bus. It may have two or more nodes. Because these nodes can be widely separated, by as much as 1000 meters, no individual node has no knowledge of the other nodes. This can lead to a collision of data bits if two or more start to transmit at the same time. The CAN design overcomes this problem by defining the electrical characteristics of the bus to be a wired-AND type, as shown in Figure 12-33. Each of the CAN nodes has a transmitter and a receiver. The transmitter uses an open-drain connection to the CAN bus, and a pull-up resistor establishes the logic levels on the bus. A logic high is called a *recessive* bit. The high is active because none of the nodes are pulling the bus low. (This is why it is called a wired-AND.) A logic low is called a *dominant* bit because one node can dominate all other nodes that are sourcing a recessive bit.

CAN Serial Bus Collision Detection and Arbitration

The problem of two or more nodes starting to transmit at the same time is solved in the following way

- Each of the nodes continuously monitors the bus with its received data line.
- Each bus transmission is started with a dominant (low) bit, called the *start-of-frame (SOF)* and proceeds with a multiple bit *identifier* (11 or 29 bits long) that defines the type of message data that is to follow.
- If two or more nodes are transmitting at the same time, eventually one of the identifiers will be different, with a low (dominant) bit in place of a high (recessive) bit.

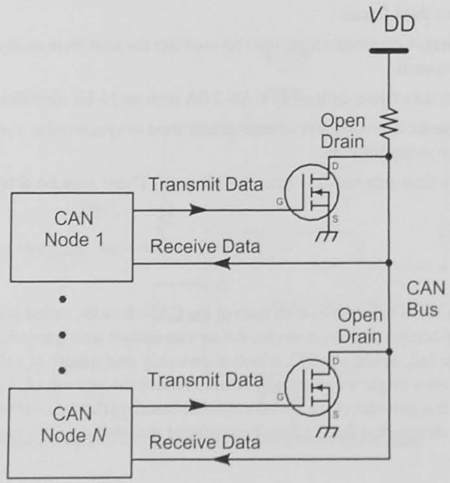


Figure 12-33 Basic CAN bus.

- Because the low is dominant, and because each of the nodes is monitoring the bus while it is transmitting, a node transmitting a recessive bit will recognize that another node is out there transmitting.
- Nodes with a recessive bit stop transmitting and allow other nodes to continue.
- Eventually only one node is left.
- Any node that stops waits until bus activity ceases and tries to send its message again.

This scheme is called *carrier sense, multiple-access with collision detect (CSMA/CD)* because nodes are able to detect other transmitters. A prioritization scheme is in effect because the node with the lower binary number for its identifier wins control of the bus. The message that follows contains up to 8 bytes of data and a 15-bit error checking code. The system is able to detect a variety of errors; and because it provides an acknowledgment bit, the receiving node can let the transmitting node know that the message was received without errors. As you might expect, the protocol to manage this consists of many more details. You will have to study your own microcontroller's CAN bus documentation to learn more.

CAN Serial Bus Interface

While the single-ended bus shown in Figure 12-33 explains the concept of the dominant and recessive bits, often in practice a differential, twisted-pair bus is used. The twisted-pair cable provides a transmission line with well-behaved characteristic impedance. This allows it to be terminated with a resistance to reduce reflections. It also has noise reduction properties to preserve data quality in noisy industrial environments.

A CAN bus transceiver, such as a Linear Technology LT1796, converts the CAN node's single-ended transmit and receive data lines to the balanced, differential CAN system signals CAN_H and CAN_L, as shown in Figure 12-34. The performance of the system in noise is greatly enhanced by the common-mode rejection of the differential receivers. The bus may be twisted-pair wires, either unshielded or shielded for additional noise rejection. The data bits are sent with start and stop bits, similar to the SCI described in Section 12.2, with additional characters to define the data frame.

Non-return-to-zero (NRZ) signaling encodes the data bits. Each node has its own clock and synchronizes it with the incoming data by detecting bit transitions. To assist clock synchronization, a *bit-stuffing* scheme is used. The receiver may lose bit synchronization if a number of consecutive bits of the same polarity are transmitted. To combat this, the transmitter will insert an additional bit of the opposite polarity into the bit stream after five consecutive ones or zeros. The receiver automatically detects the stuffed bit and removes it from the data.

The CAN Message

The CAN bus protocol includes address information, called the identifier, in the message frame. This is not a node address. Instead, it identifies the type of information being transmitted. For example, in an automotive application, nodes on the CAN bus may be sending engine rpm, coolant temperature, fuel level, and so on. Each node that needs a particular piece of information has a matching identifier filter and can pick off messages that are relevant to its job and discard others.

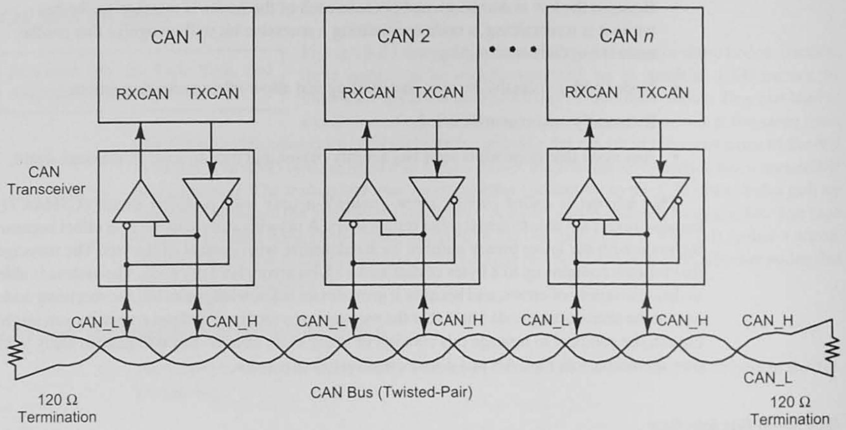


Figure 12-34 The CAN serial bus system.

The formats for standard and extended CAN frames are shown in Figure 12-35 and Table 12-11.

CAN Data Transmission

Each node on the system is responsible for *broadcasting* information to the system about its sensors. This means that the transmitter does not necessarily know which node is to be the receiver. When a node has information to send, it checks the CAN bus and if the bus is idle (recessive), it starts sending a data packet by asserting the start-of-frame bit. The 11- or 29-bit identifier identifies the type of information to come. The data length code (DLC), contains the number of data bytes in the message, zero to eight, and this is followed by the data and a 15-bit cyclic redundancy check (CRC) error detection word.

CAN Bus Clock

The clocking of the data on the CAN bus is derived from the data itself. The CAN hardware is able to synchronize its bit-sampling time with the incoming data stream. Each bit time is subdivided into smaller time elements called *time quanta*. When a message transmission starts, the first bit is the dominant SOF bit shown in Figure 12-35. The receiver detects this high-to-low transition and then looks for the low-to-high in the middle of the bit time. It can then adjust its internal timing quanta to be able to sample the following bits at the correct time.

If the oscillators in two nodes are slightly different and the bits in the frame continue to arrive, the bit changes relative to the sample point may drift around. If this occurs, and the

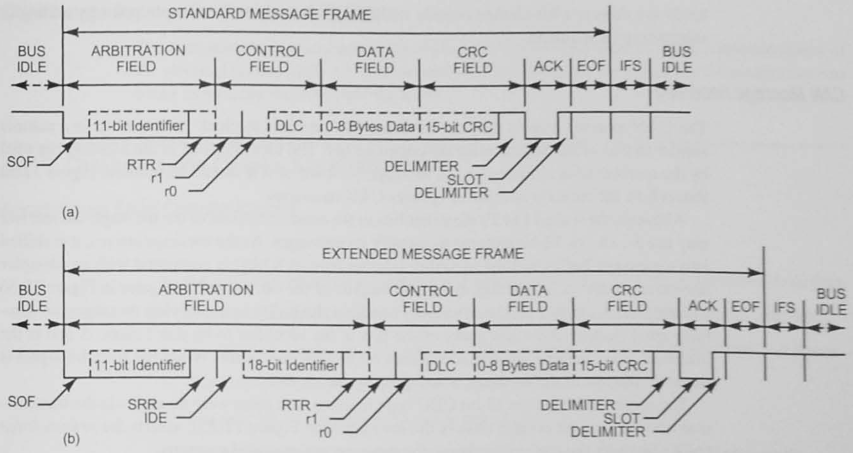


Figure 12-35 CAN message frames. (a) Standard, V2.0. (b) Extended, V2.0B.

Table 12-11 CAN Message Frame Bits

Bit	Name	Function
Bus idle	–	A recessive state (logic high)
SOF	Start-of-frame	Acts as a start bit for the frame; inserted by the CAN hardware
11-bit identifier	Identifier bits	Standard or extended message frame identifier bits
RTR	Remote transmission request	Status of the remote transmission request in the CAN frame
SRR	Substitute remote request	Used only in extended format; SRR = 1 in transmit buffers and as received in receive buffers
r1, r0	—	Two dominant bits reserved for future use
IDE	ID extended	Identifies extended (= 1) or standard (= 0) format
18-bit identifier	Identifier bits	Extended format identifier bits
DLC	Data length code	Defines the data length (0–8 bytes)
Data field	Data bytes	Up to 8 bytes of data
15-bit CRC	Cyclic redundancy check	Error checking generated by the CAN hardware
ACK	Acknowledge	Inserted by the CAN hardware
EOF	End of frame	Acts as stop bit for the frame; inserted by the CAN hardware
IFS	Interframe space	At least three recessive bit times are required after the frame and before the next frame to allow internal processing in the nodes

hardware detects a bit change outside its limits, it can adjust the sample point by adding or subtracting time quanta.

CAN Message Receiving

The CAN receiver detects the start-of-frame bit and starts to clock message bits in a manner similar to that of the SCI receiver described earlier. The identifier bits in the message are used by the receiver to determine if it is a message for itself or if it should be ignored. Figure 12-36 shows how the receiver accepts or ignores CAN messages.

Although there are 11 or 29 identifier bits in the arbitration field of the message, the receiver may use 8-, 16-, or 32-bit patterns to identify its messages. As the message arrives, it is shifted into a message buffer and the identifier (shown here as 8 bits) is compared with an *identifier acceptance pattern*. When they match, all outputs of the exclusive-NOR gates in Figure 12-36 and the *filter hit* signal will be asserted. For added flexibility in identifying messages, an *identifier mask pattern* can allow some of the bits in the identifier to be don't cares. A one in the identifier mask pattern sets the acceptance bit to be a don't care. When the filter hit signal is asserted, the rest of the message is accepted by the receiver.

The receiver checks the 15-bit CRC code to make sure there were no errors in the transmission and then, at the correct time in the message (see Figure 12-35), asserts the *acknowledge (ACK)* bit to let the transmitter know the message was received correctly.

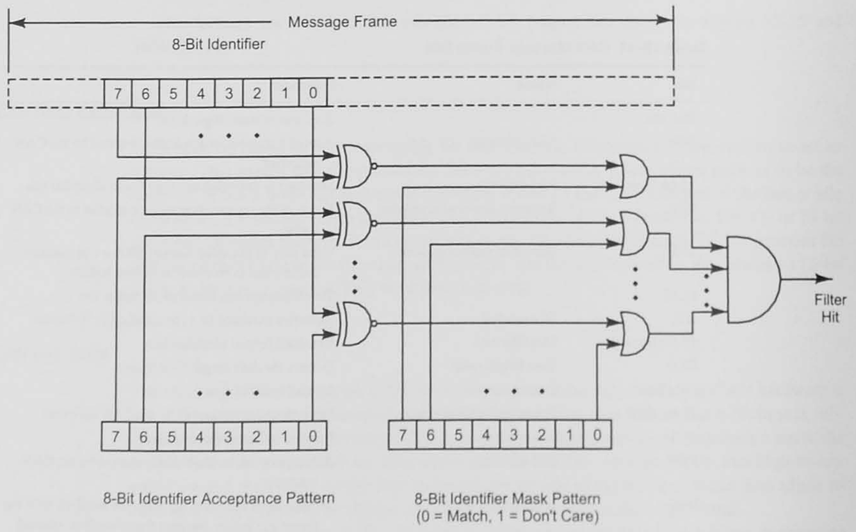

Message Frame
8-Bit Identifier
7 6 5 4 3 2 1 0
Filter Hit
7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8-Bit Identifier Acceptance Pattern
8-Bit Identifier Mask Pattern (0 = Match, 1 = Don't Care)

Figure 12-36 Receiver acceptance filtering.

CAN Message Transmission Speeds and Distances

The transmission bit rates and distances are interrelated and depend on the implementation of the physical layers. Table 12-12 shows bit rates and distances for typical CAN bus applications using terminated, twisted-pair bus wires.

12.13 Conclusion and Chapter Summary Points

Asynchronous Serial Communications

- A UART is a universal asynchronous receiver/transmitter. It sends and receives serial data.
- In microcontrollers a UART is often called an SCI (serial communications interface).
- The two logic states in asynchronous serial communication are called mark and space.
- The data sent starts with a start bit and ends with a stop bit.
- The start bit synchronizes the receiver with the transmitted data.
- The ASCII code is most often used for serial I/O when the data are alphanumeric characters.
- Any data rate may be used, but there are standard ones used for character I/O.
- Handshaking signals are defined for the RS-232-C interface.
- Data terminal equipment (DTE) and data communication equipment (DCE) are defined in the RS-232-C standard.
- Modems modulate and demodulate tones for telephone line communication.
- A null modem cable can connect two DTE devices.
- Control codes may be sent from a terminal by holding down the control key while typing another printable key.

Synchronous Serial Peripheral Interface

- The SPI provides a simple interface that allows I/O expansion to be easily accomplished.

Table 12-12 CAN Bus Length vs. Data Rate

Bus Length (m)	Bit Rate
40	1 Mbit/s
40–300	500 kbit/s
300–600	100 kbit/s
600–1000	50 kbit/s

- The SPI is a master/slave interface system.
- The master controls all data transfer between the master and the slave.
- The master generates a clock for the data transfer.
- Multiple slaves may be used as long as only one is selected by the SS_L signal at a time.

I²C Interface

- I²C stands for inter-integrated circuit.
- The I²C bus is a two-wire bus (three with ground reference); the data line is called SDA and the clock SCL.
- The interface supports multiple-master/multiple-slave architecture.
- A master controls all data transfer.
- Slaves have an address that is the first byte of any message that is transmitted by a master.
- An arbitration scheme operating on a bit-by-bit basis can resolve competition between two masters trying to access the bus at the same time.
- Although each I²C device can have its own clock, a clock synchronization scheme allows the slowest device to control the low time and the fastest device the high time of the SCL clock.

Controller Area Network (CAN) Bus

- The CAN bus was developed for automotive applications.
- It is a multiple-master/multiple-slave bus.
- The bus clock and bus clock synchronization are derived from the data.
- A message contains an identifier that specifies where the message should be received.
- A message filter allows receivers to accept only the messages destined for them.
- The arbitration scheme to resolve which of two simultaneously active masters is to be allowed to transmit is called carrier sense, multiple-access with collision detection (CSMA/CD).

12.14 Problems

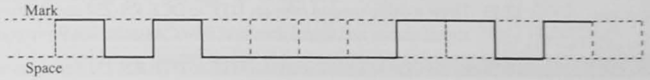
Explore

12.1 How does an asynchronous serial port achieve synchronization of the bits it is sending or receiving? [a]

12.2 An SCI is transmitting data at the baud rates given. The format is 8 data bits, no parity, and one stop bit. For each case, what is the maximum number of characters per second that can be transmitted? [a]

a. 56 kbaud
b. 9600 baud

12.3 A serial I/O port sends the following waveform: [a]



a. What is the ASCII character being sent?
b. What type of parity is being used?

12.4 Find a web-based ASCII code table. [a]

12.5 To initiate a serial data transfer, a UART first [a]

a. sends the least significant bit.
b. sends the start bit.
c. sends the stop bit.
d. sends the parity bit.
e. None of these.

12.6 Draw the waveform seen on the serial-data-out line when a UART uses 7 bits of data plus odd parity to send the ASCII character 'L'. [a]

12.7 How many bits per second (baud) is a serial port sending when the character rate is 120 characters per second? Assume ASCII characters with even parity. [a]

12.8 If the data rate is 9600 baud, at what rate can ASCII characters be sent, assuming 7 data bits and 1 parity bit? [a]

12.9 Define the following terms used in the CAN bus: [g, k]

a. Acceptance filter
b. CRC
c. CSMA/CD
d. Dominant level
e. Recessive level
f. Synchronization jump
g. CAN bus

12.10 Two computers are to be connected by means of their COM ports: [c]

a. For this to work, what operational parameters need to be specified?
b. In this application, what is meant by "data flow" synchronization?
c. What are two ways of achieving data flow synchronization?

Stimulate

12.11 You are to define a serial cable to connect two PCs configured as RS-232 DTE devices. Each PC has a DE9P connector on its back panel. The software used in each

PC for file transfer uses hardware (RTS/CTS) flow control. Draw an appropriate cable using the *minimum* number of wires. Be sure to show each connection, give the signal name, tell the data flow directions, and state what connectors are to be used on each end of the cable. [c]

12.12 How does the receiver in a UART maintain its synchronization with the transmitter in asynchronous operation? [a]

12.13 Draw a cable used to connect DTE to DCE RS-232 serial devices. Show pins 1–9 with signal names and signal direction flows. Assume that 9-pin connectors are used. [a]

12.14 Draw a cable used to connect DTE to DTE RS-232 serial devices. Show pins 1–9 with signal names and signal direction flows. Assume that 9-pin connectors are used. [a]

12.15 Why is the RS-232 voltage specification for mark and space logic levels used for serial communications voltage levels instead of TTL? [a]

12.16 How does a slave station SPI send data to the master station? [g]

Challenge

12.17 An SCI is transmitting data at 19.2 kbaud. The format is seven data bits, even parity, one stop bit. How long does it take to send a document that is one megabyte long? [a]

12.18 The clock shown Figure 12-2 connecting to the receiver serial-in/parallel-out shift register is often 16 or 64 times the basic baud rate. Why do you suppose this is so? [a]

12.19 You are to define a serial cable to connect a PC configured as an RS-232 DTE device to a microcontroller system configured as a DCE device. The PC has a DE9P connector on its back panel, and the embedded system uses a DE9S connector. There is no flow control for the data transfer between the two computers. Draw an appropriate cable using the *minimum* number of wires. Be sure to show each connection, give the signal name, tell the data flow direction, and state what connectors are to be used on each end of the cable. [c]

12.20 A system is to be designed to transfer serial data from one place to another over a distance of 200 feet. Data is to be transferred in one direction only, and there is no data flow problem. Data transfer rate is to be a minimum of 100 kbit/s. You are to compare an asynchronous serial port approach (SCI) with a synchronous serial port (SPI) approach. [b, c]

a. How many wires will be needed to connect the two systems (including the ground wire)?
b. For this distance and data rate, what signaling interface standard would you propose?

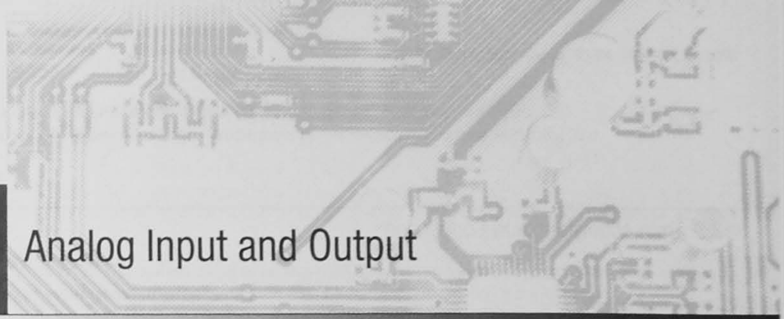
Reflect on Learning

12.21 What questions do you still have about asynchronous serial communications?

12.22 What questions do you still have about the synchronous peripheral interface?

12.23 What questions do you still have about the I2C serial bus?

12.24 What questions do you still have about the CAN bus?



13 Analog Input and Output

Objectives

In this chapter we consider the world of analog signals. Computers must read analog information and act upon it in many applications. This requires an analog-to-digital converter. In other situations, an analog output signal may be required; this calls for a digital-to-analog converter. In this chapter we will discuss both devices and learn how to specify the correct one for the job to be done.

13.1 Introduction

Analog input and output converters allow us to process continuous signals as functions of time. There are many reasons to do this. One of the numerous advantages of digital signal processing over analog is that once an analog signal has been converted to digital values, it is generally free from additional noise. Audiophiles recognize this feature when playing their CDs. The music is recorded digitally and is converted to an analog signal for playback. Dust and dirt do not corrupt the digital signal as with LP records or audiotape. Another reason for converting to the digital domain is that once signals have been digitized, they can be manipulated by the computer, often to produce effects that are unachievable by means of analog signal processing. In medical imaging, computerized tomography, or CT, scans are produced by manipulating digital images. Today, we are seeing a great migration away from analog delivery of information. Digital television and digital telephone services are now readily available. With the increased availability of broadband networks, such as optical fibers, even more digital data will be at our fingertips.

Digitizing signals does have drawbacks. We can never exactly represent or reconstruct the analog signal. There will always be some error, but it can be minimized with good system design. A digitized signal, when transmitted over a communication channel, requires a greater bandwidth than the original channel. For example, a normal analog voice telephone circuit channel requires a bandwidth of about 4 kHz. The equivalent digital channel is 64 kbit/s. The extra bandwidth is justified by the ability to enhance the signal, to repeat it over long distances without degradation, and by the opening of the communication channel to other digital services such as data transfer between computers.

This chapter covers analog-to-digital (A/D) and digital-to-analog (D/A) conversion. We will learn how to specify a converter for a particular application and how a variety of A/Ds and D/As work.

13.2 Data Acquisition and Conversion

Figure 13-1 shows a *data acquisition system* to input analog data. It receives analog information from a physical variable, such as temperature, and uses a *transducer* to convert the information to an electrical signal, either voltage or current. Following the transducer is a block labeled *signal conditioning* to provide the following functions:

Amplification: Rarely does the transducer produce the voltage or current needed by the A/D. The amplifier is designed so that the full-scale signal from the analog input results in a full-scale signal to the A/D.

Bandwidth limiting: The signal conditioning provides a low-pass filter to limit the range of frequencies that can be digitized. To understand why this is so, we will consider the sampling theorem and learn about aliasing in Section 13.3.

Isolation and buffering: The input to the A/D may need to be protected from dangerous voltages such as static discharges or reversed polarity voltages.

An *analog multiplexer* follows the signal conditioning in applications that call for the digitization of several analog inputs. This computer-controlled switch allows multiple analog inputs, each with its own signal conditioning for different transducers, to be switched into a single A/D. The CPU generates an address on the multiplexer select lines to select the multiplexer channel.

Data Acquisition System Operation

The operation of the system shown in Figure 13-1 can be described as follows.

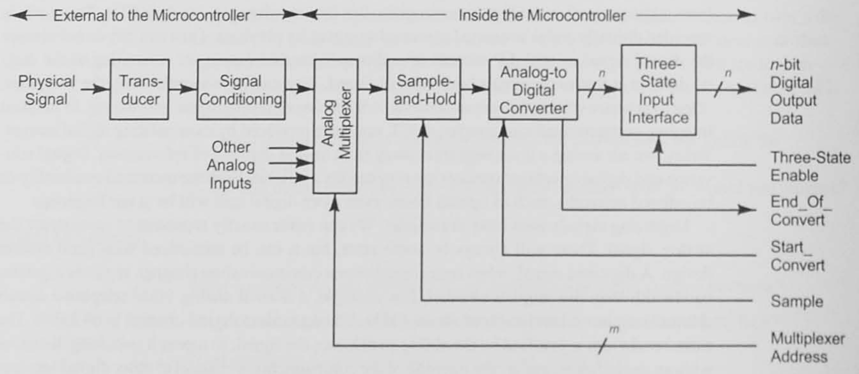


Figure 13-1 Data acquisition system.

- The program selects the analog signal to be digitized by outputting an address to the analog multiplexer.
- As we will discuss shortly, a sample-and-hold circuit may be needed to hold the analog signal constant while the analog-to-digital converter is working. In such cases, the program asserts the *sample* signal to take a quick snapshot of the analog signal.
- Following the sample-and-hold action, the program asserts the *Start_Convert* signal to start the A/D.
- When the A/D finishes the conversion, it asserts the *End_Of_Convert* signal, which allows the program to input the data through the three-state input interface.

Definitions

Let us define some of the terms you will encounter as you learn about the analog-to-digital conversion. We begin with a fundamental concept and illustrate it in Example 13-1.

Resolution: The resolution is the smallest change in the input analog signal that will produce a change in the output digital code:

$$V_{\text{resolution}} = \frac{V_{\text{full-scale value}}}{2^n}\ V$$

Resolution is also stated in terms of the number of bits in the output digital code, or as one part in 2^n. Sometimes the resolution is given as a percentage of maximum or full-scale value:

$$V_{\text{resolution}} = \frac{1}{2^n} \times 100\% \text{ of full-scale value}$$

Example 13-1 A/D Resolution

An 8-bit A/D converter is to digitize a 5 volt, full-scale signal. What is the resolution?

Solution

The resolution is 5/256 = 19.5 mV. Another way of stating the resolution is 1 part in 256, or 0.4% of the full-scale value.

Additional definitions follow.

Accuracy: Accuracy is often confused with resolution. Resolution relates the smallest signal (or noise) to the full-scale value. Accuracy relates the smallest signal to the measured signal.

Accuracy is given as a percentage and describes how close the measurement is to the actual value. We say the digital representation of our signal is accurate to within

$$\frac{V_{\text{resolution}}}{V_{\text{signal}}} \times 100\%$$

See Example 13-2.

A/D transfer function: The A/D transfer function shows the digital output code as a function of the input voltage. Figure 13-2 shows a transfer function for a 3-bit A/D. As the voltage increases from zero, the output code steps up one bit at a time, ranging from 000 to 111.

Example 13-2 A/D Accuracy

An 8-bit A/D converter digitizes a 5-volt, full-scale signal. What is the accuracy with which the A/D can digitize the following signals?
50 mV, 1 V, 2.5 V, 4.9 V

Solution

The resolution is 5 V/256 = 19.5 mV. Each measurement will be accurate to within the listed percentage value:

50 mV	(19.5 mV/50 mV) = 39%
1 V	(19.5 mV/1 V) = 1.9%
2.5 V	(19.5 mV/2.5 V) = 0.8%
4.9 V	(19.5 mV/4.9 V) = 0.4%

Aperture time: This is the time that the A/D converter is looking at the input signal. It is usually equal to the conversion time. We will see in the next section how any change in the input signal during this time may cause an error in the output code.

Conversion time: The conversion time is the time required to complete a conversion of the input signal. It establishes the upper signal frequency limit that can be sampled without aliasing:

$$f_{\max} = \frac{1}{2 \times \text{conversion time}}$$

Dynamic range: The dynamic range of a signal is the ratio of the maximum signal to the smallest, or noise level, signal.

$$\text{Dynamic range} = \frac{V_{\max}}{V_{\text{noise}}}$$

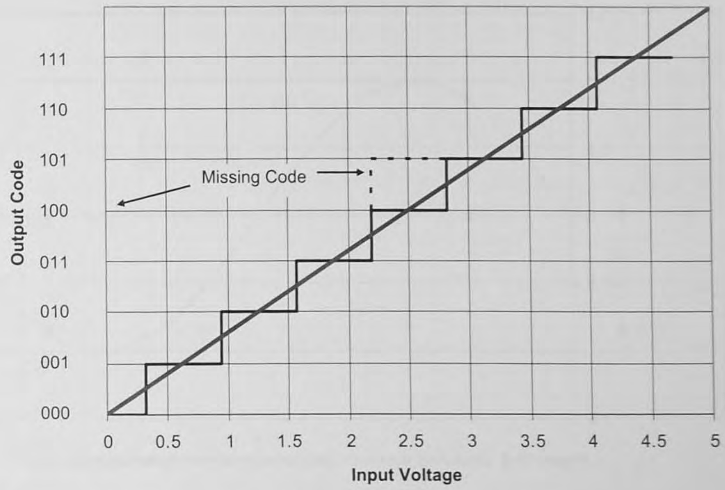


Figure 13-2 A 3-bit A/D transfer function.

Dynamic range may also be stated in decibels.

$$\text{Dynamic range} = 20 \log \frac{V_{\max}}{V_{\text{noise}}} \text{ dB}$$

Linearity: Linearity is the deviation in output codes from a straight line drawn through zero and full scale. The best that can be achieved is ±0.5 of the least significant bit (±0.5LSB), as shown in Figure 13-2.

Missing codes: The transfer function for a converter with a missing code is shown in Figure 13-2. An internal error might be the cause of a missing code.

13.3 Shannon's Sampling Theorem and Aliasing

The frequency at which signals are sampled must be at least two times the highest frequency in the signal.

Claude Shannon showed that when a signal, $f(t) = X \sin 2\pi f_{\text{sig}} t$, is to be sampled (digitized), the *minimum sampling frequency must be twice the signal frequency.*[1] Consider the waveform in Figure 13-3 whose frequency is f_{sig}. When the sampling frequency f_{sample}, is twice f_{sig}, the waveform is sampled at points A and B. The problem we now pose, and

[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Sys. Tech. J.*, vol. 27, 1948, pp. 379–423.

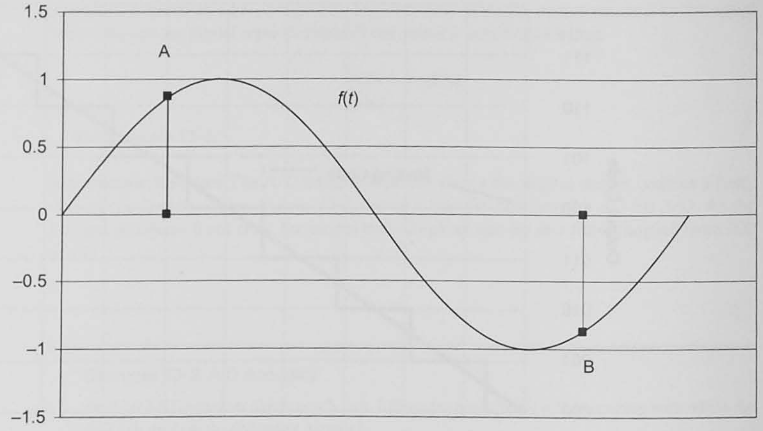


Figure 13-3 Sinusoidal waveform sampled at twice the signal frequency.

the problem that digital signal processors must solve when *reconstructing* a waveform from sampled data, is this:

Given the two samples A and B as shown in Figure 13-4, find a sinusoidal waveform to fit. You may adjust the frequency, the amplitude, and the phase, and you may assume that when the samples were taken, the sampling criteria were satisfied. That is, there are no frequencies higher than half f_{sample}.

By observing that the two samples are equal in magnitude and opposite in sign, we can convince ourselves that the frequency we are trying to reconstruct is $f_{sample}/2$. By adjusting the amplitude and the phase, we can find the correct solution $f(t) = X \sin 2\pi(f_{sample}/2)t = X \sin 2\pi f_{sig} t$. If there are more than two samples per period, the reconstruction is easier. We conclude that in digitizing signals according to Shannon's sampling theorem, the input signal can be reconstructed from the digital values.

Signals that are *undersampled* cause *aliasing.*

Now consider the following scenario. Assume that we are sampling the input signal at the sample frequency f_{sample} and that the signal $f_1(t) = Y \sin 2\pi f_{1sig} t$ (Figure 13-5 solid line), which is a little higher in frequency than $f(t)$, is present. Because $f_1(t)$ is *undersampled*, that is, not sampled fast enough, our digital signal processor has a dilemma. The digital values are again A and B, equal in magnitude and opposite in sign. Working only from the digital values, the digital signal processor *must assume* that the sampling criterion has been met, and so $f(t)$ is reconstructed, not $f_1(t)$. This is an example of *aliasing*. The second signal, $f_1(t)$ is higher frequency than $f_{sample}/2$, and undersampling a waveform makes it appear as if it were a *lower frequency*. The signal $f_1(t)$ is an *alias* for $f(t)$, and this causes an error in the signal reconstruction. To avoid aliasing in all A/D converters, the sampling frequency must be at least twice the highest frequency in the signal. The signal conditioning stage in Figure 13-1 must contain what is called an *antialiasing* filter, to pass only low frequencies and attenuate frequencies above one-half the sampling frequency. The maximum frequency that one can sample without aliasing, $f_{sample}/2$, is called the *Nyquist* frequency. See Example 13-3.

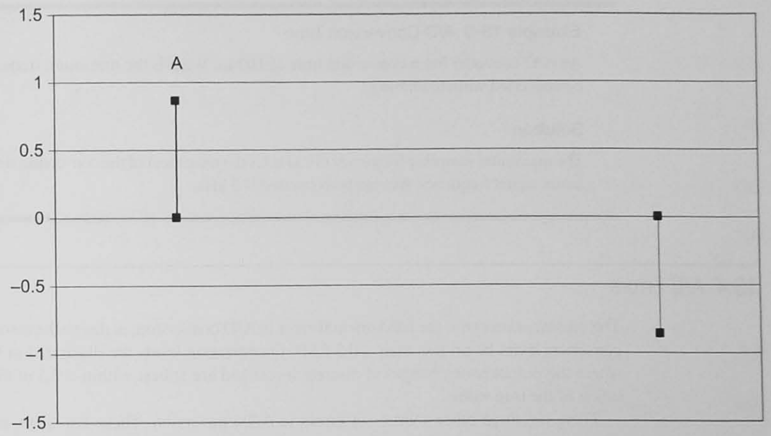


Figure 13-4 Sampled waveform.

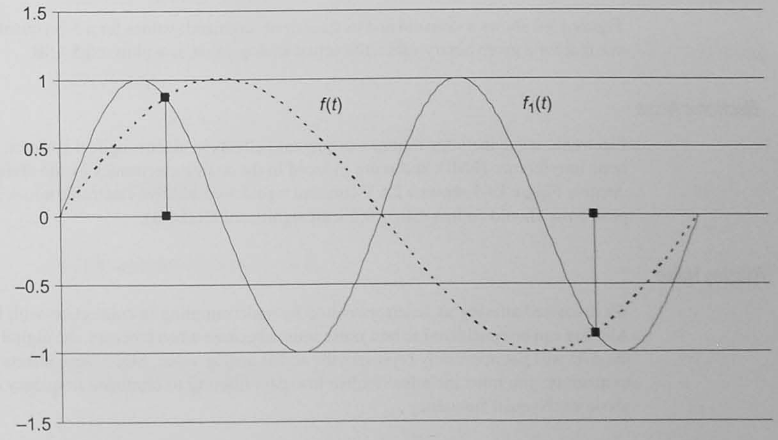


Figure 13-5 Undersampled waveform.

Example 13-3 A/D Conversion Time

An A/D converter has a conversion time of 100 μs. What is the maximum frequency that can be converted without aliasing?

Solution

The maximum sampling frequency (10 kHz) is the reciprocal of the conversion time. The maximum signal frequency that can be converted is 5 kHz.

13.4 A/D Errors

The *quantization error*, the fundamental error in A/D conversion, is due to the resolution of the converter; it can be no less than ±0.5 LSB. Quantization levels are illustrated in Figure 13-2, where the output code changes at discrete levels and are at best within ±0.5 of the A/D resolution of the true value.

There are three other sources of errors in A/D conversion. These are *noise*, *aliasing*, and *aperture error*. We would like all of these to be less than the basic quantization error.

Quantization Error

Figure 13-6 shows a sinusoid and its quantized (digitized) values for a 3-bit quantizer. We can see that for a given binary value, the actual analog value, is within ±0.5 LSB.

Electronic Noise

Electronic noise includes shot noise, quantum effects in electro-optical systems, electromagnetic interference (EMI), and noise induced in the analog electronics by the digital switching circuits. Figure 13-7 shows a 2.5 V constant signal with additive electronic noise. The peak-to-peak noise should be less than ±0.5 least significant bit (LSB).

Aliasing Noise

We discussed aliasing as an error created by undersampling in connection with Figure 13-5. Aliasing can be considered to be a noise source because when it occurs, the digital values from the A/D will not accurately represent the actual analog value. Since these effects are difficult to quantify, you must include effective low-pass filtering to eliminate frequency components above the Nyquist frequency.

Aperture Error

A significant error in a digitizing system is due to signal variation during the time the signal is sampled. This period, called the *aperture time*, limits the maximum frequency that can be sampled. Aperture error is shown in Figure 13-8, where the signal is changing when the aperture is open. A good design will attempt to have the uncertainty Δv be less than one least significant bit. We can derive a design equation for the aperture time t_{ap} in terms of the

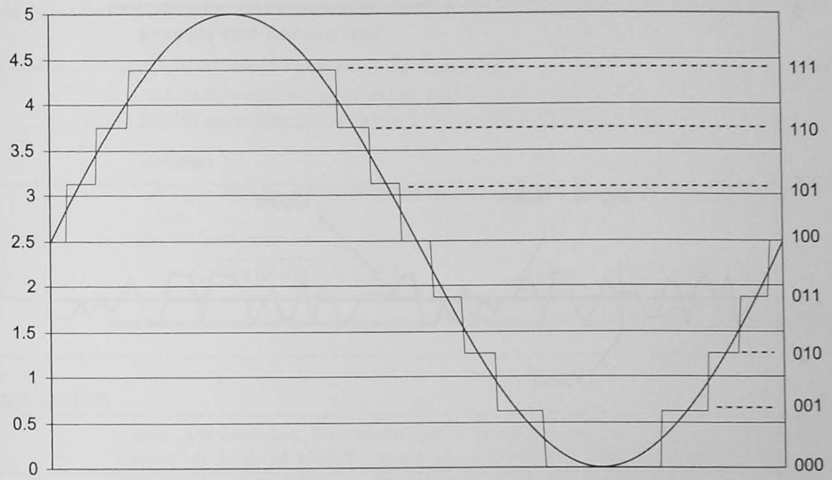


Figure 13-6 Quantized sinusoid.

maximum signal frequency f_{max} and n, the number of bits in the A/D converter, by observing the following:

$$v(t) = V_{max} \sin 2\pi f_{max} t$$
$$\Delta v = 2\pi f_{max} V_{max} \cos(2\pi f_{max} t) \;\; \Delta t = 2\pi f_{max} V_{max} \cos(2\pi f_{max} t) t_{ap}$$

for $t = 0$ (worst case slope)

$$\Delta v = 2\pi f_{max} V_{max} t_{ap}$$

and for Δv to be less than one LSB,

$$\frac{\Delta v}{V_{max}} = \frac{1}{2^n} = 2\pi f_{max} t_{ap}$$

Solving for the aperture time, we write

$$t_{ap} = \frac{1}{2\pi f_{max} 2^n}$$

The maximum frequency that can be converted with aperture errors less than ±0.5 least significant bit is given by

$$f_{max} = \frac{1}{2\pi t_{ap} 2^n}$$

Table 13-1 shows this effect. See Example 13-4.

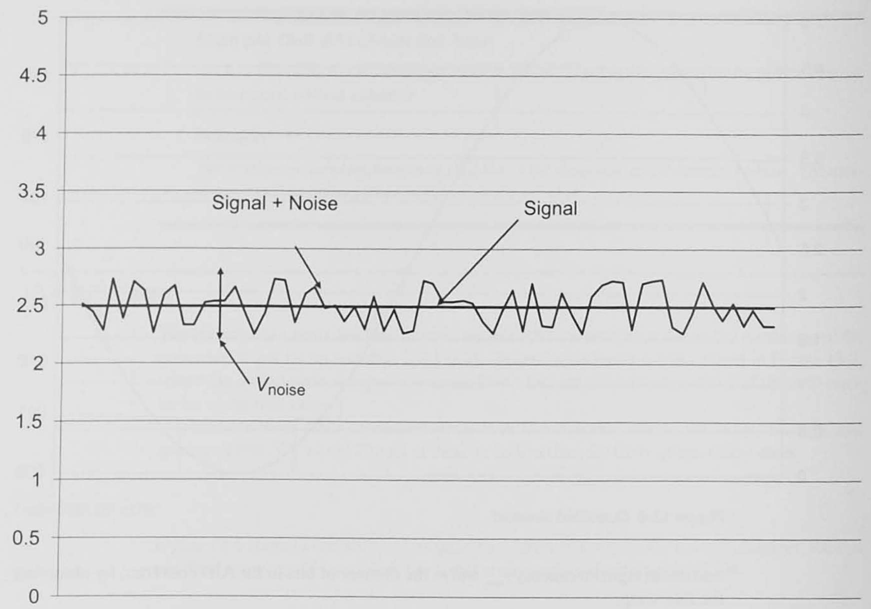


Figure 13-7 Analog signal plus noise.

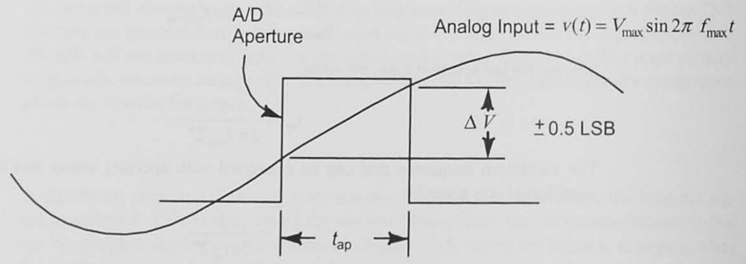


Figure 13-8 Aperture time error.

Example 13-4 Aperture Time

A 1 kHz sinusoidal signal is to be digitized to 8 bits.

(a) Find the maximum conversion time that can be used and still avoid aliasing.
(b) Find the aperture time so that the aperture error is less than ± 0.5 LSB.

Solution

(a) There must be at least two samples per period; so the maximum conversion time is 0.5 ms.
(b) The aperture time is

$$t_{ap} = 1\sqrt{(2\pi)(10^3)(256)} = 0.62\ \mu s$$

Sample-and-Hold

In many A/D converters, the aperture time is the same as the conversion time. The A/D is "looking" at the signal while it is converting it. In Example 13-4 the conversion time for a 1 kHz signal is 0.5 ms, while the aperture time is 0.62 µs. The aperture time is the more restrictive specification; it would be much more expensive, however, to buy a converter with a conversion time of 0.62 µs just to satisfy the aperture time requirements. A *sample-and-hold* (*S/H*) circuit, also called a *track-and-hold* circuit (Figure 13-9), was included in the design shown in Figure 13-1. Such a circuit can achieve the short aperture time while allowing a less expensive converter to satisfy the conversion time.

Table 13-1 Effect of Aperture Time

t_{ap}	n	f_{max}
1 µs	8	622 Hz
1 µs	10	155 Hz
1 ms	8	0.622 Hz
1 ms	10	0.155 Hz
1 ns	8	622 kHz
1 ns	10	155 kHz
1 ns	12	38.9 kHz

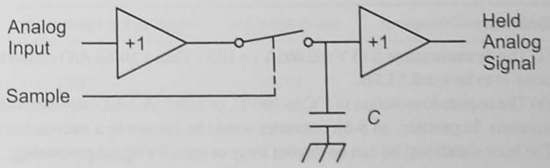


Figure 13-9 Sample-and-hold circuit.

13.5 Choosing the A/D Converter

The designer must choose the number of bits, or resolution, and the speed, or conversion time, of the converter. The type of digital code output from the converter may be chosen. The aperture time must be calculated and a decision made to include a sample-and-hold and an antialiasing filter in the system.

Choosing the A/D Resolution

There are two ways to find the resolution needed in the A/D. The first is to find the dynamic range of the input signal and to choose the number of bits based on this. The *dynamic range* of any signal is given by

$$\text{Dynamic range} = \frac{V_{\text{max}}}{V_{\text{noise}}}$$

where V_{max} is the maximum input signal and V_{noise} is the noise. We would like the noise to be within ± 0.5 LSB, as shown in Figure 13-7, and for this to be true, the number of bits is

$$N \geq \log_2 \frac{V_{\text{max}}}{V_{\text{noise}}}$$

This is the best one can do unless signal processing, such as averaging, can reduce the noise.

Another way to choose the number of bits is based on the resolution required in the signal. Here, V_{min} is the required resolution, and it determines the number of bits by means of the following relation (see Example 13-5):

$$N \geq \log_2 \frac{V_{\text{max}}}{V_{\text{min}}}$$

Example 13-5 Choosing the A/D Resolution

A transducer is to be used to find the temperature over a range of zero to 100°C. We are required to read and display the temperature to a resolution of ± 1°C. The transducer produces a voltage from 0 to 5 V over this temperature range with 5 MV of noise. Specify the number of bits in the A/D converter (a) based on the dynamic range of the signal and (b) based on the required resolution.

Solution

(a) The dynamic range is (5 V)/(0.005 V) = 1000. Thus a 10-bit A/D converter is required if the noise is to be < ± 0.5 LSB.

(b) The required resolution is 1°C in 100°C, or 100:1. A 7-bit converter will meet these specifications. In practice, an 8-bit converter would be chosen in a microcontroller-based system. The least significant bit can be thrown away or used for signal processing.

Choosing the A/D Conversion Time

The A/D conversion time is chosen by considering potential signal aliasing. The highest frequency component in the signal must be sampled at least twice in a period. A design equation for conversion time is given by

$$\text{A/D conversion time} \leq \frac{1}{2 f_{\text{max}}}$$

See Example 13-6.

Example 13-6 Choosing the Conversion Time

Find the maximum conversion time for an A/D converter to digitize the following signals: 1 kHz sinusoid, 1 Hz sinusoid, 1 MHz sinusoid, a video signal with a bandwidth limited to 5 MHz

Solution

1 kHz – 500 μs; 1 Hz – 500 ms; 1 MHz – 500 ns; 5 MHz video – 100 ns

Choosing the Output Code

The output code may be chosen at the time of specifying the A/D. Different codes are available, depending on the input signal. For unipolar devices, unsigned binary or complement binary codes are available, as shown in Table 13-2.

A bipolar-input A/D must encode negative and positive signals. Table 13-3 shows a variety of coding schemes.

Choosing a Sample-and-Hold

The specification for the aperture time usually requires a sample-and-hold if the signal has any time varying components. A separate sample-and-hold may be used, although some sampling A/D converters have the sample-and-hold built in. See Example 13-7.

Table 13-2 8-Bit Binary Codes for Unipolar A/D

Percentage of Full-Scale	+10 V Full-Scale	Unsigned Binary	One's-Complement Binary
0	0.000	0000 0000	1111 1111
0 + 1 LSB	+0.039	0000 0001	1111 1110
25%	+2.500	0100 0000	1011 1111
50%	+5.000	1000 0000	0111 1111
75%	+7.5000	1100 0000	0011 1111
Full scale – 1 LSB	+9.961	1111 1111	0000 0000

Table 13-3 8-Bit Binary Codes for Bipolar A/D

Percentage of Full-Scale	±5 V Full-Scale	Two's Complement	Signed Magnitude	Offset Binary
–(Full-scale)	–5.00	1000 0000	None	0000 0000
–(Full-scale) + 1 LSB	–4.96	1000 0001	1111 1111	0000 0001
–75%	–3.75	1010 0000	1110 0000	0010 0000
–50%	–2.50	1100 0000	1100 0000	0100 0000
–25%	–1.25	1110 0000	1010 0000	0110 0000
–1 LSB	–0.04	1111 1111	1000 0001	0111 1111
0	0	0000 0000	1000 0000 and 0000 0000	1000 0000
+1 LSB	+0.04	0000 0001	0000 0001	1000 0001
+25%	+1.25	0010 0000	0010 0000	1010 0000
+50%	+2.50	0100 0000	0100 0000	1100 0000
+75%	+3.75	0110 0000	0110 0000	1110 0000
Fullscale – 1 LSB	+4.96	0111 1111	0111 1111	1111 1111

Example 13-7 Choosing the Sample-and-Hold

For each of the signal in Example 13-6, give the required aperture time for an 8-bit A/D converter.

Solution

1 kHz – 0.62 μs; 1 Hz – 0.62 ms; 1 MHz – 0.62 ns; 5 MHz video – 0.12 ns

Choosing the Antialiasing Filter

The cutoff frequency (–3dB point) should be set to either the maximum frequency expected in the signal or to one-half the sampling frequency, whichever is lower. The order of the filter (the number of poles), which specifies the attenuation in decibels per decade, depends on the nature of the input signal and on how much energy is above the Nyquist frequency. See Example 13-8.

Example 13-8 Specifying the Complete A/D

Specify the A/D converter maximum conversion time, number of bits, cutoff frequency for the antialiasing filter, and the aperture time to digitize the following signal:

± 5 V peak-to-peak, 5 mV peak-to-peak noise, $f_{max} = 3$ kHz

Solution

The dynamic range is (10 V/5 mV) = 2000. Therefore $N \geq \log_2 2000 \geq 10.9$, and the number of bits $N = 11$. The maximum conversion time to prevent aliasing is $1/(2 \times 3000) = 170$ μs. The antialiasing filter should have a cutoff frequency of 3 kHz. The aperture time of the sample-and-hold is 26 ns.

13.6 The Analog-to-Digital Converter Interface

Figure 13-10 shows a typical A/D interface circuit. This particular circuit is for a sensor that produces a voltage in the range of 0 to 2.9 V. The Texas Instruments OPA4344 is a quad (there are four operational amplifiers in one package), rail-to-rail amplifier. This means it can operate on a single 5-volt power supply and still achieve an output voltage of nearly 5 V. The gain of this noninverting amplifier is

$$A_V = \frac{R_1 + R_2}{R_1} = 2.8$$

The gain of the input voltage divider is

$$\frac{R_6}{R_4 + R_5 + R_6} = 0.62$$

giving an overall gain of 1.7 for dc. A two-pole, low-pass filter is given by R_4, C_1 and R_5, R_6, C_2. We can treat the two filter sections independently because the coupling resistor, R_5, is sufficiently high to isolate the second filter from the first.

The cutoff frequency of the R_4, C_1 is

$$f_{C1} = \frac{1}{2\pi R_4 C_1} = 723 \text{ Hz}$$

The cutoff frequency of R_5, R_6, and C_2 is

$$f_{C2} = \frac{1}{2\pi(R_5 / R_6)C_2} = 775 \text{ Hz}$$

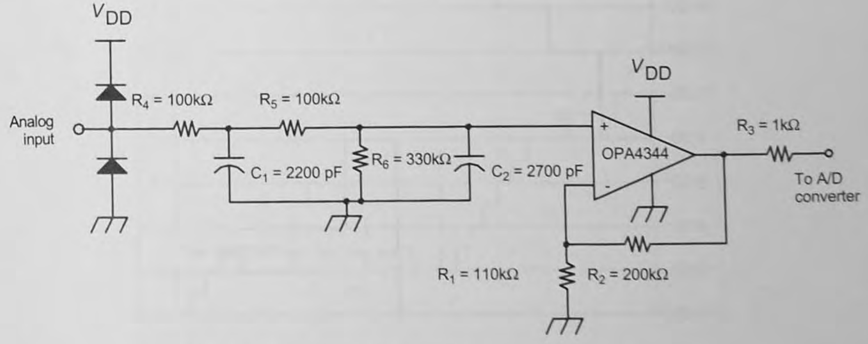


Figure 13-10 Analog-to-digital converter electronics.

13.7 Analog-to-Digital Converter Types

There are a number of analog-to-digital converter types. The one chosen depends on the application and on the performance required.

Successive Approximation A/D

Perhaps the most widely used A/D converter is the *successive approximation* type shown in Figures 13-11 and 13-12. Each bit in the successive approximation register is tested, starting at the most significant and working toward the least significant. As each bit is set, the output of the digital-to-analog (D/A) converter is compared with the input. If the D/A output is lower than the input signal, the bit remains set and the next bit is tried. Bits that make the D/A output

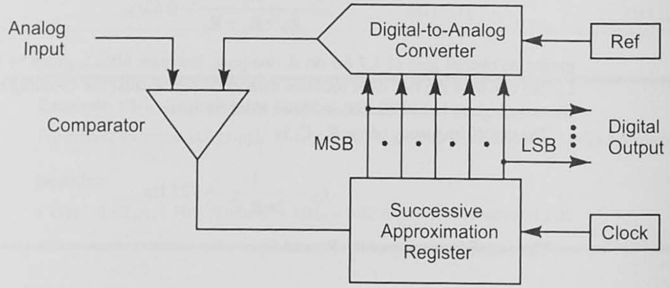

Analog Input
Digital-to-Analog Converter
Ref
Comparator
MSB
LSB
Digital Output
Successive Approximation Register
Clock

Figure 13-11 Successive approximation A/D.

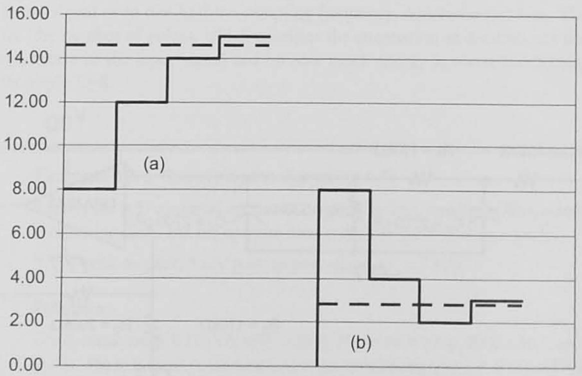


Figure 13-12 Successive approximation D/A output: solid lines D/A output; dashed lines, 14.6 V (a) and 2.8 V (b).

higher than the analog input are reset. Setting and testing each bit in the successive approximation register requires *N* bit times.

Tracking A/D

The *tracking A/D converter* is shown in Figures 13-13 and 13-14. This close cousin of the successive approximation converter has an up/down counter controlled by the comparator. If the input signal is higher or lower than the output of the D/A converter, the counter counts up or down, respectively. This converter may quickly converge to the correct digital value when the signal is not changing rapidly. If there are large, rapid input changes, the counter may have to count through its full range before reaching the final value.

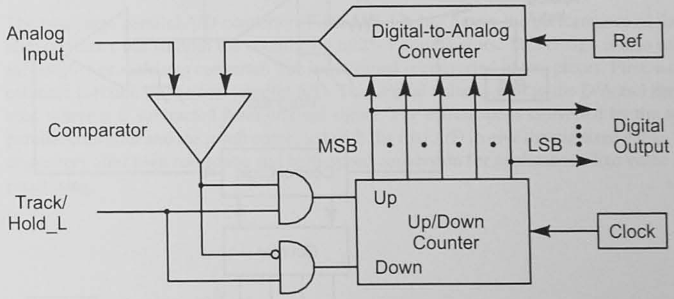


Figure 13-13 Tracking A/D.

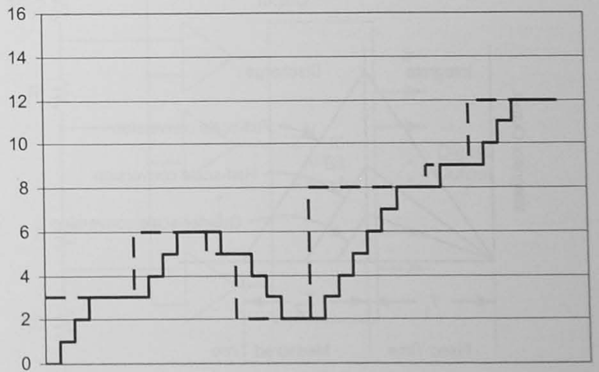


Figure 13-14 Tracking converter D/A output: dashed line, signal; solid line, D/A output.

Dual-Slope A/D

A *dual-slope* A/D can have very high rejection of periodic noise.

An interesting and useful converter is the *dual-slope* or *integrating A/D converter* (Figure 13-15.) The converter integrates the input signal for a fixed time, T_1, with higher input signals integrating to higher values. During the second period, T_2, the switch is changed to the minus reference voltage and the integrator discharges to zero at a constant rate. The time needed to discharge, T_2, gives the digital value.

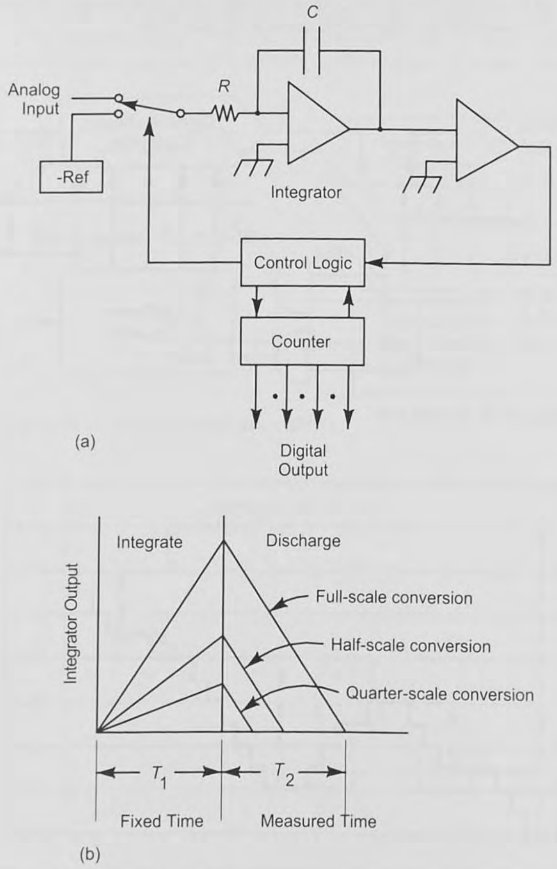


Figure 13-15 Dual-slope A/D.

The dual-slope integrating A/D is remarkably efficient at recovering signals from periodic noise. A common problem in many applications is 60 Hz noise from power lines. By making T_1 equal to the period of the interference (1/60th of a second) the positive half-cycle of interference is canceled by the negative half-cycle.

Parallel or Flash A/D

Figure 13-16 shows a parallel, or *flash*, A/D converter. This array of 2^N–1 comparators produces an output code in the propagation time of the comparators and the output decoder. Thus it is very fast, but also more costly in comparison to other designs.

Two-Stage Parallel A/D

The two-stage parallel A/D converter (Figure 13-17) has nearly the performance of the parallel converter but without the complexity of 2^N–1 comparators. This design is also called a *subranging* or *multistep* converter. The input signal is converted in two pieces. First, a coarse estimate is found by the first parallel A/D. This digital value is sent to the D/A and the summer, where it is subtracted from original signal. The difference is converted by the second parallel converter and the result combined with the first A/D to give the digitized value. These converters offer high resolution and high-speed conversion for applications like video signal processing.

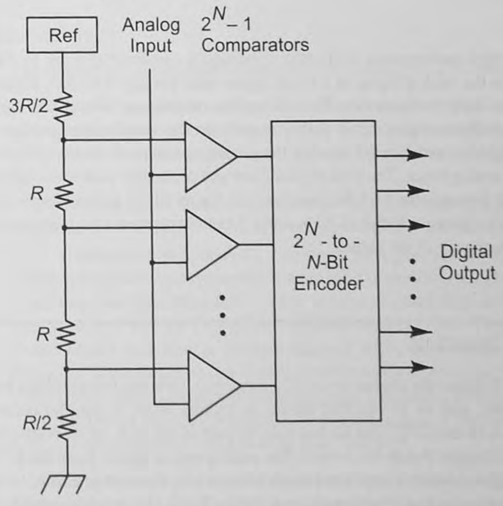


Figure 13-16 Parallel or flash A/D.

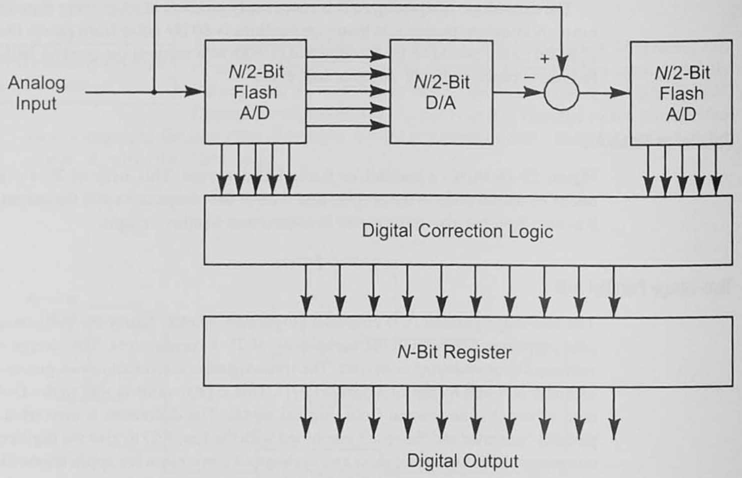


Figure 13-17 Two-stage parallel A/D.

Sigma-Delta A/D

A popular high-performance A/D is the *sigma-delta* converter (Figure 13-18). This converter oversamples the analog signal at a much higher rate, perhaps 64 times higher, than a conventional A/D's Nyquist frequency. This makes the antialiasing filter much easier to design and implement without degrading its ability to perform the antialiasing function. The comparator is a 1-bit digitizer, and for each sample, the preceding sample's analog value is subtracted from the current analog input. The final digital filter and decimator performs another low-pass filter operation. It averages the 64 1-bit samples (in a digital filter) and produces the final N-bit digital output. This process is called *decimation*. Decimation trades high-frequency, 1-bit samples for lower frequency, N-bit values.

13.8 Digital-to-Analog Conversion

Figure 13-19 shows the *digital-to-analog converter*. Few microcontrollers have an integrated D/A converter, and so an external device is usually used. A parallel output interface connects the D/A to the CPU. The latches may be part of the D/A, or an output interface like that designed in Chapter 9 may be chosen. The analog output signal from the D/A is quantized as shown in Figure 13-6. A signal conditioning block may be used as a filter to smooth the quantized nature of the output. The signal conditioning block also provides isolation, buffering, and voltage amplification if needed.

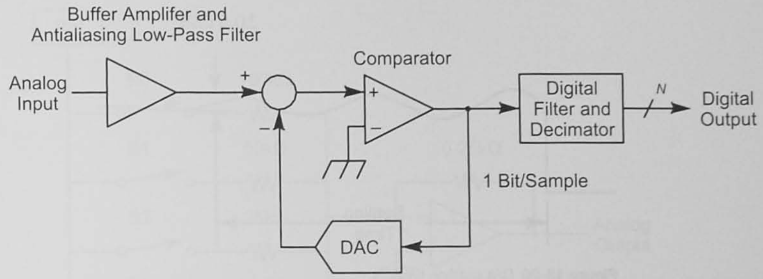


Figure 13-18 Sigma-Delta A/D.

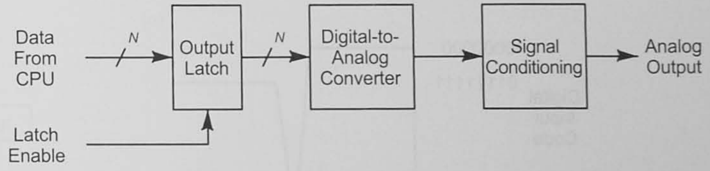


Figure 13-19 Digital-to-analog converter.

D/A Converter Specifications

The following concepts are essential for developing D/A converter specifications.

1. **Resolution and linearity:** The *resolution* is determined by the number of bits and is given as the output voltage corresponding to the smallest digital step (i.e., 1 LSB). The *linearity* shows how closely the output voltage follows a straight line drawn through zero and full-scale.

2. **Settling time:** This is the time taken for the output voltage to settle to within a specified error band, usually ± 0.5 LSB. Settling time is shown in Figure 13-20.

3. **Glitches:** High-speed D/A converters have glitches as well as settling time problems. A glitch is caused by asymmetrical switching in the D/A switches. If a switch changes from a one to a zero faster than from a zero to a one, a glitch may occur. Consider changing the output code of an 8-bit D/A from 10000000 to 01111111. These codes are adjacent, and we would expect the output to go from one-half full-scale to one resolution value less than that. However, if the switches can switch faster from a one to a zero than from a zero to a one, the output code will go through a transitory state sequence 10000000 to 00000000 to 01111111. This results in a short but sometimes noticeable glitch in the output signal. (Figure 13-21). Glitches are especially noticeable in video displays.

D/A converter glitches can be eliminated by following the D/A with a sample-and-hold, as shown in Figure 13-22. The S/H is strobed to sample the data after the glitch has occurred and after the D/A settling time.

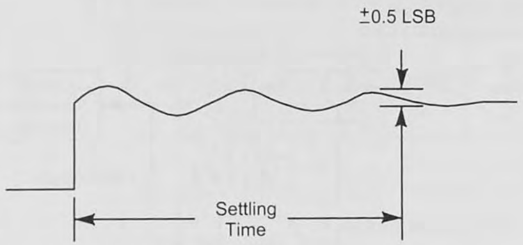


Figure 13-20 D/A settling time.

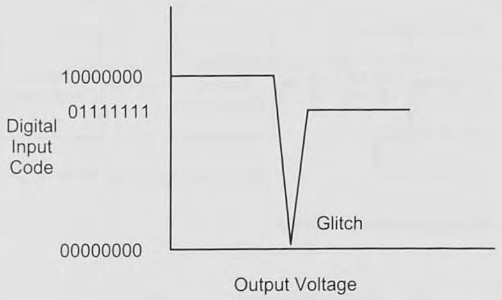


Figure 13-21 D/A output glitch.

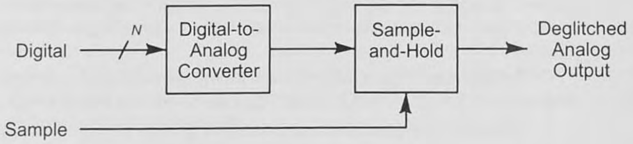


Figure 13-22 Deglitched D/A.

D/A Converter Types

We mention briefly three widely used D/A converter designs.

1. **Binary-weighted:** The most basic circuit is the *binary-weighted register D/A* (Figure 13-23a). As the switches for the bits are closed, a weighted current is supplied to the summing junction of the amplifier. For high-resolution D/A converters, the binary-weighted type must have a wide range of resistors. This can lead to temperature stability and difficulties in switching.

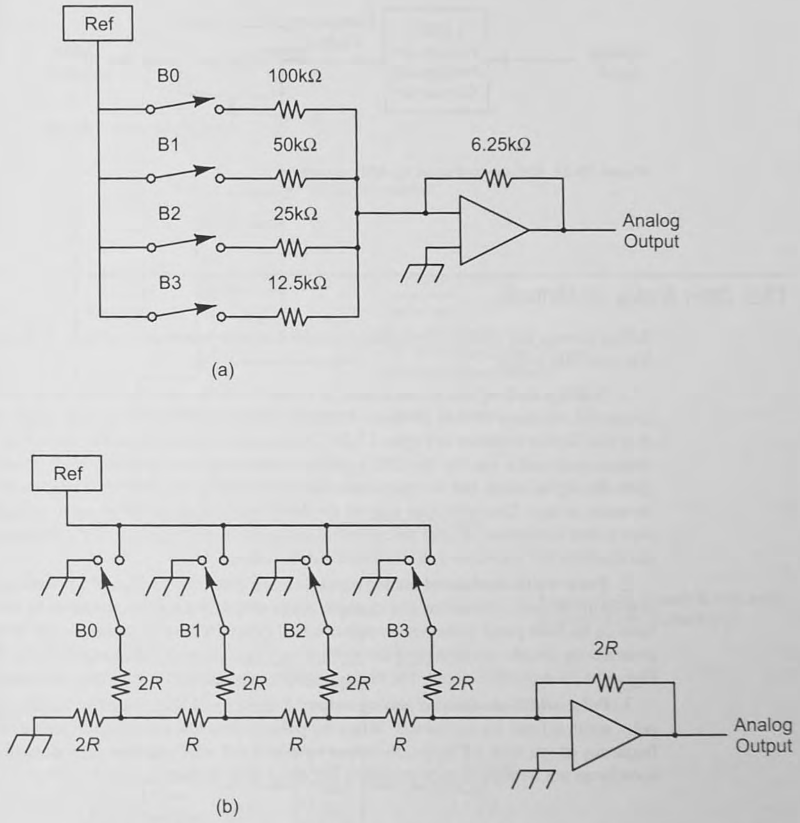


Figure 13-23 D/A converters. (a) Binary weighted. (b) R-2R ladder network.

2. **R-2R ladder D/A:** Figure 13-23b shows another popular design. Here, a wide range of resistor values is not required; it is relatively easy to create highly accurate *R*/2*R* resistance ratios. However, single-pole, double-throw switches are needed. As the switches are changed from the grounded to the reference position, a binary-weighted current is supplied to the summing junction.

3. **Multiplying D/A:** The R-2R ladder D/A can serve as a *multiplying D/A* if the reference voltage is used as an input. The reference voltage can vary over the maximum voltage range of the amplifier and is multiplied by the digital code.

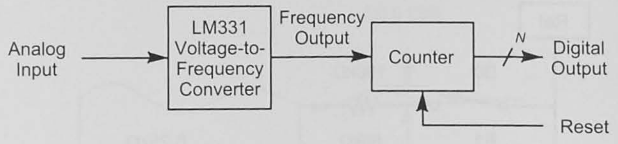


Figure 13-24 Voltage-to-frequency A/D conversion.

13.9 Other Analog I/O Methods

Before closing this analog I/O chapter, let us look at three nontraditional ways to achieve analog input and output.

1. **Voltage-to-frequency converters:** A *voltage-to-frequency* (*V-F*) converter or *voltage-controlled oscillator* (*VCO*), produces an output frequency proportional to the input voltage. A typical device is shown in Figure 13-24. The counter is set to zero at the start of the conversion cycle and is read by the CPU a predetermined time later. The number in the counter gives the digital value, but the microcontroller must wait for the prescribed amount of time, no more, no less. This technique is good for slowly varying signals or when an average value over a time is required. If your microcontroller can measure the period of the frequency output from the V-F converter you can eliminate the counter.

2. **Pulse-width-modulated analog input:** In some cases, the position of a potentiometer may be the desired information. For example, a user may vary a control parameter by turning a knob on the front panel. If the potentiometer is not needed for another purpose—say, to control some analog circuit—it can control the width of an output pulse of a monostable multivibrator. Figure 13-25 shows this circuit. The microcontroller must measure the width of the output pulse.

3. **Pulse-width-modulated analog output:** Figure 13-26 shows a pulse waveform. The pulse width is t and the period is T. When the pulse train is low-pass-filtered with a cutoff frequency of less than $1/T$ hertz, the output voltage is $A * t/T$. Pulse-width-modulated (PWM) waveforms are frequently used to control the speed of dc motors.

13.10 Conclusion and Chapter Summary Points

- A data acquisition system consists of transducers, signal conditioning, an analog multiplexer, a sample-and-hold, an analog-to-digital converter, and a parallel input interface to the CPU.
- Transducers convert physical processes to electrical signals.
- Signal conditioning provides isolation and buffering, low-pass filtering, and amplification.
- Shannon's theorem specifies the maximum frequency that can be sampled for a given sampling frequency.

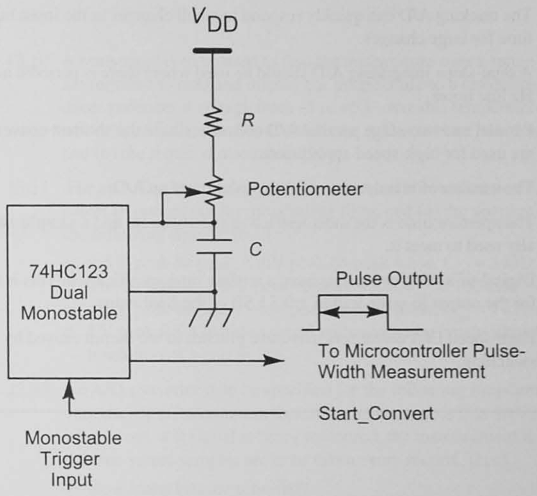


Figure 13-25 Analog input with pulse-width modulation.

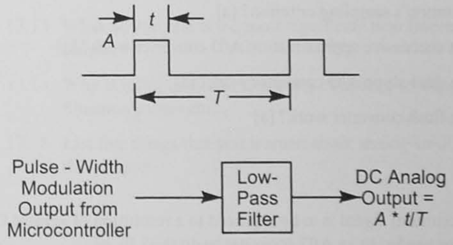

t
A
T
Pulse - Width Modulation Output from Microcontroller
Low-Pass Filter
DC Analog Output = A * t/T

Figure 13-26 Analog output with pulse-width modulation.

- The maximum frequency that can be sampled, called the Nyquist frequency, is equal to one-half the sampling frequency.
- An undersampled waveform can cause aliasing.
- Aliasing causes errors in the digitized values.
- Other error sources include electronic noise, aperture error, and quantization error.
- In a well-designed system, all noise sources should be less than the quantization error.
- The successive approximation A/D is probably the most common.

- The tracking A/D can quickly respond to small changes in the input but requires more time for large changes.
- A dual-slope integrating A/D should be used where there is periodic noise, such as 60 Hz line noise.
- Parallel and two-stage parallel A/D converters have the shortest conversion time and are used for high-speed applications.
- The number of bits determines the resolution of an A/D.
- The aperture time is the most restrictive specification, and a sample-and-hold is generally used to meet it.
- Digital-to-analog converters have a settling time specification. This is the time taken for the output to settle within ±0.5 LSB of the final value.
- High-speed D/A converters may have glitches in the output caused by asymmetrical switching.

13.11 Problems

Explore

13.1 Briefly explain the following terms: aperture time, conversion time, aliasing, Nyquist frequency. [a]

13.2 What is Shannon's sampling criterion? [a]

13.3 How does a successive approximation A/D converter work? [a]

13.4 How does a dual-slope A/D converter work? [a]

13.5 How does a flash converter work? [a]

Stimulate

13.6 A 10 V (maximum) signal is to be digitized to a resolution of at most 0.01 V. How many bits are needed in an A/D converter to do this? [b, c]

13.7 The A/D converter conversion time is 100 μs. What is the maximum frequency that can be digitized without the occurrence of aliasing? [b, c]

13.8 An A/D converter is required to digitize a 1 kHz sinusoidal waveform. What is the maximum allowable conversion time for the A/D? Assume that a sample-and-hold circuit is being used to give the correct aperture time. [b, c]

13.9 An A/D converter is to digitize a 10 V full-scale signal to a resolution of 1 part in 1024. [b, c]

a. How many bits are required?
b. When a 9 V signal is being digitized, what is the accuracy of the measurement?
c. What is the accuracy when a 1 V signal is digitized?

Challenge

13.10 A transducer is to be used to find the temperature over a range of –100 to 100°C. We are required to read and display the temperature to a resolution of ±1°C. The transducer produces a voltage from –5 to +5 V over this temperature range with 5 mV of noise. Specify the number of bits in the A/D converter based on (a) the dynamic range and (b) the required resolution. [b, c]

13.11 For an A/D converter, specify (1) maximum conversion time, (2) number of bits, (3) cutoff frequency for the antialiasing filter, and (4) the aperture time to digitize each of the following signals: [b, c]

a. ±5 V peak-to-peak, 5 mV peak-to-peak noise, f_{max} = 3 kHz
b. 0 to 10 V peak, 5 mV peak-to-peak noise, f_{max} = 100 kHz
c. ±1 V peak-to-peak, 5 mV peak-to-peak noise, f_{max} = 1 kHz
d. 1 V peak RS-170 video signal with maximum bandwidth of 5 MHz with a required resolution of 1 part in 256

13.12 An A/D converter is to be specified for the following measurement: the signal will not vary during the conversion time; the signal range is 0 to 10 V; there is 1 mV of noise; when a one volt signal is being measured, the measurement is to be within ±0.5% of the true value; samples are to be taken every second. [b, c]

a. How many bits are required?
b. How would you specify the conversion time and aperture time?

Reflect on Learning

13.13 What do you feel is the most significant new information you learned from this chapter?

13.14 Why is it important for designers of analog-to-digital converter systems to understand Shannon's theorem?

13.15 List five things that you learned about analog-to-digital conversion while studying this chapter.

14 Counters and Timers

Objectives

Many embedded applications require a timer to generate waveforms of a specific frequency, to time external events, to count events, and to generate interrupts at specific intervals. In this chapter we will look at the basic operation of the timer circuits found in modern microcontrollers.

14.1 Introduction

Designers of embedded systems often refer to "real-time" events, or real-time control. Real time does not mean time in hours, minutes, and seconds. Instead, the term usually refers to generating time intervals to create waveforms of a specific frequency for driving stepper motors, for example, or for generating interrupts to acquire data from an external source such as an analog-to-digital converter. The timer module in your microcontroller can do the following functions:

- Generate accurate timing signals and waveforms.
- Measure time intervals.
- Generate interrupts at specific intervals.
- Capture and count external events.

14.2 The Timer/Counter

The Timer Overflow

The heart of the timer module is actually a counter, as shown in Figure 14-1. A programmable (in some microcontrollers but not in others) divider or prescaler reduces the bus clock

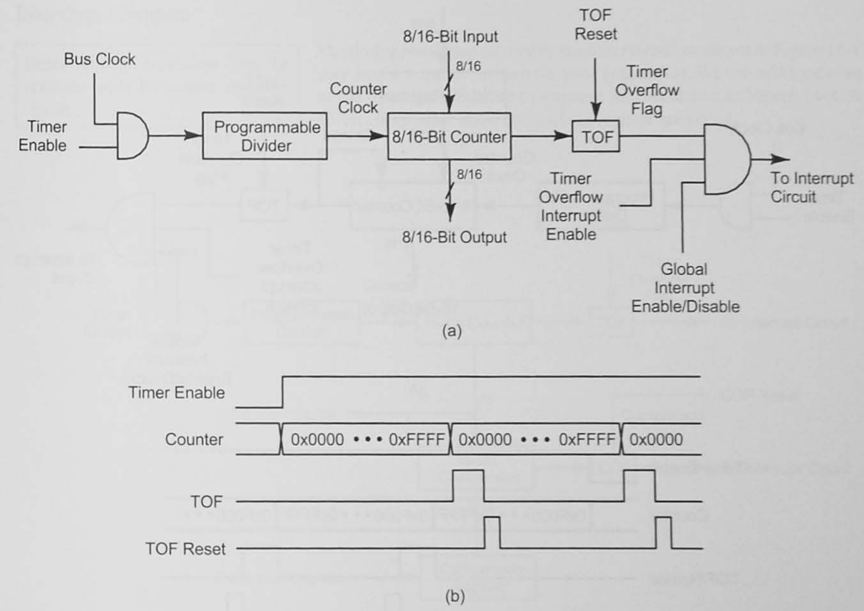


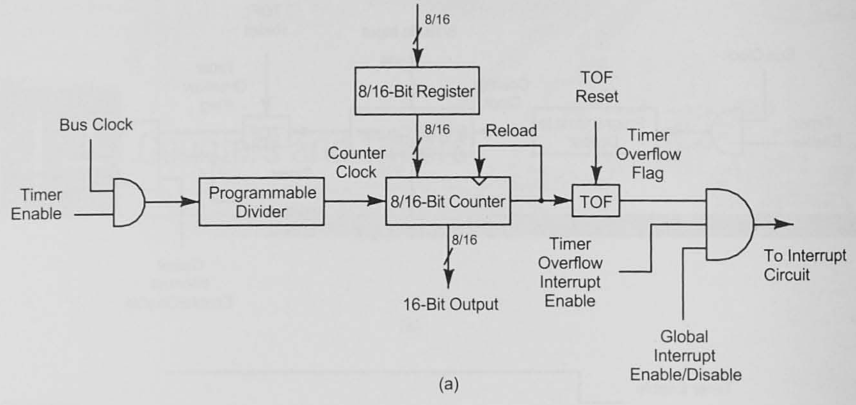
Figure 14-1 A16-bit timer/counter. (a) Timer overflow. (b) TOF timing.

The heart of a timer system is a counter.

frequency to increment an 8- or 16-bit counter. The counter is free running and counts from 0x0000 to 0xFFFF. When it reaches the maximum it rolls over (overflows) to 0x0000 and sets a hardware bit called the *timer overflow flag* (*TOF*). Your software must enable the timer by setting the timer enable bit, and it can either poll the timer overflow flag or use the flag to generate an interrupt. You can also read the 8/16-bit counter output. The timer overflow gives us our first level of (fairly crude) timing intervals. See Example 14-1. After the timer overflow flag has been set by the hardware, you must reset it in your software. Each microcontroller will have a different way to do this. Figure 14-1b shows the timing sequence for timer overflows.

In another version of the basic timer function, you are allowed to initialize the 16-bit counter with a value. Then, when the timer enable bit is asserted, you will have a fixed time given by the count and the counter clock frequency to wait until the TOF is set.

Figure 14-2 shows yet another version of the basic counter; here the timer overflow flag reloads the counter from a register. This feature is very useful when you are generating square waves. Another variation in the basic timer that some microcontrollers offer is the ability to allow the counter to count up or count down.



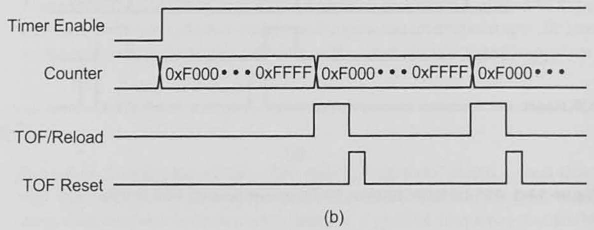


Figure 14-2 Automatic reload counter. (a) Hardware design. (b) TOF reload timing.

Example 14-1 Timer Overflow Intervals

Calculate the time intervals between timer overflows for a 16-bit counter like the one shown in Figure 14-1, assuming the following counter clock frequency input to the counter:

(a) 8 MHz
(b) 4 MHz
(c) 100 kHz

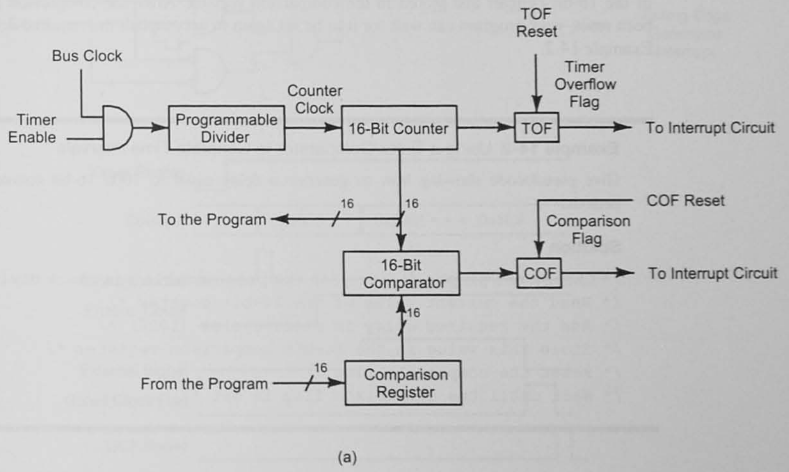
Solution

(a) $2^{16}/8 \times 10^6 = 8.192$ ms
(b) 16.384 ms
(c) 0.65536 s

Timer Output Compare

Better timing resolution can be achieved with an *output compare* circuit.

The timing resolution offered by the timer overflow shown in Figure 14-1 may not be accurate enough for your application. We can add hardware to the basic timer/counter to improve this, as shown in Figure 14-3. A 16-bit comparator and a comparison register are added.



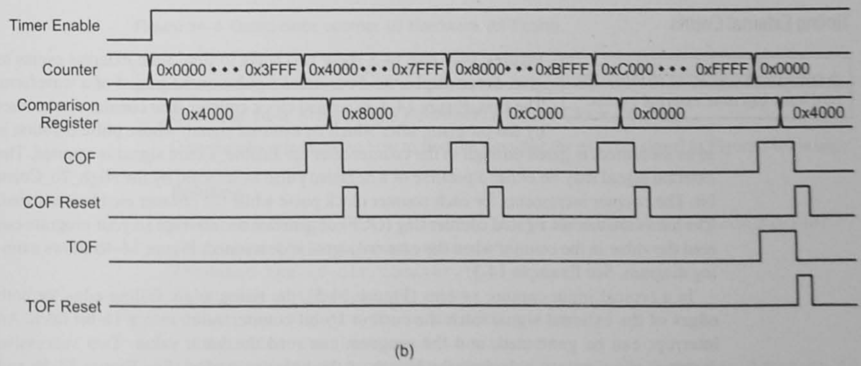


Figure 14-3 Timer output compare. (a) Hardware. (b) Output compare timing.

The timing diagram in Figure 14-3b shows how the output compare circuit works. The comparison register is initialized to 0x4000, and when the counter reaches this value, the comparison flag, COF, is set. The program detects this, either by polling the bit or with an interrupt, and then changes the comparison register to the new value, 0x4000 clock cycles away (0x8000) and resets the COF bit. Each time the comparison is made, the next comparison value is written and the COF reset.

This hardware can generate time intervals to the accuracy of the counter's clock. Any time delay (up to the maximum of 2^{16} counter clock cycles) can be added to the current value of the 16-bit counter and stored in the comparison register. After the comparison flag has been reset, your program can wait for it to be set again to accomplish the required delay. See Example 14-2.

Example 14-2 Using a Timer/Comparator to Generate Time Intervals

Give pseudocode showing how to generate a delay equal to 1000 16-bit counter clock periods.

Solution

```
/* Choose the proper divider for the programmable bus clock divider */
/* Read the current value of the 16-bit counter */
/* Add the required delay in clock cycles (1000) */
/* Store this value in the 16-bit comparison register */
/* Reset the comparison flag */
/* Wait until the comparison flag is set */
```

Timing External Events

A *gated clock counter* or an *input capture latch* can time external events.

Figures 14-4 and 14-5 show two ways to time such external events to give, for example, the duration of a pulse or the period of a waveform. The first, Figure 14-4, is a gated clock counter. The counter can be reset by the program, after which an external signal, whose pulse duration is to be measured, is gated through to the counter after the Enable_Count signal is asserted. The external signal may be either a positive or a negative pulse as selected by the High_To_Count bit. The counter increments for each counter clock pulse while the counter enable is asserted. The hardware can set a gated counter flag (GCF) or generate an interrupt so your program can read the value in the counter when the external signal is deasserted. Figure 14-4b shows a timing diagram. See Example 14-3.

In a typical input capture system (Figure 14-5), the rising edge, falling edge, or both edges of the external signal latch the current 16-bit counter value into a 16-bit latch. An interrupt can be generated, and the program can read the latch value. Two successive captures allow you to calculate the length of the pulse or period. See Figure 14-5b and Example 14-4.

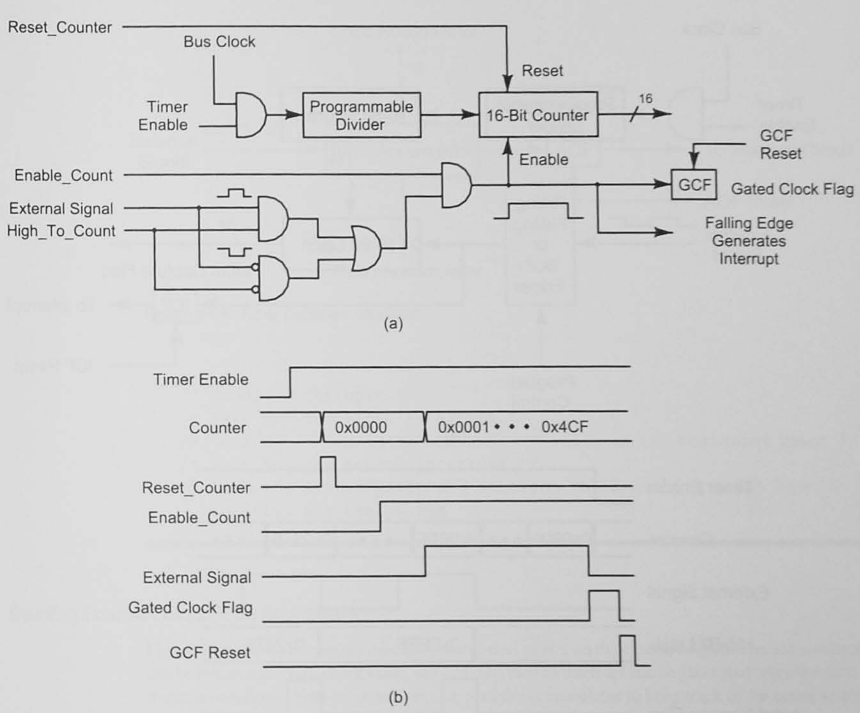


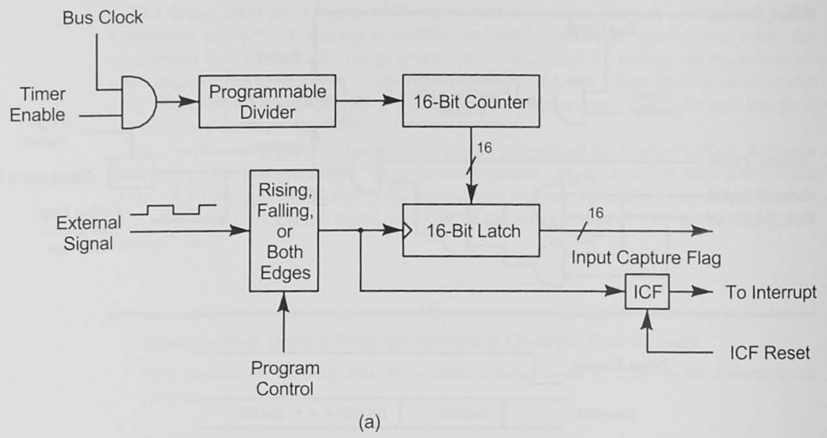
Figure 14-4 Gated clock counter. (a) Hardware. (b) Timing.

Example 14-3 Measuring a Pulse with a Gated Clock Counter

Give pseudocode showing how to find the time that the external signal in Figure 14-4 is high.

Solution

```
/* Choose the proper divider for the programmable bus clock divider */
/* Enable the interrupt system */
/* Reset the 16-bit counter */
/* Enable interrupts */
/* Set the High_To_Count control bit */
/* Wait for the interrupt to occur */
/* Read the 16-bit counter value and calculate the pulse duration */
```



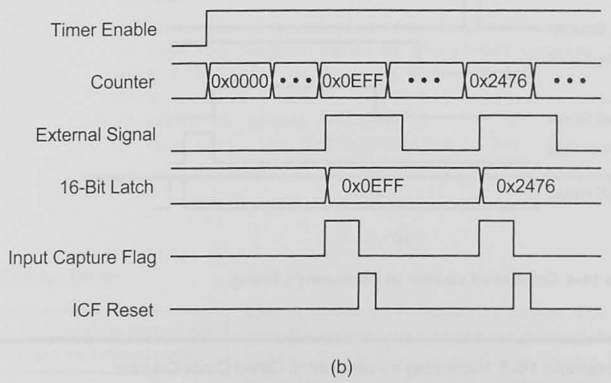


Figure 14-5 Timing input capture. (a) Hardware. (b) Timing.

Example 14-4 Measuring the Period of a Waveform

Give pseudocode showing how to find the period of the external signal in Figure 14-5.

Solution

```
/* Choose the proper divider for the programmable bus clock divider */
/* Enable rising edges to latch the 16-bit counter */
```

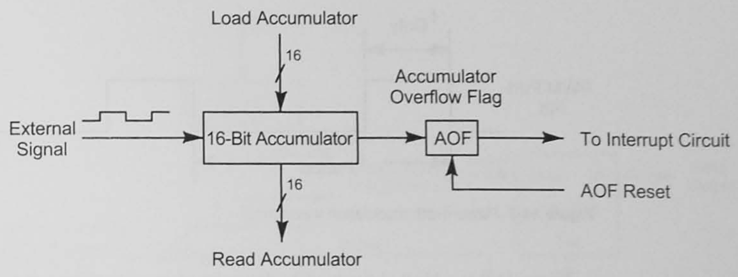


Figure 14-6 Timer pulse accumulator.

```
/* Enable interrupts */
/* Wait for the first interrupt */
/* Read the 16-bit latch and save the value as the beginning count */
/* Wait for the second interrupt */
/* Read the 16-bit latch and subtract the beginning count from it
   to calculate the period */
```

Counting External Events: Pulse Accumulator

Figure 14-6 shows a timer accumulator that counts pulses on the external signal. The accumulator can be initialized with a count value, say –24, and then an interrupt can be generated when the accumulator overflows. Your program can also read the accumulator to keep track of the count value.

14.3 Pulse-Width Modulation (PWM) Waveforms

If PWM hardware is included in the microcontroller, you can generate PWM waveforms with no software overhead.

Pulse-width modulation waveforms are used in many embedded applications. For example, a PWM waveform can control the speed of a dc motor. Figure 14-7 shows a pulse-width-modulated waveform. Two time intervals must be specified and controlled. These are the period (t_{period}) and the time the output is high (t_{duty}). A term that describes a pulse-width-modulated waveform is *duty cycle*. Duty cycle is the ratio of t_{duty} to t_{period} and is usually given as a percentage:

$$\text{Duty cycle} = \frac{t_{duty}}{t_{period}} \times 100\%$$

You can generate a PWM waveform by using the timer/comparator hardware shown in Figure 14-3. Your software must keep track of the delay — the number of clock cycles it must wait until the next change in the output.

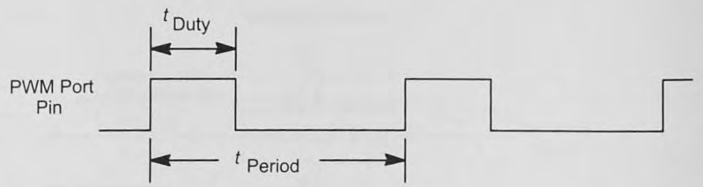


Figure 14-7 Pulse-width modulation waveform.

Figure 14-8 is a block diagram of hardware found in some microcontrollers to generate PWM waveforms without software intervention once the system has been initialized. The bus clock divided by the programmable divider drives the 16-bit counter. The counter starts at 0x0000, and the output of the flip-flop is high. As the counter counts up, it reaches the value in the *duty cycle register*, and the *duty cycle comparator* resets the flip-flop. The counter continues to count until it reaches the value in the *period register* when the *period comparator* resets the counter to 0x0000 and sets the output flip-flop to start the PWM again.

14.4 "Real" Real-Time Clock: Clock Time

As mentioned in Section 14.1, the timer/counter in your microcontroller does not give you "clock" time, that is, hours, minutes, and seconds, at least not unless you have written an application using the timer.

To generate "real" time (hours, minutes, seconds), you can add a *real-time clock* peripheral.

Figure 14-9 shows a typical external real-time clock (RTC). The Dallas Semiconductor/Maxim DS3234 has an accurate on-chip oscillator for keeping the time, and it counts seconds, minutes, hours, day, date, month, and year with leap year compensation. The data output is in binary coded decimal (BCD), which makes it easy to display the time information without having to convert from binary to BCD. It can generate two alarms at times you set, and it has 256 bytes of user RAM. As you can see, a battery can be used to keep the RTC powered up to maintain its time when V_{DD} is lost. This particular chip has a voltage monitoring circuit and can assert the RESET_L signal when V_{DD} drops below a power-failure voltage V_{PF}. The microcontroller interface for the DS3234 uses the serial peripheral interface (SPI, described in Chapter 12). Other chips from Dallas Semiconductor/Maxim and other manufacturers offer I^{2}C and parallel bus interfaces.

14.5 Conclusion and Chapter Summary Points

In this chapter we have presented a variety of features found in microcontroller timers. Although your microcontroller's timer may have a variety of additional features and controls, the fundamentals follow what we have presented here.

- The heart of the timer is a counter clock, driven by a derivative of an internal clock.
- The internal clock frequency may be divided by a programmable divider or prescaler.

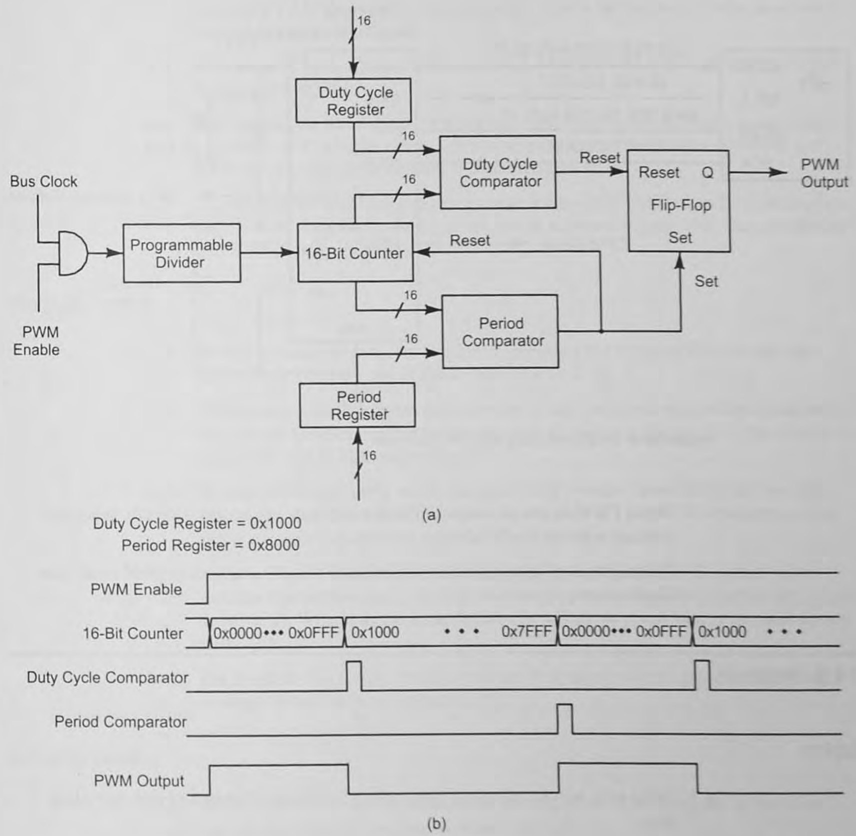


Figure 14-8 Block diagram of a pulse-width modulator. (a) Hardware. (b) Timing.

- The counter overflows when it reaches the maximum count (0xFF or 0xFFFF) and sets a timer overflow flag.
- The timer overflow flag can be used to generate timing intervals.
- More precise timing intervals can be generated with a timer output compare circuit.
- External events can be measured with a gated clock counter or an input capture circuit.
- External events can be counted with a timer pulse accumulator.

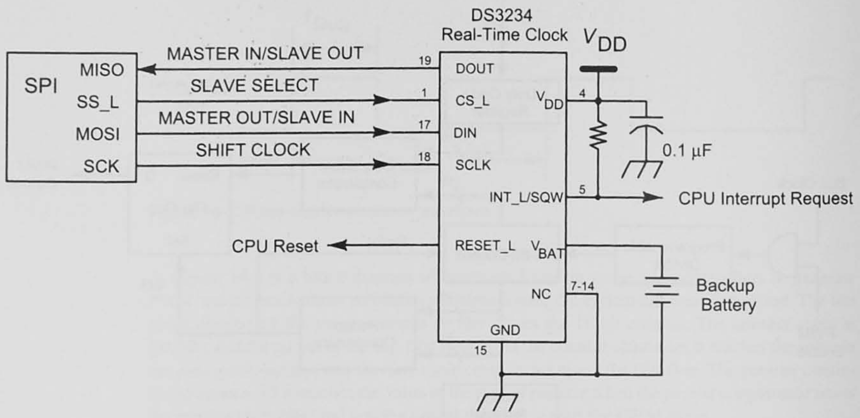


Figure 14-9 Real-time clock with SPI interface.

- When PWM hardware is added, PWM waveforms can be automatically generated without software overhead.
- To keep track of hours, minutes, and seconds ("real" time), you can add a real-time clock circuit.

14.6 Problems

Explore

14.1 Using your web-based search tool, make a list of manufacturers of real-time clock chips.

14.2 Assume the bus clock in Figure 14-1 is 8 MHz and that your choices for the programmable divider are to divide by 1, 2, 4, 8, 16, or 32. [b, c]

a. For each of these divider values, find the period of the counter clock.

b. For each of these divider values, find the period of the timer overflow, assuming an 8-bit and then a 16-bit counter.

Stimulate

14.3 Describe how to use the timer/counter circuit in Figure 14-2 to generate a 10 kHz square wave. Assume a counter clock frequency of 8 MHz. [c]

14.4 Assuming a 2 MHz counter clock frequency, what is the interval between timer overflows in Figure 14-2b? [b]

14.5 Assuming a 2 MHz counter clock frequency, what is the period of the external signal in Figure 14-5b? [b]

14.6 Explain why the timer/counter comparison circuit in Figure 14-3 can wait for a comparison up to 2^{16} counter clock cycles away even though the counter overflows and resets to zero when it reaches the maximum count. [a, b]

14.7 A timing circuit is needed that can generate a time delay longer than 2^{16} counter clock cycles of the timer/counter comparison circuit shown in Figure 14-3. Use pseudocode to describe a strategy to do this.

Challenge

14.8 Write a pseudocode program that could generate a PWM waveform that uses the timer output compare circuit shown in Figure 14-3. [c]

14.9 Write a pseudocode program to implement a real-time clock with binary coded decimal output, assuming a timer output compare as shown in Figure 14-3. The clock is to display hh:mm in 24-hour format. [c]

14.10 What might occur during use of the gated clock counter shown in Figure 14-4 that would give you an erroneous value for the duration of a pulse? What strategy could you use to guard against this problem? [b, c]

14.11 Assume you are to use the timer in Figure 14-2 to generate a 50 Hz square wave. Assume that the bus clock is 8 MHz and the programmable divider factors are 1, 2, 4, 8, 16, or 32. Write a pseudocode design that will allow you to output required square wave. [c]

14.12 For Problem 14.11, what limits the highest frequency you can generate with your strategy? What limits the lowest? [a, b]

Reflect on Learning

14.13 Make a list of as many applications as you can think of in which one of the timer circuits described in this chapter would be useful.

14.14 List five things you learned about timers in this chapter.

15 Single-Chip Microcontroller Interfacing Techniques

Objectives

The single-chip microcontroller you use in an embedded system must be connected to the outside world to be useful. In earlier chapters we covered a variety of the I/O capabilities such as the analog-to-digital converter and serial and parallel I/O. This chapter will give examples showing how to connect external devices such as keypads, LCD displays, LEDs, and dc and stepper motors to your microcontroller.

15.1 Microcontroller Chip I/O

Modern microcontrollers package a variety of I/O devices, such as analog-to-digital converters, timers, and parallel and serial input and output interfaces into a single IC chip. Typical examples of modern microcontrollers are the Atmel ATtiny256 (Figure 15-1), the Texas Instruments MSP430 (Figure 15-2), and the Freescale Semiconductor Flexis microcontroller (Figure 15-3). They all contain I/O devices that must connect to switches, LEDs, and so on in the outside world. Typically, the microcontroller's internal need to connect to external devices is greater than the number of pins or port bits available. For example, in the case of the Freescale Flexis, there are 178 I/O functions that need a pin, far too many for reasonable size IC packages (the largest Flexis package is 80 pins). The Freescale designers have chosen to multiplex I/O functions onto the port I/O pins (Table 15-1). Smaller microcontrollers such as the Atmel (24 pins) and the Texas Instruments (64 pins) devices face similar problems.

Because parallel and serial I/O are built into the microcontroller itself, the input and output interfaces covered in Chapter 9 do not have to be designed. Connecting the microcontroller to external devices is much easier, as we will see in this chapter.

Microcontroller Initialization

Chapter 7 showed that a C program's start-up code usually initializes the microcontroller's hardware before it begins to execute your program. Typically the stack pointer register is initialized to point to RAM, and a watchdog timer that will allow your program to recover if it gets lost may be started. In C programs the start-up code also may initialize RAM data areas with zero.

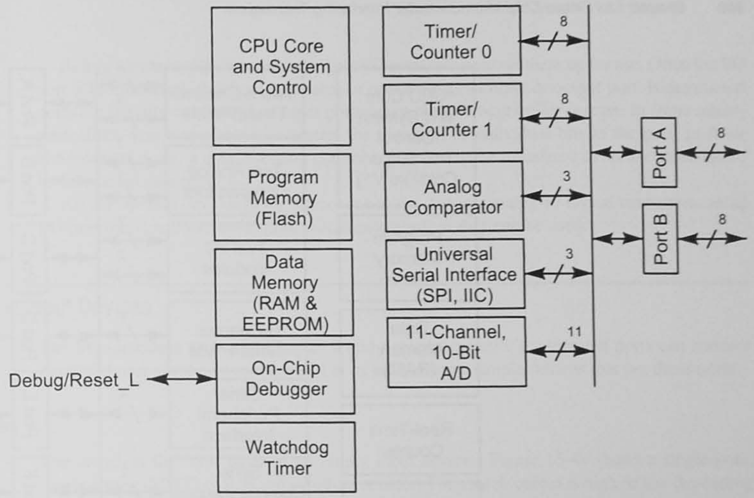


Figure 15-1 Atmel ATtiny261 microcontroller.

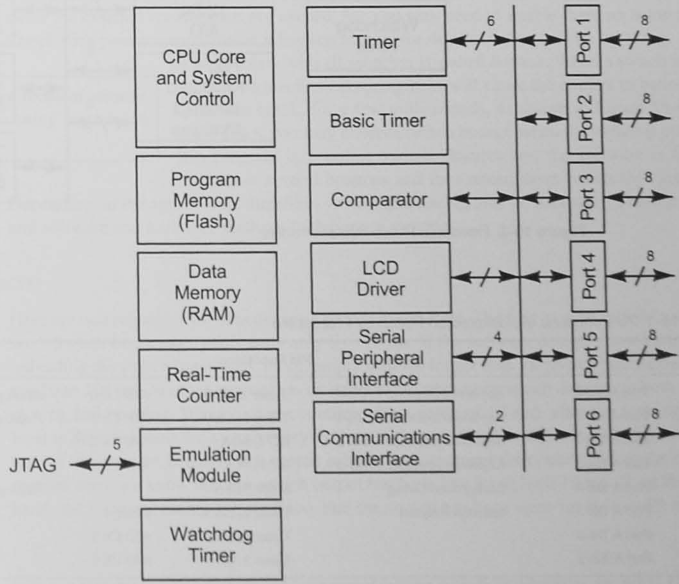


Figure 15-2 Texas Instruments MSP430 microcontroller.

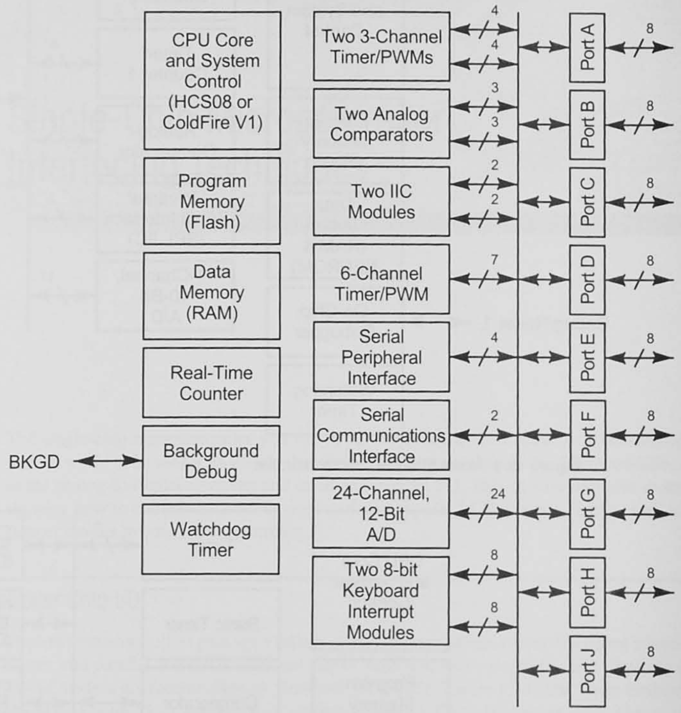


Figure 15-3 Freescale Flexis microcontroller.

Table 15-1 I/O Functions Multiplexed on Port A by Freescale

Pin	Pin Functions				
0	Port A Bit-0	Keyboard Interrupt 1, Bit-0	Timer 1, Ch 0	A/D Ch 0	Analog Comparator 1+
1	Port A Bit-1	Keyboard Interrupt 1, Bit-1	Timer 2, Ch 0	A/D Ch 1	Analog Comparator 1–
2	Port A Bit-2	Keyboard Interrupt 1, Bit-2	IIC 1, SDA	A/D Ch 2	
3	Port A Bit-3	Keyboard Interrupt 1, Bit-3	IIC 1, SCL	A/D Ch 3	
4	Port A Bit-4	Background Debug	Mode Select		Analog Comparator 1 Out
5	Port A Bit-5	Interrupt Request	Timer 1, clock	Reset_L	
6	Port A Bit-6		Timer 1, Ch 2	A/D Ch 8	
7	Port A Bit-7		Timer 2, Ch 2	A/D Ch 9	

Before we can use the microcontroller's I/O ports, we must set them up for use. Often the I/O port is bidirectional; that is, it is capable of operating as an input or output port. Bidirectional ports are initially configured as input ports when the microcontroller is reset. In some microcontrollers, you may choose to control the direction of individual bits in the port. In these bidirectional ports, a data direction control register must be initialized to set the direction of its associated data port.

All the internal devices in the microcontroller—timers, analog-to-digital converters, serial I/O interfaces, and so on—require initialization before they can be used.

15.2 Simple Input Devices

The I/O interfaces shown in Chapter 9 and a microcontroller's integrated ports can connect parallel devices to the system buses. Let us look at some simple devices that use these ports.

Input Switches

The switch is the most basic of all binary input devices. Figure 15-4a shows a single-pole, single-throw (SPST) switch and a pull-up resistor. The switch output is high or low depending on the switch position. Figure 15-4b shows a multiple pole, rotary switch. Pull-up resistors are necessary for each of these switches to provide a high logic level when the switch is open. The input ports in most microcontrollers often have an internal pull-up resistor on all inputs; in such cases the external resistors are not needed, but you may need to enable them on some ports. Check with your microcontroller reference manual for details of the port you are using.

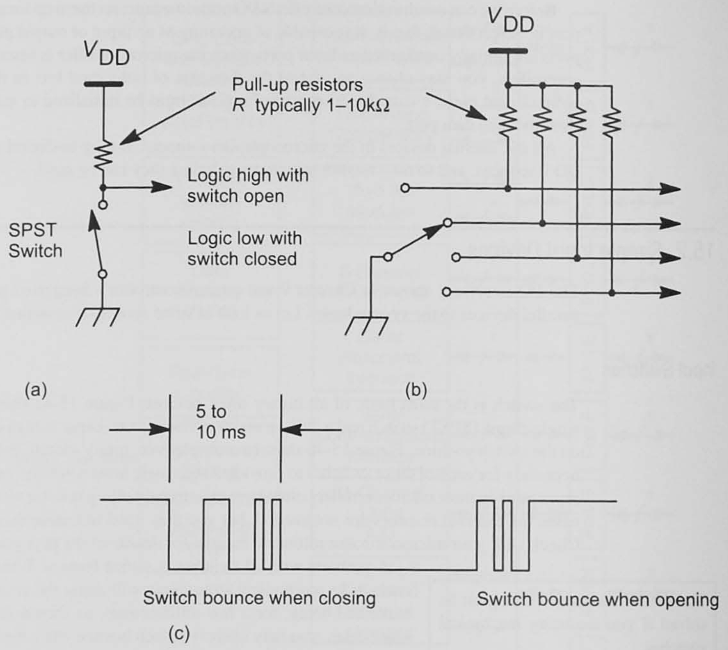
The *switch bounce* problem must be solved if you are using mechanical switches.

A problem with all switches is *switch bounce*. When a switch makes contact, its mechanical springiness will cause the contact to bounce, or make and break, for a few milliseconds, as shown in Figure 15-4c. In some cases, you may observe switch bounce when the switch is opened. If a program is counting switch closures and the software is fast, it may count several bounces and thus return more counts than are real. Depending on the application, therefore, switch debouncing may be necessary. There are several software and hardware methods to debounce switches.[1]

Software Debouncing

Here are two strategies for debouncing a switch in software. The first may be called *"wait and see."* Switch bouncing usually lasts only 5 to 10 ms. If the software detects a low logic level, indicating the switch has closed, it can simply wait for longer than the switch bounce duration, say 20 to 100 ms. Another approach is an integrating debouncer, which debounces both switch closing and opening. We initialize a counter with a value of 10 and, after the first logic low level is detected, poll the switch every millisecond or so. If the switch output is low, we decrement the counter. If the switch output is high, we increment the counter. When the counter reaches zero, we know that the switch output has been low for at least 10 ms. If, on the other hand, the counter reaches 20, we know that the switch has been open for at least 10 ms. The

[1] For a good review of switch bouncing and hardware and software methods for debouncing, see *Jack G. Ganssle, A Guide to Debouncing* (2004), http://www.ganssle.com/debouncing.pdf.



VDD
Pull-up resistors
R typically 1–10kΩ
Logic high with switch open
Logic low with switch closed
SPST Switch
(a)
(b)
5 to 10 ms
Switch bounce when closing
Switch bounce when opening
(c)

Figure 15-4 Switches used for binary input: (a) single-pole, single-throw (SPST) switch; (b) multiple pole switch; (c) switch bounce.

initial value of the counter is set longer than the expected bounce time. Example 15-1 shows the pseudocode for this algorithm, and a C program is given in Example 15-2.

Example 15-1 Pseudocode Design for Integrating Switch Debouncer

```
INITIALIZE Count = 10
WHILE ((Count > 0) and (Count < 20))
     DO
        Delay 1 millisecond
        Get Switch Input
        IF Switch Closed
           THEN Decrement Count
           ELSE Increment Count
        ENDIF Switch Closed
     ENDO
```

```
ENDWHILE ((Count > 0) and (Count < 20))
IF Count = 0
     THEN Switch is closed
     ELSE Switch is open
ENDIF Count = 0
```

Example 15-2 C Program Debouncer

Write a C program that implements the debouncer algorithm shown in Example 15-1.

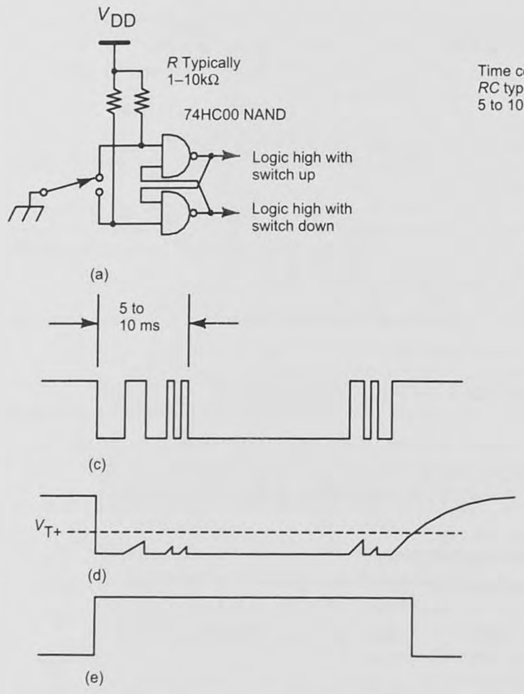
Solution

```
/*************************************************************
 * Debounce routine using integrating debouncer.
 * Calls delay_X_ms which generates an X millisecond delay.
 * Checks the switch defined below
 * Calling:
 * unsigned char debounce( unsigned int length );
 *   where length is the time in milliseconds for switch bounce
 *   to last.
 *   The return is the final value of the switch (0 or 1).
 *************************************************************/
 /*********************************************************
  * Define the switch port location
  * Put your own definition here
  *********************************************************/
/* Pointer to switch */
#define p_SwPort (volatile unsigned char *) 0x0258
#define SwBit 0x20   /* Bit 5 */
/*********************************************************/
void delay_X_ms( unsigned int X );     /* Variable delay */
/*********************************************************/
unsigned char debounce(unsigned int bounce_length) {
/*********************************************************/
unsigned int count;
unsigned char switch_val;
  /* initialize count */
  count = bounce_length/2;
  while (( count > 0 ) && ( count < bounce_length )){
    /* Delay a millisecond */
    delay_X_ms(1);
    /* If switch == 0, decrement count else increment it */
    if ((*p_SwPort & SwBit) == 0)
      --count;
    else ++count;
  }
```

```
    if (count == 0) switch_val = 0;
    else switch_val = 1;
    return( switch_val );
}
```

Hardware Debouncing

Figure 15-5 shows two hardware debouncing schemes. In Figure 15-4a, a NAND latch debounces a single-pole, double-throw (SPDT) switch. NOR gates can be used also for the



V_{DD}
R Typically 1–10kΩ
74HC00 NAND
Logic high with switch up
Logic high with switch down
(a)
V_{DD}
Time constant *RC* typically 5 to 10 ms
74HC14 Schmitt Trigger
R
Logic low with switch open
Logic high with switch closed
C
(b)
5 to 10 ms
(c)
V_{T+}
(d)
(e)

Figure 15-5 Hardware debouncing methods: (a) NAND latch debouncer; (b) Schmitt trigger; (c) switch bounce waveform; (d) Schmitt input voltage (V_c); (e) Schmitt output.

latch. A disadvantage of this debouncer is the requirement for a single-pole, double-throw switch, a type of switch that is more expensive and bulky than the single-pole, single-throw (SPST) shown in Figure 15-4a. Figure 15-5b shows an integrating debouncer with a Schmitt trigger gate. Before the switch closes, the capacitor is charged and the output of the gate is low. When the switch closes, the capacitor discharges and the gate output switches high. Because of the gate's built-in hysteresis, it will not switch low again until the input voltage exceeds a threshold V_{T+}. The *RC* time constant of the circuit should be designed to prevent the gate's input voltage from exceeding the threshold while the switch is bouncing. See Example 15-3.

Example 15-3 Schmitt Trigger Debouncer

For the Schmitt trigger debouncer shown in Figure 15-5, assume the following: $R = 10\ \text{k}\Omega$, $V_{T+} = 1.7$ V (positive-going threshold for the Schmitt trigger when $V_{DD} = 4.5$ V), and switch bounce (Figure 15-4) lasts no more than 10 ms.

Calculate a value for the capacitor *C* so that the Schmitt trigger output will not switch during the switch bounce period.

Solution

The Schmitt trigger input voltage is given by

$$V_i = V_{DD}[1 - e^{\frac{t}{RC}}]$$

Setting $V_i = 1.7$ V and letting $t = 10$ ms, solving for *C* gives 2 µF.

Arrays of Switches

Switches can be organized as linear or matrix arrays; a linear array is shown in Figure 15-6. A variety of switches can be found, including dual in-line package (DIP) switch arrays. The switch bounce problem may need to be solved, and the array of switches must be scanned to find out which ones are closed or open. The output of the switch array could be interfaced directly to an 8-bit input port (at point A). To save some I/O lines, a 74HC151 eight-input multiplexer can be used. Software is required to scan the array shown in Figure 15-6. As the software outputs a 3-bit sequence from 000 to 111, the multiplexer selects each of the switch inputs. The software scanner then reads one bit at an input port. See Example 15-4.

Figure 15-6 shows pull-up resistors connected to the SPST switches. If your microcontroller has internal pull-ups enabled, you may not have to use these resistors when the switches are connected. If you use an 8-to-1 multiplexer, you should use these resistors.

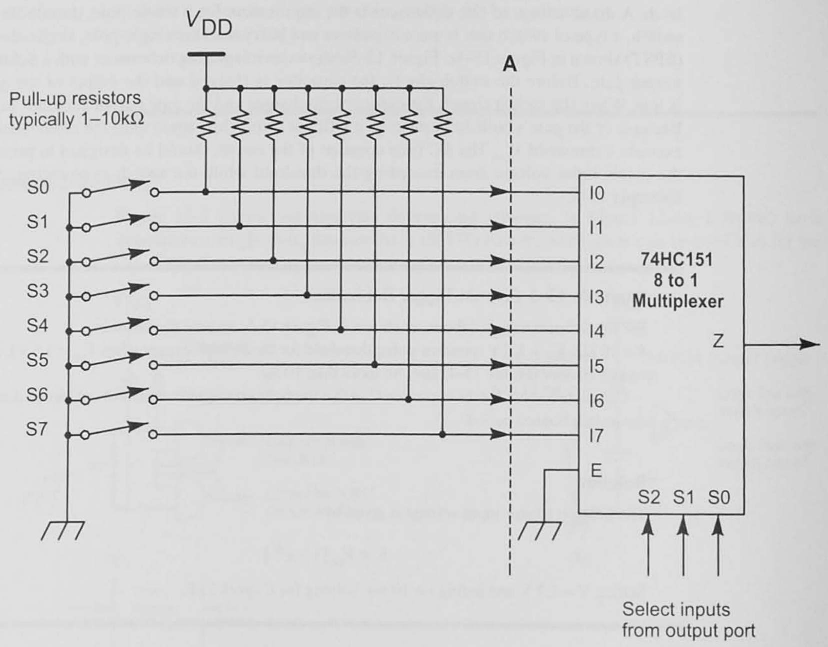

V_{DD}
Pull-up resistors typically 1–10kΩ
A
S0
S1
S2
S3
S4
S5
S6
S7
I0
I1
I2
I3
I4
I5
I6
I7
E
74HC151 8 to 1 Multiplexer
Z
S2 S1 S0
Select inputs from output port

Figure 15-6 Linear array of switches.

Example 15-4

For the linear array of switches in Figure 15-6, give a truth table showing which switch is read for each scan code output by the processor.

Solution

Scan Code	Switch	Scan Code	Switch
000	S0	100	S4
001	S1	101	S5
010	S2	110	S6
011	S3	111	S7

Example 15-5 C Code for Scanning Switches

```
/************************************************************
 * Example code segment showing how to output a 3-bit scan
 * code to select the multiplexer and to input the switch
 * position. The code scans all eight switches and returns
 * the switch positions as an unsigned char value.
 ************************************************************/
/************************************************************/
/* Define the port locations for the microcontroller in use */
/* The following are for the MC9S12C32 */
#define p_DDRT (volatile unsigned char *) 0x0242
#define p_PTT (volatile unsigned char *) 0x0240
#define p_PTA (volatile unsigned char *) 0x0000
/************************************************************/
unsigned char get_switches (void) {
  unsigned char switch_data;
  unsigned char mux_sel,port_data;
  unsigned char ddrt;
/************************************************************/
  /**********************************************************
   * Initialize your microcontroller's output bits for the
   * port being used to scan the multiplexer.
   * The code below is used on a Freescale MC9S12C32 micro
   * where Port T bits-2, 1, and 0 are the scan code bits.
   * The multiplexer output is connected to Port A bit-0
   **********************************************************/
  /* Set Port T, 2, 1 and 0 as outputs */
  /* Make 2-0 output bits */
   *p_DDRT |= 0x07;
  }
  /**********************************************************/

  switch_data = 0;     /* Initialize the return value */

  /* Now scan the multiplexer and read the output of the mux */

    for (mux_sel = 0; mux_sel < 8; ++mux_sel) {

      /* Output the mux scan code (Port T bits 2-0) */
      /* Get the current value on Port T and reset bits 2-0 */
      port_data = *p_PTT & 0xf8;
      port_data = port_data + mux_sel; /* Add the scan code */
      *p_PTT = port_data;            /* Output to the port */
      /* Now read the bit selected on Port A bit-0 and shift
       * it into switch_data.
       * Shift the data first */
       switch_data = 2*switch_data;
       /* If the bit is one, OR in 0x01 */
```

```
            if ( ( *p_PTA & 0x01 ) == 1 )
               switch_data |= 0x01;
        }
        return( switch_data );
    }
```

Explanation of Example 15-5

A three-bit scan code is output on Port T, bits 2–0. For an MC9S12C32 microcontroller, the port data direction register must be initialized so that these bits can be used as outputs. Because an application might be using the other bits in Port T for some other use, we would like the initialization code to modify only these bits. The initialization code shows us that ORing the current value of the port (`*p_DDRT`) with 0x07 accomplishes this. Similarly, when outputting the 3-bit scan code on Port T, we read the port data first, reset bits 2–0, and then add in the current scan code (`port_data = port_data + mux_sel`). Each time we read the multiplexer we shift the last reading left (`switch_data = 2*switch_data`) and then OR it with 0x01 if the output of the multiplexer is 1.

16-Key Keypad

A keypad is an array of switches arranged in a two-dimensional matrix as shown in Figure 15-7. A switch and a diode connect each intersection of the vertical and horizontal lines as shown by the blow-up view, and closing the switch connects the horizontal line to the vertical. You can connect a 4 x 4 keypad directly to four output and four input bits. For Port AD, bits 3–0 are outputs and bits 7–4 are inputs. Software can scan the keyboard by outputting the 4-bit "ring" counter code as shown in Table 15-2 and then, for each of these codes, reading the values on input bits PAD7–4. The combination of the 4-bit output and input scan codes identifies which switch is closed. A lookup table can then convert the 8-bit code to a more convenient code, such as the ASCII character code, for the hexadecimal keypad. See Example 15-6.

A problem that occurs when a keypad user hits more than one key at once, or rapidly rolls a finger from one key to another, is called *n-key rollover*. Keyboard interfaces commonly provide for two-key rollover. The keyboard hardware and software store the rapidly depressed keys in a first-in, first-out (FIFO) buffer for later readout. In an alternative strategy, *n-key lockout*, only the first or last of the sequence of keys depressed within some short period is recorded. Keyboard encoder chips incorporating all the scanning, debouncing, diodes, *n*-key rollover, and interrupt generation are available. A typical chip is the 74C922. Using these chips can eliminate the need for scanning software and hardware and provide a keyboard interface that is much easier to implement.

Example 15-7 shows a keypad input routine written in C that calls the keypad scanner of Example 15-6. Because the code in Example 15-6 does not debounce the switch, the C program enters a debouncing routine in case the key was just pressed and is still bouncing.

Table 15-2 Keyboard Scanning Codes

		Values Input on P7 P6 P5 P4				
	Output Scan Values on P3 P2 P1 P0	**1111**	**0111**	**1011**	**1101**	**1110**
		Key Pressed				
Row 3	1110 (0xE)	None	1	2	3	A
Row 2	1101 (0xD)	None	4	5	6	B
Row 1	1011 (0xB)	None	7	8	9	C
Row 0	0111 (0x7)	None	*	0	#	D
			Col 3	**Col 2**	**Col 1**	**Col 0**
			0x7	0xB	0xD	0xE

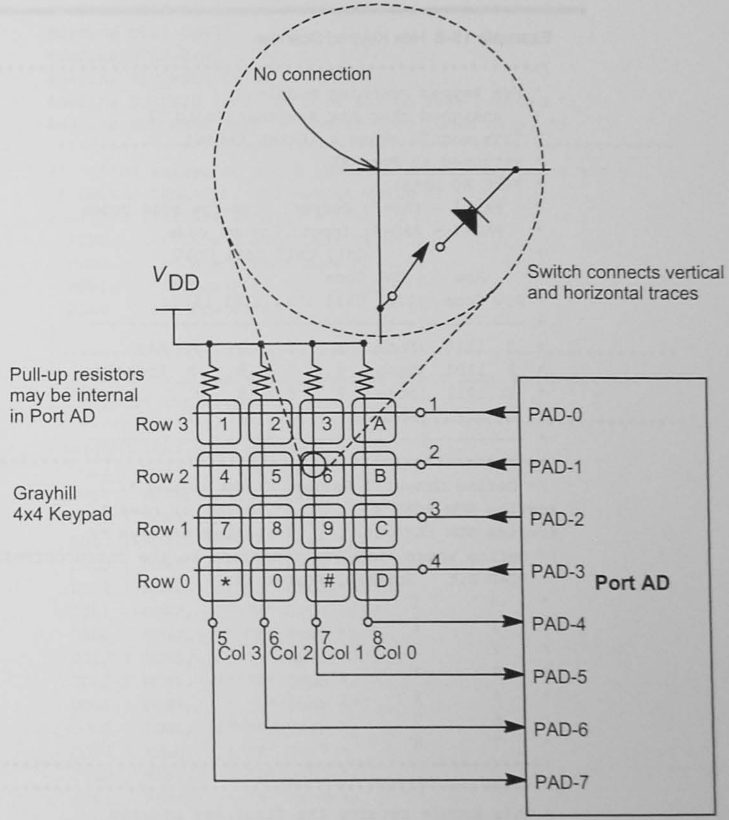


Figure 15-7 A 16-key keypad.

Example 15-6 Hex Keypad Scanner

```
/************************************************************
 * Hex keypad scanning module
 *   unsigned char hex_key_scan( void );
 * This module scans a 16-key keypad
 * attached to Port AD.
 * Port AD bits:
 *   PAD-3 - PAD-0: Output: Scan row scan codes
 *   PAD-7 - PAD-4: Input: Column code
 *           |       Col3 Col2 Col1 Col0
 *     Row   | Col Code
 * Row Code  |1111 0111 1011 1101 1110
 * ----------|-------------------------------
 *  3  1110  |None   1    2    3    A  Key
 *  2  1101  |None   4    5    6    B  Pressed
 *  1  1011  |None   7    8    9    C
 *  0  0111  |None   *    0    #    D
 * ----------|-------------------------------
 ***********************************************************/
 /* Define Grayhill Series 96 4x4 keypad */
#define NUM_ROWS 4         /* Number of rows */
#define NUM_KEYS 16        /* Number of keys */
/* Define where they are connected to the microcontroller
 * PTAD Bit   Grayhill Keypad Pin
 *    0          1
 *    1          2
 *    2          3
 *    3          4
 *    5          6
 *    6          7
 *    7          8
 ***********************************************************/
/***********************************************************
 * This module returns the first key pressed
 * when scanning. It does not check for
 * multiple col keys pressed at once.
 ***********************************************************/
/***********************************************************/
/* Define constants */
#define ROW3 0x0e     /* Row 3 scan code */
#define ROW2 0x0d     /* Row 2 scan code */
#define ROW1 0x0b     /* Row 1 */
#define ROW0 0x07     /* Row 0 */
#define OUTPUTS 0x0f  /* Row outputs */
#define INPUTS 0xf0   /* Col inputs */
#define COL3 0x70     /* Col 3 scan code */
```

```
#define COL2 0xb0     /* Col 2 */
#define COL1 0xd0     /* Col 1 */
#define COL0 0xe0     /* Col 0 */
#define KEY_MASK 0xf0
#define NO_KEYS 0xf0  /* Code for no keys pressed */
#define END_MARK 0xff /* End of Good_Codes array */
/***********************************************************/
/* Define arrays to store the scan codes, key codes and a
 * lookup table for the return value */
unsigned char Row_Codes[] = {
  ROW3,    /* Row 3 scan code */
  ROW2,    /* Row 2 scan code */
  ROW1,    /* Row 1 */
  ROW0     /* Row 0 */
};
/***********************************************************
 * This lookup table contains the 8-bit scan codes for all
 * keys on the keypad
 ***********************************************************/
unsigned char Good_Codes[ ] = {
  COL3 | ROW3,  /* "1" 0x7e */
  COL2 | ROW3,  /* "2" 0xbe */
  COL1 | ROW3,  /* "3" 0xde */
  COL0 | ROW3,  /* "A" 0xee */
  COL3 | ROW2,  /* "4" 0x7d */
  COL2 | ROW2,  /* "5" 0xbd */
  COL1 | ROW2,  /* "6" 0xdd */
  COL0 | ROW2,  /* "B" 0xed */
  COL3 | ROW1,  /* "7" 0x7b */
  COL2 | ROW1,  /* "8" 0xbb */
  COL1 | ROW1,  /* "9" 0xdb */
  COL0 | ROW1,  /* "C" 0xeb */
  COL3 | ROW0,  /* "*" 0x77 */
  COL2 | ROW0,  /* "0" 0xb7 */
  COL1 | ROW0,  /* "#" 0xd7 */
  COL0 | ROW0,  /* "D" 0xe7 */
  END_MARK      /* End marker */
};
/***********************************************************
 * This lookup table returns the ASCII code for the key.
 ***********************************************************/
/* User defined key codes. These are ASCII. */
unsigned char Key_Codes[ ] = {
  "123A456B789C*0#D"
};
/***********************************************************
 * Define the ports on the microcontroller to connect to the
 * keypad
```

the keys. In this case we have chosen to return the ASCII character code. The definition section also includes microcontroller-specific definitions for the ports used by the scanner.

The program first initializes the microcontroller's I/O ports and then scans through the rows and reads the column codes returned. If no keys are pressed, `col_code = NO_KEYS` so the return value is zero (`key_hit = 0;`). If a key has been pressed, the `Good_Codes[]` array is scanned, to look for a match for the `scan_code` that was returned when a key press was detected. The index into this array is used to return the ASCII code from the `Key_Codes[]` array. If no match was found, indicating keys in two columns were pressed, the return value is zero.

Let us say we press the 9 key. This key is in row 1 and column 1. When the row 1 code (`0x0b`) is output, reading Port AD will return `scan_code = 0xdb` where the most significant nibble (`0xd`) is the column code and the least significant nibble (`0xb`) the row code (when a port on this microcontroller is read, the output bits are read too). The `scan_code` byte is used with the two lookup tables to find the key code. First, the `Good_Codes[]` array is scanned to find a match for `scan_code = 0xdb`. The match is found for the index value of 10. This index is then used to return `0x39` (the ASCII code for 9) from the `Key_Codes` table.

Example 15-7 C Routine to Get a Keycode With Switch Debouncing

```
/*************************************************************
 * Sample program for debouncing the keys
 * on a 16 key Grayhill keypad
 * Calling:
 * unsigned char keypad_debounce(unsigned int bounce_length);
 * where
 *   bounce_length is the bounce duration in milliseconds
 *   It returns the key valid at the end of the debounce time
 ************************************************************/
unsigned char hex_key_scan( void );  /* 4x4 keypad scanner */
void delay_X_ms( int );       * Variable millisecond delay */
/*************************************************************/
/* Debouncer for the keypad */
unsigned char keypad_debounce(unsigned int bounce_length){
  int count;
  unsigned char get_key,new_key;
/*************************************************************/
  /* Get a key from the keypad */
  get_key = hex_key_scan();
  /* Debounce the keypad in case we caught it bouncing */
  count = bounce_length/2;     /* initialize count */
  while ( (count > 0) && (count < bounce_length) ){
    delay_X_ms(1);            /* Delay a millisecond */
    new_key = hex_key_scan(); /* Read the keyboard again */
    if (new_key == get_key) ++count;
    else --count;
  }
```

```
  /* IF count == bounce_length then get_key has been read
   * long enough to quit bouncing so return it */
  if (count == bounce_length) return(get_key);
  /* Otherwise, the new_key was read so return it */
  else return(new_key);
}
```

15.3 Simple Display Devices

The most simple display device is a single light-emitting diode (LED). An LED lights when current of 10 to 20 mA is passed in the forward direction. Figure 15-8 shows how to drive a single LED. In designing an LED driver, you must determine the output current capability (sourcing or sinking) of the device turning on the LED. In Figure 15-8a, a low-power Schottky 74LS04 can sink up to 16 mA but can source only 400 μA. Therefore, by using the inverter, logic 1 at the inverter's input will turn the LED on. The current-limiting resistor, R, is designed to limit the current through the diode. In Figure 15-8b a latch is used as an *output device* to latch

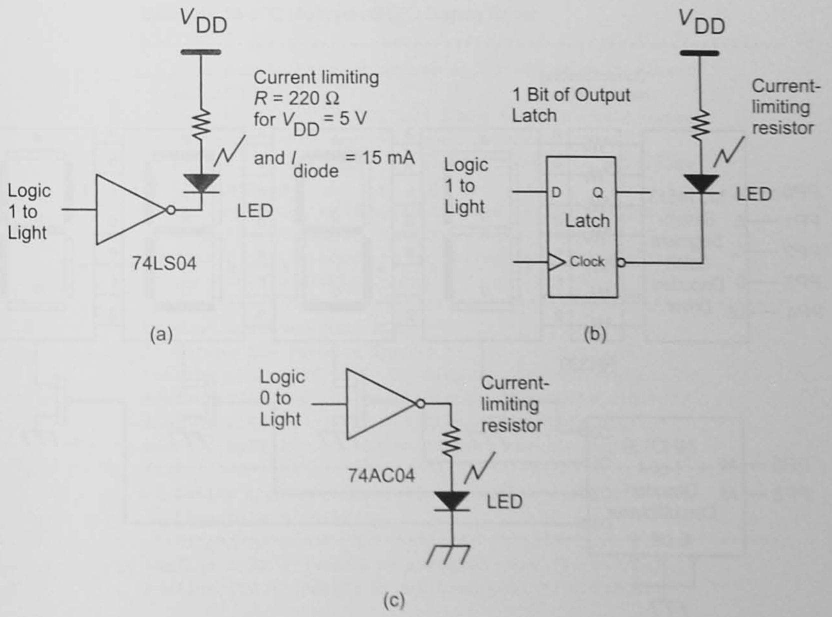


Figure 15-8 Single LED driver circuits.

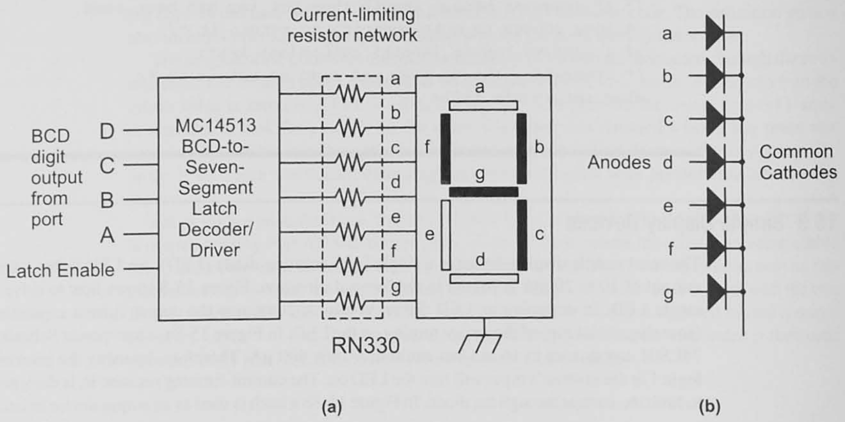

Current-limiting resistor network
BCD digit output from port
D
C
B
A
Latch Enable
MC14513 BCD-to-Seven-Segment Latch Decoder/ Driver
a
b
c
d
e
f
g
RN330
Anodes
Common Cathodes
(a)
(b)

Figure 15-9 (a) Common anode seven-segment display. (b) LED showing anodes and cathodes.

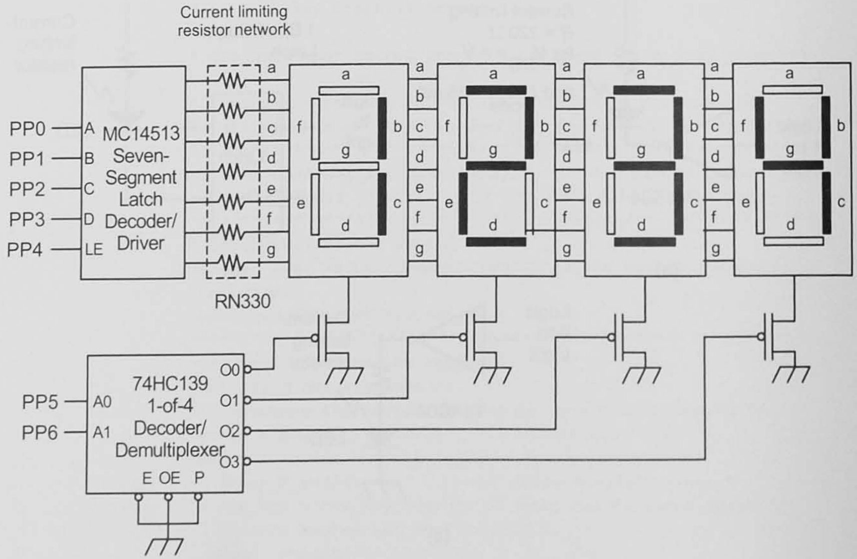

Current limiting resistor network
PP0
PP1
PP2
PP3
PP4
A
B
C
D
LE
MC14513 Seven-Segment Latch Decoder/ Driver
RN330
a
b
c
d
e
f
g
PP5
PP6
A0
A1
74HC139 1-of-4 Decoder/ Demultiplexer
O0
O1
O2
O3
E OE P

Figure 15-10 Multiplexed LED display.

1s and 0s to keep the LED on or off. In Figure 15-8c a 74AC04 with ± 24 mA drive capability can drive the LED connected to the ground. To use an LED in this configuration, you must make sure that the device can source sufficient current to turn on the LED.

A seven-segment LED display shows numeric characters. LED displays come in two varieties, common anode and common cathode. Figure 15-9a is a common cathode display using a MC14513 BCD-to-seven-segment decoder/driver. A BCD number is output by the CPU to the MC14513, and its active-high outputs turn on the appropriate segments to display the number.

Sometimes more than one display is required. Figure 15-10 shows how to multiplex a four-digit display by using only one decoder/driver and common cathode LEDs. Four bits from an output port or a decoder illuminates each of the four digits in turn. The information on each display is output on the seven segment lines from a port or an active-high, seven-segment decoder such as an MC14513. This is called a refreshed display, and if each display is turned on at a greater rate than about 20 Hz, our eyes will not detect any flickering. See Example 15-8.

If your microcontroller has sufficient I/O lines, you can eliminate the MC14513 in Figure 15-9a and the MC14513 and 74HC139 chips in Figure 15-10.

Example 15-8 C Multiplexed LED Display Driver

```
/*************************************************************
 * This sample program drives a four-digit multiplexed
 * BCD LED display.
 * Port P, bits 0, 1, 2, and 3 are the BCD digit output
 * to the MC14513 seven-segment decoder driver.
 * Port P, bits 4 and 5 is a two-bit code to multiplex
 * the display.
 * The input to the display routine is the 4-digit BCD
 * number - thousands, hundreds, tens, ones.
 * This function needs to be called repetitively to
 * refresh the display.
 ************************************************************/
typedef unsigned char BCD;
/* Define the decoder inputs */
#define DISP_1000 0x00  /* Display the 1000's digit */
#define DISP_100  0x10  /* Display the 100's digit */
#define DISP_10   0x20  /* Display the 10's digit */
#define DISP_1    0x30  /* Display the 1's digit */
/*************************************************************
 * Define the microcontroller specific I/O ports used
 * Freescale MC9S12C32
 ************************************************************/
#define DDRP (*(volatile unsigned char *)  0x025a)
#define PTP (*(volatile unsigned char *)   0x0259)
```

```
void delay_X_ms( unsigned int X);     /* X millisecond delay */
/*********************************************************/
void led_mux( BCD thousands,BCD hundreds,BCD tens,BCD ones) {
/*********************************************************/
  /*******************************************************
   * Initialize your microcontroller's I/O port connected
   * to the multiplexed LEDs.
   *******************************************************/
  /* Check to see if Port P, bits 0 - 5 are output */
    if ((DDRP & 0x3F) != 0x3F) {
    /* THEN set the data direction register bits for output */
      DDRP = DDRP | 0x3F;  /* Set bits 5 - 0 */
  }
  /* Output the thousands digit */
    PTP = thousands + DISP_1000;
    delay_X_ms( 50 );
  /* Output the hundreds digit */
    PTP = hundreds + DISP_100;
    delay_X_ms( 50 );
  /* etc for the rest of the digits */
    PTP = tens + DISP_10;
    delay_X_ms( 50);
    PTP = ones + DISP_1;
    delay_X_ms( 50 );
}
```

15.4 Parallel I/O Expansion

Although microcontrollers have many parallel I/O lines, you may have an application that requires more I/O bits or some specialized I/O devices. One solution is to build an expanded mode system with external address, data and control buses, and external I/O devices; or you might chose a different microcontroller with more I/O. Figure 15-11 shows another solution useful for simpler systems without a requirement for high-speed I/O.

One bidirectional port, such as Port AD, can be used to emulate an external bidirectional data bus. Device selection, normally done by decoding an address, can be done in this case with bits from a second, output port, such as Port T. We saw in Chapter 9 that the fundamental component of an output port is a latch; for an input port, it is a three-state gate. Figure 15-11 shows two 8-bit output latches and two 8-bit input three-state gates. You can expand the number of these input and output interfaces depending on the number of select signals available in Port T. If more devices are needed, a decoder can be added to the output on Port T. Examples 15-9 and 15-10 shows modules to use for inputting and outputting data.

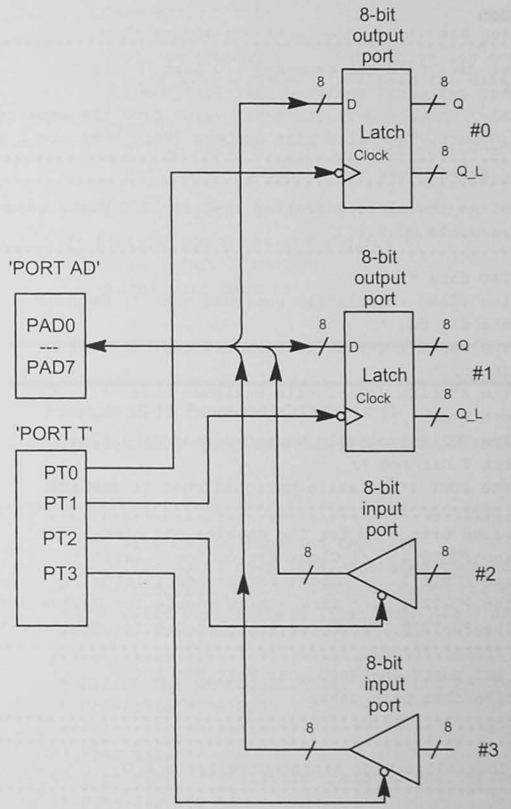


Figure 15-11 Parallel I/O expansion.

Expanded I/O Routines in C

Example 15-9 Expanded I/O Input in C

Write a function to input data from either of the two input interfaces shown in Figure 15-11.

Solution

```
/************************************************************
 * Get data from the expanded I/O port
 * char get_port( unsigned char Port_Num );
 * This function gets the 8-bit value from the expanded
 * I/O port. The valid port numbers (Port_Num) are 2 and 3
 ************************************************************/
/***********************************************************
 * Define the microcontroller specific I/O ports used on a
 * Freescale MC9S12C32
 ************************************************************/
/* PTAD data */
#define PTAD  (*(volatile unsigned char *) 0x0270)
/* Data dir reg */
#define DDRAD (*(volatile unsigned char *) 0x0272)
/* ATD Input */
#define ATDDIEN (*(volatile unsigned char *) 0x008D)
/* Port T data */
#define PTT (*(volatile unsigned char *) 0x0240)
/* Port T dir reg */
#define DDRT (*(volatile unsigned char *) 0x0242)
/************************************************************/
/* Define bits used for the enables and clocks */
#define PORT0 1   /* Clock for device 0 latch */
#define PORT1 2   /* Clock for device 1 latch */
#define PORT2 4   /* Three-state enable for device 2 */
#define PORT3 8   /* Three-state enable for device 3 */
/************************************************************/
char get_port( unsigned char Port_Num ) {
volatile char port_data;
/************************************************************/
  /***********************************************************
   * Initialize your microcontroller's I/O
   **********************************************************/
  /* Make Port AD an input port */
    DDRAD = 0;
    ATDDIEN = 0xFF;
  /**********************************************************/
  /* Make Port T an output port. Set the Port T bits high
   * first before setting the direction */
    PTT = PORT0 | PORT1 | PORT2 | PORT3;
  /* Set the direction bits in DDRT */
    DDRT = PORT0 | PORT1 | PORT2 | PORT3;
  /***********************************************************/
  /* Get the data from the required port */
  /* Make sure a valid port number is given */
    switch ( Port_Num ) {
```

```
      case 2: {
        /* Enable the three-state gate and get the data */
        PTT &= ~PORT2;          /* Active low enable */
        port_data = PTAD;       /* Read the data */
      }
      case 3: {
        PTT &= ~PORT3;          /* Active low enable */
        port_data = PTAD;       /* Read the data */
      }
    }
  /* Disable the three-state gates */
    PTT |= (PORT2 | PORT3);
    return( port_data );
}
```

Example 15-10 Expanded I/O Output in C

Write a function to output data to either of the two output interfaces shown in Figure 15-11.

Solution

```
*************************************************************
 * Put data to the expanded I/O port
 * void put_port( char data, unsigned char Port_Num);
 * This function puts an 8-bit value to the expanded I/O port.
 * the valid port numbers (Port_Num) are 0 and 1
 ************************************************************/
/************************************************************
 * Define the microcontroller specific I/O ports used on a
 * Freescale MC9S12C32
 ************************************************************/
/* PTAD data */
#define PTAD  (*(volatile unsigned char *) 0x0270)
/* Data dir reg */
#define DDRAD (*(volatile unsigned char *) 0x0272)
/* ATD Input */
#define ATDDIEN (*(volatile unsigned char *) 0x008D)
/* Port T data */
#define PTT (*(volatile unsigned char *) 0x0240)
/* Port T dir reg */
#define DDRT (*(volatile unsigned char *) 0x0242)
/************************************************************/
/* Define bits used for the enables and clocks */
#define PORT0 1   /* Clock for device 0 latch */
#define PORT1 2   /* Clock for device 1 latch */
#define PORT2 4   /* Three-state enable for device 2 */
#define PORT3 8   /* Three-state enable for device 3 */
```

```
/**********************************************************/
void put_port( char data, unsigned char Port_Num ) {
/**********************************************************/
  /*********************************************************
   * Initialize your microcontroller's I/O
   *********************************************************/
  /* Make sure Port AD is an output port */
    DDRAD = 0xFF;
  /* Make Port T an output port. Set the Port T bits high
   * first before setting the direction */
    PTT = PORT0 | PORT1 | PORT2 | PORT3;
  /* Set the direction bits in DDRT */
    DDRT = PORT0 | PORT1 | PORT2 | PORT3;
  /**********************************************************/
  /* Make sure a valid port number is given */
    switch ( Port_Num ) {
      case 0: {
        /* output the data to the port */
        PTAD = data;
        /* Strobe the latch signal active low and then high */
        PTT &= ~PORT0;
        PTT &= ~PORT0;  /* Give the latch time to settle */
        PTT |= PORT0;
      }
      case 1: {
        PTAD = data;
        /* Strobe the latch signal active low and then high */
        PTT &= ~PORT1;
        PTT &= ~PORT1;    /* Do a little time */
        PTT |= PORT1;
      }
    }
    /* Make PTAD an input port again */
    DDRAD = 0;
}
```

15.5 Parallel I/O Electronics

We need some additional design to interface our somewhat fragile electronics to the real, sometimes cruel, world. We must take care to protect our electronics from overvoltages and static discharges and to provide signal levels compatible with the logic circuits we are using. Although good printed circuit board design, shielding, and power supply design are outside the scope of this text, we present here some simple interface circuits for digital input and output.

Input Electronics

Figure 15-12 shows a simple input interface. The two 1N4001 (or similar) diodes limit the voltage excursion on the digital input signal to a maximum of a diode drop higher than V_{DD} and to a minimum of one diode drop below ground. If the input signal contains high frequencies with fast rise and fall times, resistor R_1 can provide an impedance match to the driving circuit. It may be eliminated for low-frequency signals. The 1 kΩ series resistor provides some current limiting and further protection for the microcontroller's input pin.

Figure 15-13 shows a transient voltage suppression (TVS) diode, also called a *transorb*. This device, which is available in unipolar and bipolar styles, is designed to clamp high-voltage transients. It operates much like a zener diode but reacts very quickly to high-speed voltage spikes. The device does have some internal capacitance, which may degrade high-frequency signals.

Transient voltage protection may not be needed where voltage excursions are not extreme. For such situations, a rail-to-rail amplifier voltage follower, such as the OPA4344 shown in Figure 15-14, can be used. The voltage follower ensures that even though the input voltage may exceed V_{DD}, the output to the microcontroller will not be greater than this limit.

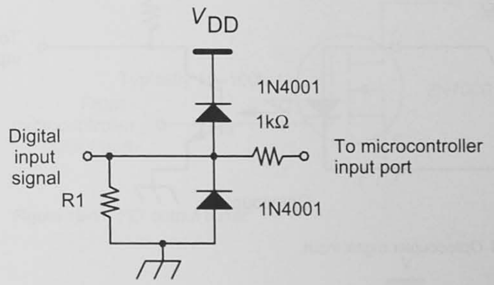


Figure 15-12 Digital input.

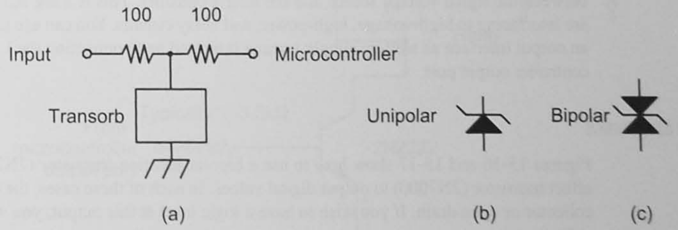


Figure 15-13 (a) Transient voltage suppression diode; (b) Unipolar; (c) Bipolar.

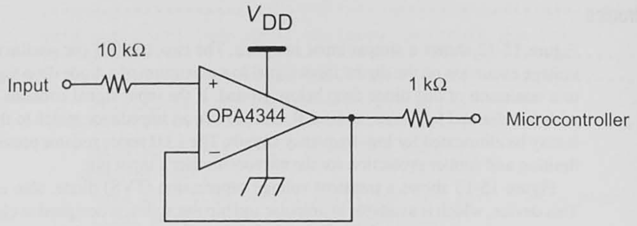
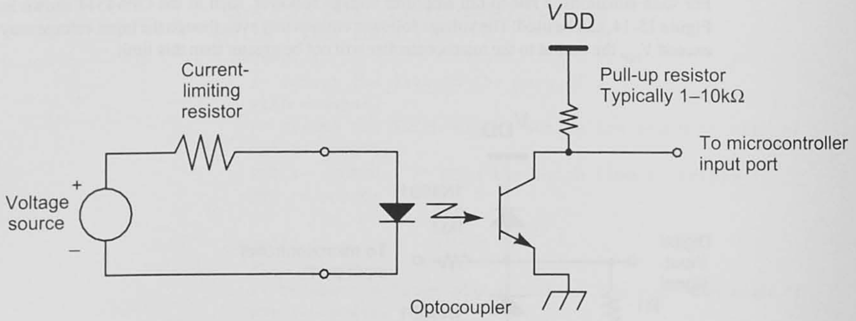


Figure 15-14 Voltage follower.



V_{DD}
Current-limiting resistor
Pull-up resistor Typically 1–10kΩ
To microcontroller input port
Voltage source
Optocoupler

Figure 15-15 Optocoupler digital input.

Figure 15-15 shows a circuit to use when the input signal is from a high-voltage source. The optocoupler is a light-emitting diode with a phototransistor. The current-limiting resistor is designed to provide the correct current for the LED. Notice that there is no physical connection between the signal voltage source and the microcontroller. This is a big advantage when you are interfacing to high-voltage, high-power, and noisy circuits. You can use the optocoupler as an output interface as well by simply turning it around and connecting the LED to the microcontroller output port.

Output Electronics

Figures 15-16 and 15-17 show how to use a bipolar junction transistor (2N2222) and a field effect transistor (2N7000) to output digital values. In each of these cases, the output is an open collector or open drain. If you wish to have a logic level at this output, you will have to add a pull-up resistor.

Microcontrollers often have to drive relays. Figure 15-18 shows a relay driver using the 2N2222 transistor. Relays often have multiple contacts, some of them normally open (NO) and some normally closed (NC), where normally refers to the relay not being energized. The 1N4001 clamp diode across the relay coil, an important addition to this circuit, limits the voltage spike produced by the coil to one diode drop greater than *V* when the relay is de-energized. A voltage spike greater than 100 V can be generated in the course of switching off a 12 V relay. Make sure you include the clamp diode in all your circuits that drive an inductive load!

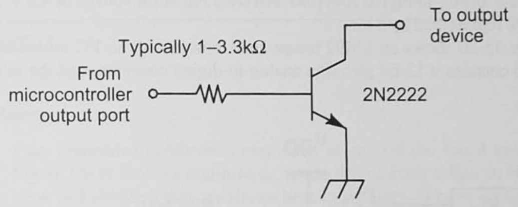


Figure 15-16 Transistor output buffer.

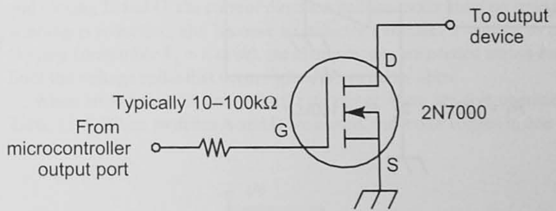


Figure 15-17 FET output buffer.

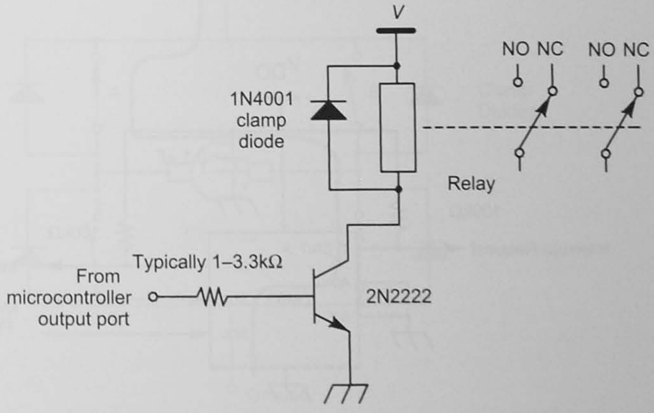


Figure 15-18 Transistor relay driver.

15.6 Temperature Measurements

There are a variety of device for measuring temperature, including thermistors, thermocouples, and solid-state temperature sensors. Figure 15-19 shows an LM19 solid-state sensor. When V_{DD} of 2.4 to 5.5 V is applied, the output voltage ranges between 2.4 and 0.3 V for temperatures ranging from −55 to +130°C. By following the sensor with an amplifier whose gain, $(R_1 + R_2)/R_1$, is 2, the analog-to-digital converter will see a full-scale voltage of 4.8 V which can conveniently be represented by 8 bits.

Figure 15-20 shows an LM92 temperature sensor with an I²C microcontroller interface. The chip contains a 12-bit plus sign analog-to-digital converter, and the microcontroller can

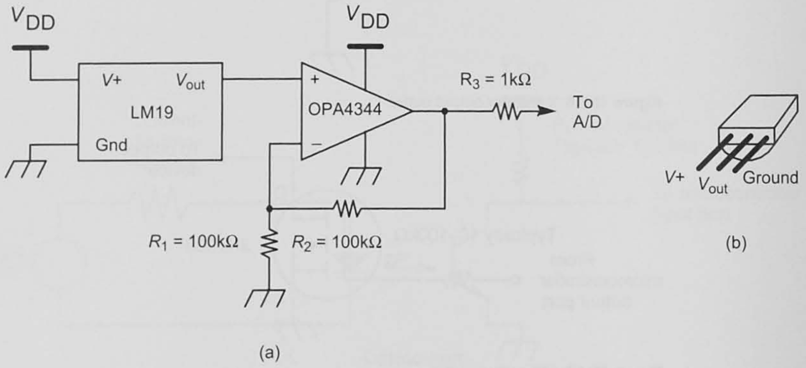


Figure 15-19 LM19 temperature sensor. (a) Sensor with amplifier. (b) TO92 case.

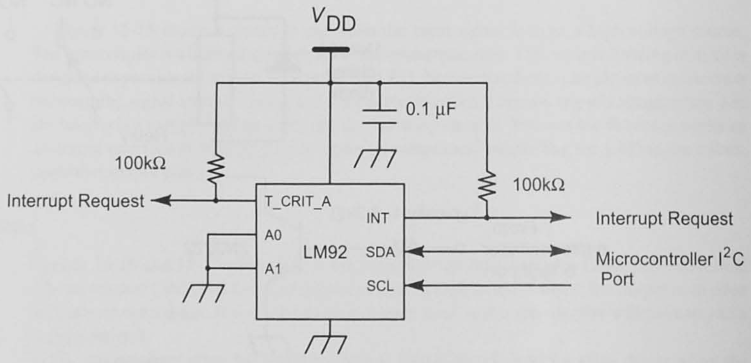


Figure 15-20 LM92 temperature sensor with I²C interface.

read the temperature at any time by interrogating the chip on the I²C bus. The two address bits, A1–A0, select up to four devices. With both grounded, the device will respond to address 00. The LM92 can also be set up to act as a comparator and can generate an interrupt when the temperature exceeds a programmable set value. The amount of hysteresis that the temperature can change before the alarm condition resets is programmable as well.

15.7 Motor Control

Direct-Current Motors

Your embedded application may need to control the speed and direction of a dc motor. Figure 15-21 shows a common dc motor drive circuit called an H-bridge. Four switches are used, and although they are shown here as switches, in most applications they are transistors. As shown, with switches A and D closed and B and C open, the current direction flows through the dc motor from left to right, causing the motor to rotate clockwise. Upon opening A and D and closing B and C, the current direction and the motor rotation reverse. Because the motor winding is inductive, and because an inductor generates a voltage to try to keep the current flowing (remember $V_L = L\,di/dt$), the clamp diodes are needed across each switch to reduce or limit the voltage spike that occurs when the switches open.

When an H-bridge is controlling a dc motor, there are five operating states, as shown in Table 15-3. When switches A and D are closed, the motor rotates in one direction; when B and

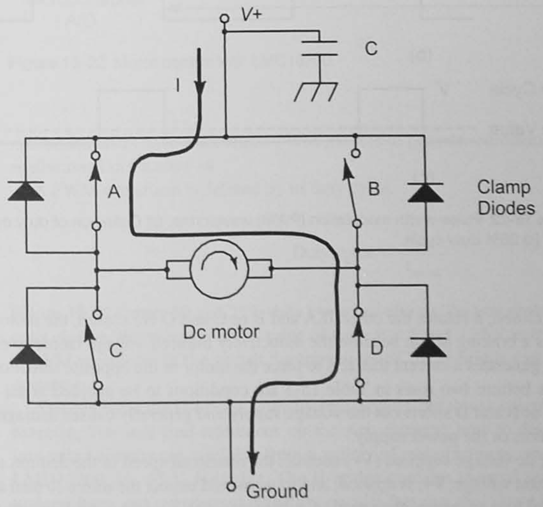


Figure 15-21 Simplified motor control H-bridge.

Table 15-3 H-Bridge Motor Control

	A	B	C	D	Motor
1	Open	Open	Open	Open	Free wheeling
2	Closed	Open	Open	Closed	Rotate right
3	Open	Closed	Closed	Open	Rotate left
4	Closed	Closed	Open	Open	Braking
5	Open	Open	Closed	Closed	Breaking
6	Closed	Don't care	Closed	Don't care	DO NOT DO!
7	Don't care	Closed	Don't care	Closed	DO NOT DO!

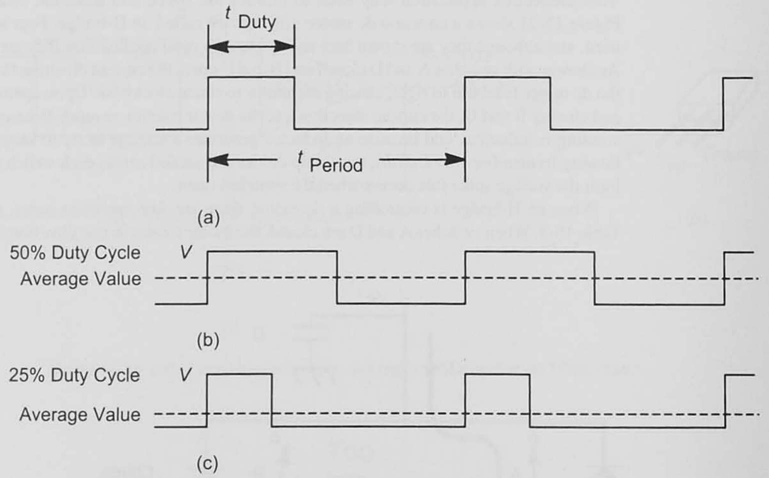


Figure 15-22 Pulse-width modulation (PWM) waveforms. (a) Definition of duty cycle. (b) 50% duty cycle. (c) 25% duty cycle.

C are closed, it rotates the other. If A and B or C and D are closed, the motor is shorted. This creates a braking action because the inductively induced voltage (sometimes called the back EMF) generates a current that acts to force the motor in the opposite direction.

The bottom two rows in Table 15-3 are conditions to be avoided at all costs. Closing A and C or B and D shorts out the voltage supply and generally causes damage to the switching transistors or the power supply.

The dc voltage supplied ($V+$) controls the rotational speed of the dc motor. In Figure 15-21 a constant voltage, $V+$, is applied, and so we would expect the motor to turn at a constant rpm. A useful way to control the speed of a dc motor with a microcontroller is to generate a pulse-width modulation (PWM) waveform, as shown in Figure 15-22. This is so common that many

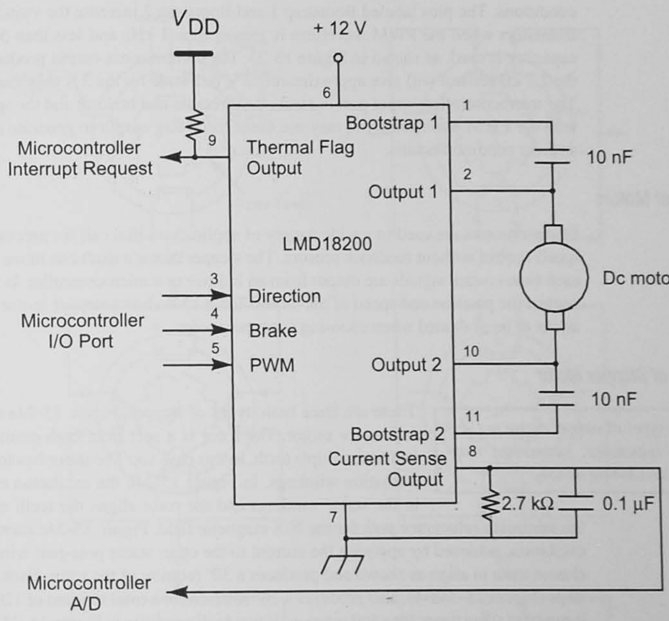


Figure 15-23 Motor control with LMD18200.

microcontrollers have a dedicated PWM module to generate automatically PWM waveforms, as discussed in Chapter 14.

A PWM waveform is defined by its duty cycle.

$$\text{Duty cycle} = \frac{t_{duty}}{t_{period}}$$

Figure 15-22 shows 50 and 25% duty cycle waveforms. As you probably learned in your circuits classes, the average, or dc, value of these is 0.5 and 0.25 V, respectively. Thus, applying a PWM waveform to the switch the transistors on the H-bridge can control the speed of the dc motor.

You can build the H-bridge from discrete circuits, but this is a significant electronics design exercise. You will find references on the web showing how to design a discrete H-bridge. Integrated circuits are available from a variety of manufacturers, and Figure 15-23 shows an LMD18200 3A, 55 V, H-bridge. The H-bridge contains thermal sensing circuits, as well as undervoltage and overcurrent detection circuits that can shut down the motor to protect it. A current sense output and a thermal flag output allow the microcontroller to monitor these error conditions. The pins labeled Bootstrap 1 and Bootstrap 2 increase the switching speed of the transistors when the PWM waveform is greater than 1 kHz and less than 500 kHz. A 10 nF capacitor is used, as shown in Figure 15-23. The current sense output produces 377 μA/A, so the 2.7 kΩ resistor will give approximately 3 V full-scale for the 3 A maximum motor current. The microcontroller output port controls the direction and braking and the speed of the motor with the PWM waveform; you may use the thermal flag output to generate an interrupt when an error condition occurs.

Stepper Motors

Stepper motors are used in a wide variety of applications that call for precise positioning and speed control without feedback sensors. The stepper motor's shaft can move a precise amount each time control signals are output from an indexer or a microcontroller. In this way you can control the position and speed of the motor. Table 15-4 shows stepper motor design considerations to be evaluated when choosing a stepper motor.

Types of Stepper Motor

Three types of stepper motor are *variable reluctance, permanent magnetic,* and *hybrid* motors.

There are three basic types of stepper. Figure 15-24a shows a *variable reluctance* motor. The rotor is a soft iron, high-permeability material and has multiple teeth, in this case six. The stator has four poles and two excitation windings. In Figure 15-24b the excitation current is applied to the stator windings and the rotor aligns the teeth as shown. This is the minimum reluctance path for the N-S magnetic field. Figure 15-24c shows the next stator excitation, achieved by applying the current to the other stator pole-pair winding. It pulls the closest teeth to align as shown and produces a 30° rotation of the rotor. Each of the following steps (Figure 15-24d–f), also produces a 30° rotation for a total rotation of 120°. The sequence is repeated three times for a full rotation. If you look carefully at Figure 15-24, you might wonder how the rotor knows which way to rotate when the applied field changes from (b) to (c). In practice, the teeth are aligned so that clockwise and counterclockwise rotation of the applied field will produce rotation in opposite directions. The variable reluctance motor has fairly low torque and so is used in small-equipment-positioning applications.

The *permanent magnet* stepper uses permanent magnets for the rotor. In principle, it looks like Figure 15-25a and operates with a rotating excitation field as described for the variable reluctance motor. Figure 15-25b shows a more realistic view of the permanent magnet rotor. It can have many more magnets than shown in Figure 15-25a, and with more stator poles, it can make much smaller steps than the 30° shown in this example. The permanent magnet motor is also called a *canstack motor*.

Table 15-4 Stepper Motor Design Considerations

Stepper type	Variable reluctance, permanent magnetic, hybrid
Winding type	Bifilar, unifilar
Excitation type	Unipolar, bipolar
Phases	Two or more
Degrees/step	0.9, 1.8, 3.6, 7.5, 15
Stepping modes	Full-step, half-step, microstep
Torque	Starting torque, start and stop region

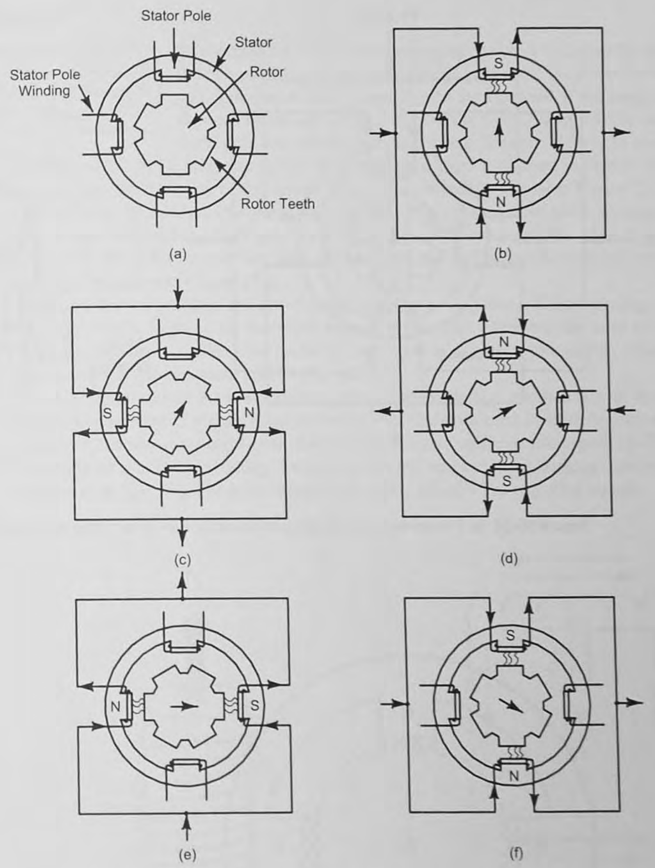


Figure 15-24 Stepper motor operation. (a) Variable reluctance motor. (b) Starting position. (c) First step, 30°. (d) Second step, 60°. (e) Third step, 90°. (f) Fourth step, 120°.

A *hybrid stepper motor* combines the attributes of the permanent magnet and the variable reluctance motor. Figure 15-26 is a cross sectional view of a hybrid stepper motor. The rotor has a fine-pitch tooth structure as shown, and it can form one pole of a magnet. The teeth in the rotor line up with teeth on the stator, minimizing the magnetic reluctance path. The teeth are offset a small amount on the next stator winding to be energized. Typical step sizes range from 0.9°/step to 3.6°/step.

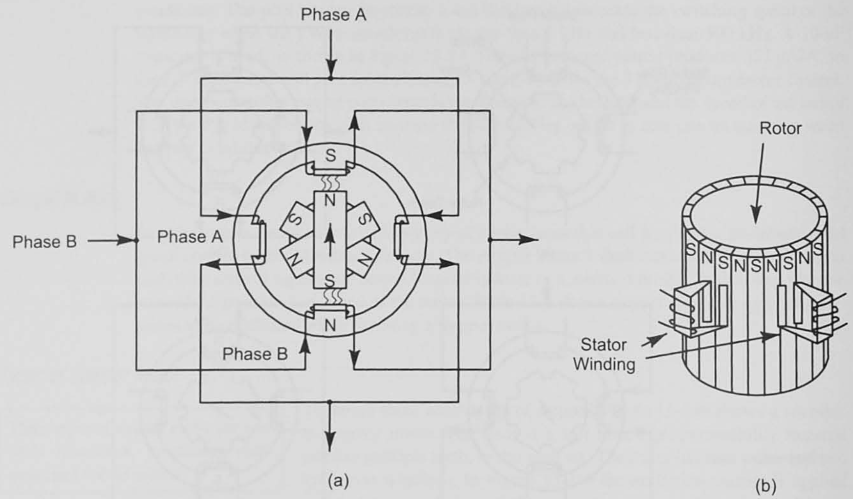


Figure 15-25 (a) Permanent magnet stepper motor. (b) Permanent magnetic "canstack."

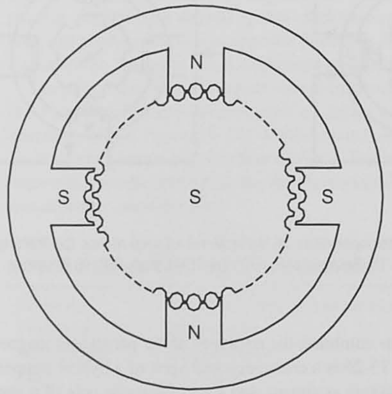


Figure 15-26 Hybrid stepper motor.

Stepper Motor Windings

Stepper motor windings are designed so the current direction can be reversed to reverse the rotation direction.

To energize the stator's electromagnetic poles, a coil must be wound on the two poles that make the magnetic pole pair. As current flows in the coil, a magnetic field is formed, and the direction of the magnetic field depends on the direction of the current flow. Figure 15-27a shows the north pole that results from the current direction shown. To reverse the direction of the field, as shown in the stepper motor rotation sequence in Figure 15-24, we must reverse the direction of the current in the stator winding. Compare Figure 15-24b and 15-24d. There are two ways the stator field windings are constructed: *unipolar* winding (Figure 15-27b) and *bipolar* winding (Figure 15-27c). As we will see, the unipolar winding is simpler to drive by using two output bits from our microcontroller. The bipolar winding requires an H-bridge, as shown in Figure 15-21.

Stepper motor windings are also classed as *bifilar* or *unifilar*. A bifilar winding is one in which the coil is wound with two wires instead of one. This eliminates the need for a center tap on the winding and allows the motor to operate either as bipolar or unipolar. However, the bifilar winding requires more space in the stator.

Most stepper motors have two winding *phases*, although there are motors with three or five phases. The number of phases refers to the number of signals used to drive the stepper motor. Figure 15-28a shows an eight-pole (four pole pairs), two-phase motor. Figure 15-28b shows one phase of a unipolar winding. Two-phase motors that use these windings are sometimes referred to as *four-phase* motors because each phase actually has two drive signals.

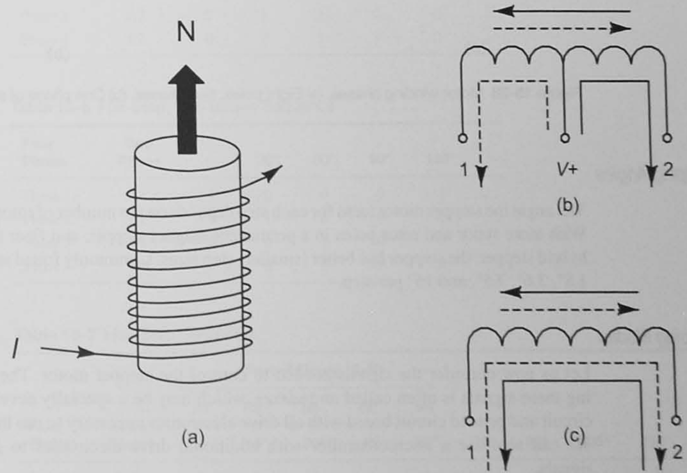


Figure 15-27 (a) Magnetic field direction depends on the current direction. (b) Unipolar winding. (c) Bipolar winding.

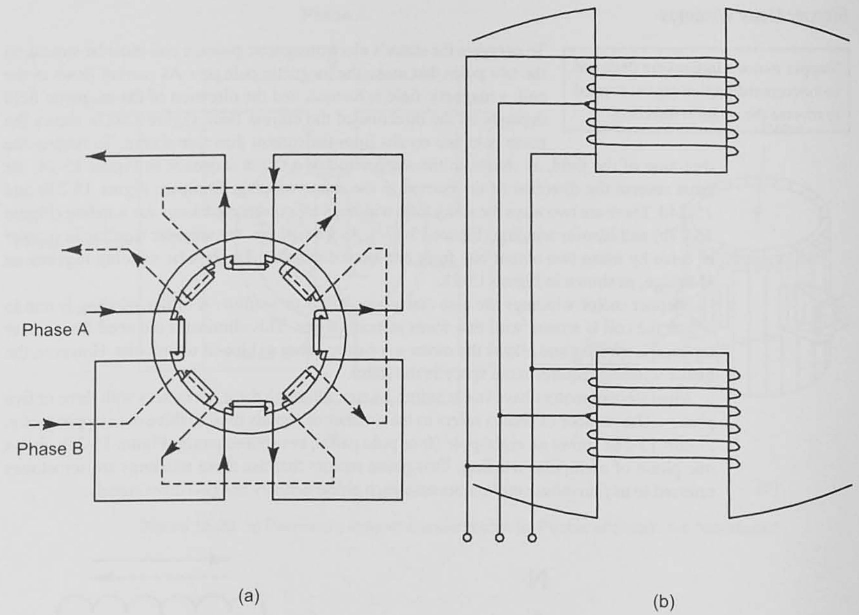


Figure 15-28 Motor winding phases. (a) Eight poles, two phases. (b) One phase of a unipolar winding.

Stepping Angles

The angle the stepper motor turns for each step depends on the number of rotor and stator poles. With more stator and rotor poles in a permanent magnet stepper, and finer tooth pitch in the hybrid stepper, the stepper has better (smaller) step sizes. Commonly found step sizes are 0.9°, 1.8°, 3.6°, 7.5°, and 15° per step.

Stepping Modes

Let us now consider the signals needed to control the stepper motor. The device generating these signals is often called an *indexer*, which may be a specially developed integrated circuit and printed circuit board with all drive electronics necessary to run the stepper motor. We can also use a microcontroller with additional drive electronics to generate control signals.

Full-Step Sequence: Figure 15-29 shows a permanent magnet stepper with a six-pole rotor and four-pole stator. This steps 30° per step. The sequence starts with unipolar windings A1 and B1 energized (Figure 15-29a). A sequence of four full steps (Figure 15-29b–e) results in

120° rotation. Windings A1–B1–A2–B2 are energized in the sequence shown in Table 15-5. This sequence is also called *full-step, full-torque*. Table 15-6 shows a full-step, low-torque sequence, where only one winding is excited at a time. Table 15-5 also shows the winding designations when the unipolar stepper is said to have four phases.

Half-step sequence: A half-step sequence is shown in Table 15-7 and Figure 15-30. As you might expect, each step now is 15° and is produced by the excitation sequence shown. The half-step sequence will produce a smoother operation of the stepper but with less motor torque.

Microstep sequence: Both the full-step and half-step sequences produce motion that is not a smooth rotation because the step size is finite. The steppers with smaller step sizes are smoother. Microstepping increases the step resolution beyond that limited by the number of poles. Figure 15-31 shows two phases being driven by stepped sinusoids that are 90° out of phase. Controllers for microstepping motors can produce up to 500 microsteps per full step, resulting in a much smoother stepper motor rotation.

Table 15-5 Full-Step, Full-Torque Sequence

		Part of Figure 15-29					
		(a)	(b)	(c)	(d)	(e)	...
Four Phase	**Two Phase**		**30°**	**60°**	**90°**	**120°**	
Phase 1	A1	1	0	0	1	1	...
Phase 2	B1	1	1	0	0	1	...
Phase 3	A2	0	1	1	0	0	...
Phase 4	B2	0	0	1	1	0	...

Table 15-6 Full-Step, Full-Torque Sequence

Four Phase	Two Phase		30°	60°	90°	120°	
Phase 1	A1	1	0	0	0	1	...
Phase 2	B1	0	1	0	0	0	...
Phase 3	A2	0	0	1	0	0	...
Phase 4	B2	0	0	0	1	0	...

Table 15-7 Half-Step Sequence

		Part of Figure 15-30									
		(a)	(b)	(c)	(d)	(e)					
Four Phase	**Two Phase**		**15°**	**30°**	**45°**	**60°**	**75°**	**90°**	**105°**	**120°**	**...**
Phase 1	A1	1	0	0	0	0	0	1	1	1	...
Phase 2	B1	1	1	1	0	0	0	0	0	1	...
Phase 3	A2	0	0	1	1	1	0	0	0	0	...
Phase 4	B2	0	0	0	0	1	1	1	0	0	...

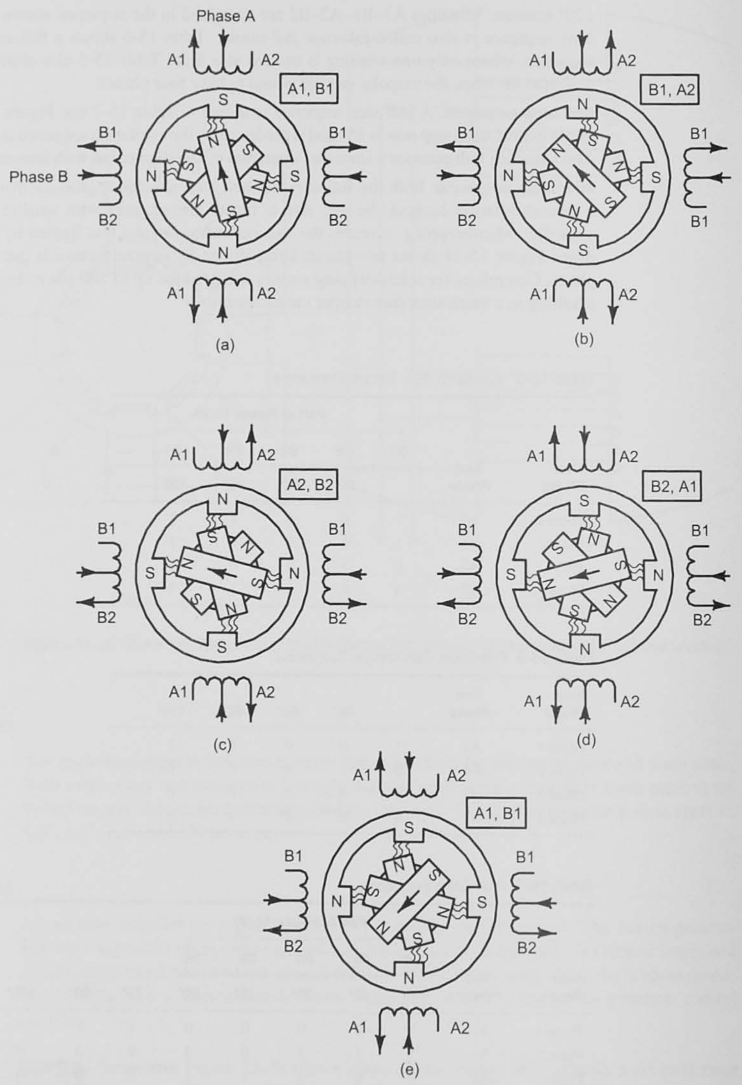


Figure 15-29 Full-step, full-torque sequence.

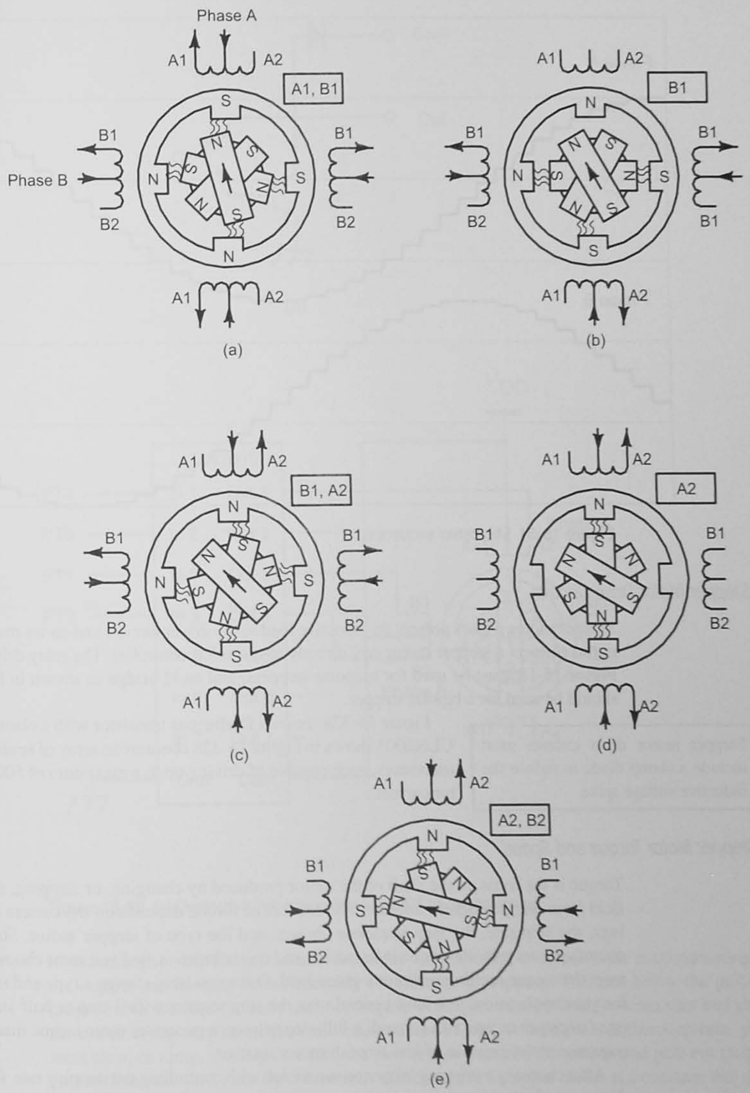


Figure 15-30 Half-step sequence.

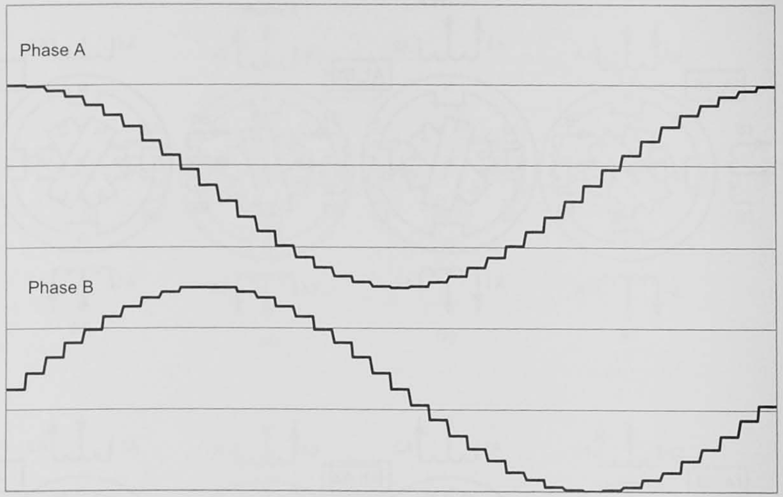


Figure 15-31 Microstep sequence.

Stepper Motor Drive Circuits

Stepper motor phases present an inductive load to the driver circuit, and so we must be careful not to connect a stepper motor coil directly to our microcontroller. The relay driver circuit of Figure 15-18 must be used for unipolar steppers, and an H-bridge as shown in Figure 15-23 should be used for a bipolar stepper.

Stepper motor drive circuits must include a clamp diode to reduce the inductive voltage spike.

Figure 15-32a shows a Darlington transistor with a clamp diode. The ULN2003 shown in Figure 15-32b contains an array of seven Darlington transistors, each capable of driving up to a maximum of 500 mA collector current.

Stepper Motor Torque and Speed

Torque is the force on the shaft of the motor produced by changing, or stepping, the magnetic field from one position to another. The amount of torque depends on the current in the windings, the step rate, the step sequence chosen, and the type of stepper motor. Stepper motor manufacturers provide torque information and characteristics, and you must choose a motor to meet the torque requirements for a given load. Once you have chosen a type and size of motor for your application, you have control over the step sequence (full-step or half-step) and step rate (steps per second). In general, a full-step sequence generates more torque than a half-step sequence at the expense of less smooth motor rotation.

After choosing a stepping sequence, we are left with controlling the stepping rate. Figure 15-33 shows a typical stepper motor torque versus step rate characteristic. The *pull-out torque* curve defines the maximum torque the motor can develop at a given step rate. For a given motor load,

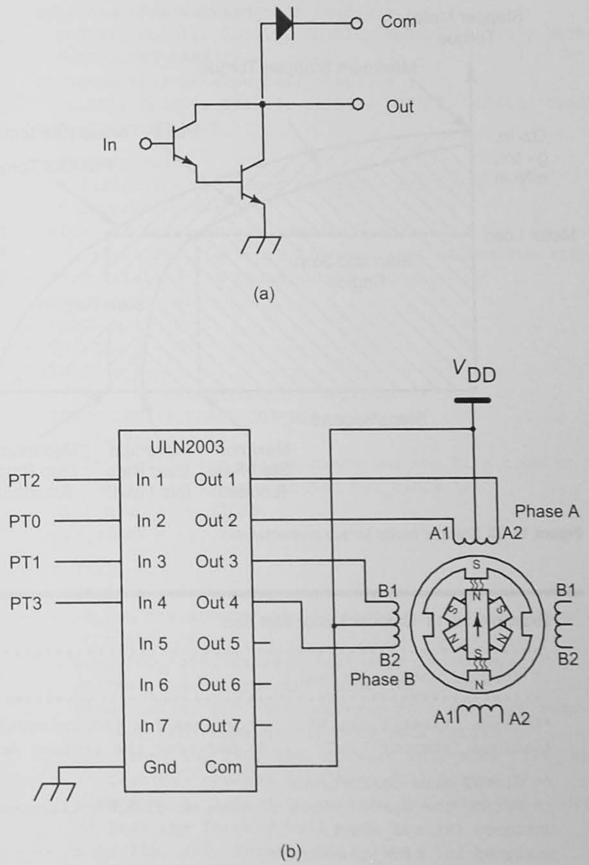


Figure 15-32 (a) Darlington transistor. (b) ULN2003 transistor array.

if you attempt to step faster than this, the motor will stall. *Pull-in torque* is the maximum torque the stepper can develop instantaneously for starting or stopping. The area below the pull-in torque curve is called the *start and stop region*. For a given motor load, you can start and stop the motor at any step rate below the maximum start rate. To achieve a higher step rate, you must slew, or ramp, the step rate into the *slew region* between the pull-in and pull-out torque curves. Example 15-11 shows C functions that can drive a stepper motor at a constant rate and ramp the motor speed up and down. Examples 15-12 through 15-15 show how components of the test bed can be programmed.

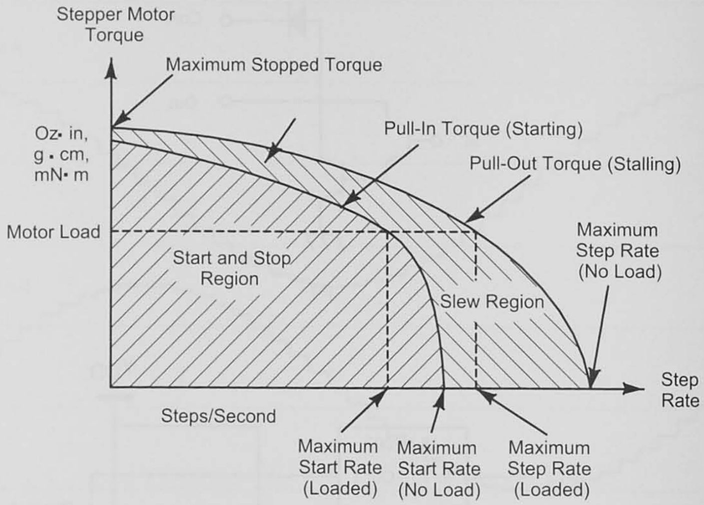


Figure 15-33 Stepper motor torque characteristic.

Example 15-11 Stepper Motor Test Bed

```
/*************************************************************
 * Stepper motor test bed
 *************************************************************/
#include "timer_ticks.h"    /* Values for the interval timer */
#include "stepper_1.h"      /* Defs for the stepper motor */

/* Global data definitions */
/* Define the system clock ticking at TICKS millisecond */
unsigned int sys_clock;
unsigned int tick_counter;

/* Define the step sequences */
/* B2 A2 B1 A1 */
/* 0 0 0 0 marks the end of the sequence */
unsigned char forward_full_step_low_torque[] = {
  0b0001, 0b0010, 0b0100, 0b1000, END_MARK };
unsigned char forward_full_step_full_torque[] = {
  0b0011, 0b0110, 0b1100, 0b1001, END_MARK };
unsigned char reverse_full_step_low_torque[] = {
  0b1000, 0b0100, 0b0010, 0b0001, END_MARK };
unsigned char reverse_full_step_full_torque[] = {
  0b1001, 0b1100, 0b0110, 0b0011, END_MARK };
```

```
unsigned char forward_half_step[] = {
  0b0001, 0b0011, 0b0010, 0b0110, 0b0100, 0b1100, 0b1000,
  0b1001,END_MARK };
unsigned char reverse_half_step[] = {
  0b1001, 0b1000, 0b1100, 0b0100, 0b0110, 0b0010, 0b0011,
  0b0001, END_MARK };

void main(void) {
 /* Initialize the timer to create the sys_clock
  * interval timer */
 timer_ch_interrupt_initialization();
 /* Initialize the 4-bit I/O port that drives the stepper */
 /* Initialize the phase output signals 0 */
 PHASE_A_1 = 0;
 PHASE_B_1 = 0;
 PHASE_A_2 = 0;
 PHASE_B_2 = 0;
 /* Set the data direction register */
 PHASE_DDR |= PHASE_OUTPUTS;
 Enable Interrupts;
 /* Initialize the system clock and the tick counter */
 /* Number of ms incrementing sys_clock */
 tick_counter = TICKS;
 sys_clock = 0;

 for(;;) {
   /* Test all the step sequences */
   /* Run the stepper at 100 steps/sec for 300 steps at low
    * torque */
   run( 100, 300, forward_full_step_low_torque );
   /* Test the reverse_full_step_low_torque */
   slew_run( 0, 100, 300, reverse_full_step_low_torque);
   /* Test the forward_full_step_full_torque */
   slew_run( 50, 160, 300, forward_full_step_full_torque );
   /* Test the reverse_full_step_full_torque */
   slew_run( 0, 160, 400, reverse_full_step_full_torque );
   /* Test the forward_half_step */
   run( 200, 300, forward_half_step );
   /* Test the reverse_half_step */
   run( 200, 300, reverse_half_step );

 } /* wait forever */
}
```

Example 15-12 Stepper_1.h

```
/***********************************************
 * Definitions for the stepper motors
 ***********************************************/
```

```
/* Define the stepper motor output bits */
/* Port T output bits to use */
/* Port T is used to drive the stepper */
#define PHASES PTT
#define PHASE_A_1 PTT_PTT0   /* Phase A1 (Phase 1) */
#define PHASE_B_1 PTT_PTT1   /* Phase B1 (Phase 2) */
#define PHASE_A_2 PTT_PTT2   /* Phase A2 (Phase 3) */
#define PHASE_B_2 PTT_PTT3   /* Phase B2 (Phase 4) */
#define PHASE_DDR DDRT       /* Data direction register */
#define PHASE_OUTPUTS 0x0f   /* Four bits on port T */

/* Define the end mark for the step sequences */
#define END_MARK 0

/* Function prototypes */

/**********************************************************
 * Run the stepper at steps_per_sec, for number_of_steps steps,
 * and use the step sequence pointed to by p_sequence.
 **********************************************************/

void run( unsigned int steps_per_sec,
          unsigned int number_of_steps,
          unsigned char * p_sequence);

/**********************************************************
 * Slew the stepper from start_steps_per_sec to
 * final_steps_per_sec and then run at steps_per_sec, for
 * number_of_steps steps using the step sequence pointed to
 * by p_sequence.
 **********************************************************/

void slew_run(unsigned int start_steps_per_sec,
          unsigned int final_steps_per_sec,
          unsigned int number_of_steps,
          unsigned char * p_sequence);

/**********************************************************
 * Timer initialization to enable interrupt
 **********************************************************/

 void timer_ch_interrupt_initialization( void );
```

Example 15-13 Timer_ticks.h

```
/**********************************************************
 * Define the bus clock frequency and interval needed
 * to generate the sys_clock that updates at TICKS
 * millisecond intervals
 **********************************************************/
/* Define the timer interval value */
```

```
#define BUS_CLOCK 8000000              /* Bus clock freq */
#define MS_CLOCK_TICKS BUS_CLOCK/1000 /* Bus clocks in ms */
#define TICKS 1                       /* # milliseconds for a
                                       * sys_clock timer tick */
#define TICKS_PER_SEC 1000/TICKS  /* # timer ticks per sec */
```

Example 15-14 Stepper Motor Run: run.c

```
/**********************************************************
 * Stepper drivers -- run the motor at a constant speed
 **********************************************************/
#include "timer_ticks.h"
#include "stepper_1.h"
/* Global data definitions */
/* Define the system clock ticking every TICKS milliseconds.
 * TICKS is defined in stepper_1.h */
extern unsigned int sys_clock;  /* System clock updated by the
                                 * timer every TICKS ms */

/**********************************************************
 * Run the motor at a constant speed
 * void run( unsigned int steps_per_sec,
 *           unsigned int number_of_steps,
 *           unsigned char * p_sequence);
 **********************************************************/
 void run(
      unsigned int steps_per_sec,    /* Motor rate desired */
      unsigned int number_of_steps, /* Number steps to move */
      unsigned char * p_sequence) { /* Pointer to the step
                                      * sequence */

   unsigned int wait_time;           /* Time to wait until
                                      * the next step */
   unsigned char * p_sequence_used;  /* Pointer to step seq */

   p_sequence_used = p_sequence;

   /* The sys_clock is incrementing at TICKS milliseconds */
   /* The number of TICKS to wait is
    * TICKS_PER_SEC/steps_per_sec */
   wait_time = TICKS_PER_SEC/steps_per_sec;
   /* while the number of steps taken is < number_of_steps */
   while (number_of_steps > 0 ){
     /* Output a step pattern to the port */
     PHASES = *p_sequence_used++;
     sys_clock = 0;
     /* Reset the pointer if it is pointing at the end mark */
```

of clock ticks needed in `sys_clock` and waits for that time to output the step sequence at the correct rate.

Example 15-14 shows a function to run the stepper at a constant speed (`steps_per_sec`) for `number_of_steps` steps in the direction and with the type of step sequence pointed to by the `p_sequence` pointer. The time between steps is calculated by `wait_time = TICKS_PER_SEC/steps_per_sec`.

Example 15-15 shows a function that slews, or ramps, the stepper motor from an initial `start_steps_per_sec` to a `final_steps_per_sec`. The motor starts off at `start_steps_per_sec` and increases one step per second per step to `final_steps_per_sec`. It then runs at that speed until the total `number_of_steps` have occurred.

15.8 Conclusion and Chapter Summary Points

In this chapter we have shown a wide variety of application examples using a microcontroller.

- Microcontrollers have internal I/O devices such as analog-to-digital converters, timers, and serial and parallel I/O.
- Most often, external I/O devices can be connected to a microcontroller's internal parallel I/O.
- Switches are used to input binary information.
- Mechanical switches bounce when contact is made, and in some applications software or hardware debouncing must be used.
- Pull-up (or pull-down) resistors must be used with switch inputs to avoid floating inputs.
- Internal pull-up (or pull-down) resistors may be enabled in the microcontroller's I/O port.
- Keypads and keyboards are switches and diodes that connect an output from a I/O port to an input line. The keypad is scanned to determine what key is being pressed.
- When an LED is interfaced, the circuit designer must make sure the output device can supply enough current to light the LED.
- LEDs require a current of several milliamperes to light.
- Parallel I/O expansion can be done with a bidirectional port acting as a data bus.
- Parallel I/O expansion can be done also with the SPI and serial/parallel shift registers.
- Integrated circuit temperature transducers with linear characteristics make temperature measurements easy.
- A dc motor can be controlled by an H-bridge circuit and a PWM waveform.
- Stepper motors allow you to precisely control the position and speed of the motor.

15.9 Bibliography and Further Reading

74HC138 1–of–8 Decoder/Demultiplexer, http://onsemi.com, 2007.

74HC139 Dual 2-Line to 4-Line Decoders/Demultiplexer, http://www.ti.com, 2003.

74HC151 8-Line To 1-Line Data Selectors/Multiplexers, http://www.ti.com, 2003.

74HC165 8-Bit Parallel-Load Shift Registers, http://www.ti.com, 2003.

74HC595, 8-Bit Shift Registers with 3-State Output Registers, http://www.ti.com, 2004.

Ganssle, J. G., *A Guide to Debouncing*, http://www.ganssle.com/debouncing.pdf, 2004.

LM19 2.4 V, 10 μA, TO-92 Temperature Sensor, http://www.national.com, 2007.

LM92 ±0.33°C Accurate, 12-Bit + Sign Temperature Sensor and Thermal Window Comparator with Two-Wire Interface, http://www.national.com, 2005.

LMD18200 3 A, 55 V H-Bridge, http://www.national.com, 2005.

MAX512 Low-Cost, Triple, 8-Bit Voltage-Output DACs with Serial Interface, http://www.maxim-ic.com, 1996.

MC14513B BCD-to-Seven Segment Latch/Decoder/Driver, http://onsemi.com, 2006.

OPA4344 Low Power, Single-Supply, Rail-To-Rail Operational Amplifiers MicroAmplifier Series, http://www.ti.com, 2000.

Kissell, T. E., *Industrial Electronics, Applications for Programmable Controllers, Instrumentation & Process Control, and Electrical Machines & Motor Controls*, 2nd. ed. Prentice Hall, Upper Saddle River, NJ, 2000.

15.10 Problems

Explore

15.1 Design an output circuit with eight LEDs connected to a port on your microcontroller. The LEDs are to be on when bits in a byte stored in location DATA1 are 1s. Show the hardware and software required. [c]

15.2 Design an input circuit to input the states of eight switches to a port on your microcontroller. [c]

Stimulate

15.3 A mythical microprocessor has two 8-bit output ports (P and Q) and two 8-bit input ports (R and S). Assume that a set of eight switches is connected to Port S and a set of eight LEDs is connected to Port P. Describe (a diagram would be nice) how you would use these resources (plus any others you would like: more switches, buffers, latches, etc.) to implement a scheme that would allow you to input data from the switches only after the user has completed entering new data, and then to display the 8-bit data on the LEDs. The hardware is to be as simple and cheap as possible. Describe how your system will input data from the switches and output to the LEDs. [c,k]

15.4 Now, assuming the hardware you have proposed for Problem 15.3, describe, from a high-level using pseudocode, how you would do the following.

a. Input data from the switches

b. Output data to the LEDs

15.5 An eight-digit LED display is multiplexed, with each digit being refreshed at 100 Hz, by an interrupt service routine. The ISR changes the display to the next digit and requires 8 μs to refresh each digit. [b]

a. If the interrupt service routine is started by an interrupt from the timer system, what interrupt rate would allow us to refresh each digit in the display at 100 Hz?

b. What percentage of the processor's time is spent refreshing the eight-digit display?

15.6 Why do bidirectional data ports on a microcontroller default to the input direction when the microcontroller is reset? [a]

Challenge

15.7 Use a 74HC138 1-of-8 decoder and a 74HC151 8-to-1 multiplexer to design a keyboard scanner that will scan an 8 x 8 keyboard matrix. Show your hardware, and give a software scanning algorithm to scan the keyboard and return a 6-bit keycode. [c]

15.8 The 16-key keypad scanner program shown in Example 15-6 does not debounce the switch. You may not have to implement a debouncing routine depending on your hardware (although you probably will). Propose a strategy and write a program, either in pseudocode, in the programming language of your microcontroller, or in C, that will test whether your keypad switches bounce when pressed.

15.9 Several of the program examples in this chapter call a function called delay_X_ms(int X). Write a function that will run on your microcontroller that will provide this variable delay, where X specifies the number of milliseconds to delay.

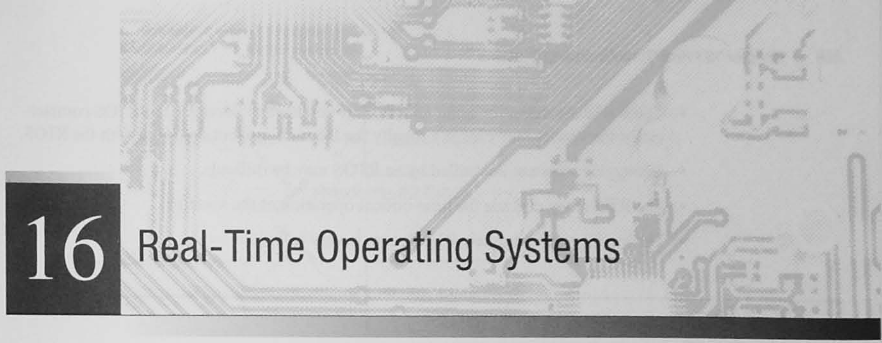
15.10 Find a data sheet for the LM19 temperature sensor shown in Figure 15-19, and use a spreadsheet and the transfer characteristic equations relating output voltage to temperature to develop an 8-bit lookup table that will allow an application program to display the temperature as a number ranging from –55 to +127.

Reflect on Learning

15.11 List five things you learned about interfacing to a microcomputer from this chapter.

15.12 What discovery or insight about input interfacing have you gained from this chapter?

15.13 What discovery or insight about output interfacing have you gained from this chapter?



16 Real-Time Operating Systems

Objectives

In this chapter we give a brief overview of real-time operating systems (RTOSs). We define terms and show how an RTOS works. Using and applying any RTOS for your microcontroller is far outside what we can accomplish in this chapter. You will need to refer to the documentation supplied with your RTOS.

16.1 Introduction

A real-time operating system is software that operates in the background, or behind, the application software, and manages the execution of the application software. Your application is partitioned into *tasks* that are usually independent of one another and thus can be developed and implemented independently.

Advantages of Using an RTOS

- In a real-time system the tasks are isolated from and independent of one another.
- Data structures, also, can be associated with a task and separated from other structures.
- Event-driven tasks are relatively easy to implement.
- Tasks that require periodic servicing are relatively easy to implement.

Disadvantages of Using an RTOS

- The RTOS consumes ROM, RAM, and CPU time.
- Each task requires its own stack, and therefore more RAM is needed in a real-time system.

- There is likely a cost for using an RTOS. If you have to purchase the RTOS commercially, the vendor may charge a royalty fee for each product delivered with the RTOS.
- Debugging a system controlled by an RTOS may be difficult.
- The RTOS may degrade the time-critical operation of the system.
- The learning curve for new RTOS users is very steep.

16.2 The Real-Time Operating System (RTOS)

RTOS Glossary

Clock tick: A real-time operating system maintains an internal clock that interrupts periodically.

Context: The context of a task is the current state of the CPU registers when the task is running.

Critical section: A section of code that cannot be interrupted by another process is said to be critical. A typical example is a section consisting of reading, changing, and writing to a data variable that another task may be using too. The other task should be prevented from accessing the data while it is being read, changed, and written to by the first.

Deadlock: In this condition, also called *deadly embrace,* processes wait for resources that may never become available. . In Figure 16-1 Task #1 is using Resource #1 but needs Resource #2 to complete. Task #2 is using Resource #2, so it blocks #1 from acquiring the second resource. In addition, Task #2 needs Resource #1 to complete, but Resource #1 is blocked by Task #1. To avoid deadlock, tasks should acquire all resources needed before running.

Deterministic: Often events must occur in real-time systems at precise intervals or within some time constraint. To achieve this, we would like the time taken by our tasks to be known or determinable. Elements contributing to nondeterministic behavior in functions include interrupt latency, unequal processing time in conditional loops, and the impact of higher priority tasks that might execute. An RTOS is deterministic if the execution time of the system calls is calculable.

Hard real time: A hard real-time system is one in which the task must be completed in tightly controlled periods; otherwise, a serious system error may occur. For example, a flight control system must recognize that an aircraft is approaching a stall before the onset of the stall. Hard real-time systems have deadlines for task completion that must be met. See soft real time.

Idle task: An idle task is one that runs when there are no other tasks to run.

Interrupt latency: This period, also called *dispatch latency*, is the time between the initiation of an interrupt request and entering of the interrupt service routine. In a real-time system, latency can refer to the time from between request of a task and the actual running of the task.

Kernel: The RTOS kernel is responsible for managing how the CPU executes tasks in a multitasking system. The kernel contains a scheduler or dispatcher that decides which task is to be executed next, as well as a way to preserve the CPU's registers, called the context, during switching between tasks.

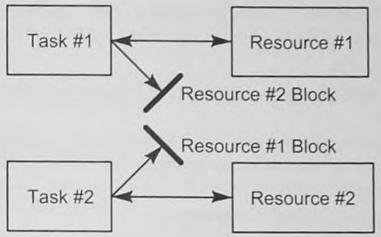


Figure 16-1 Deadlock.

Mailbox: A mailbox is a nonglobal data structure to allow tasks to exchange information.

Multitasking: A single CPU switching between several tasks is said to be multitasking.

Preemption: Preemption refers to how a kernel switches between tasks in a multitasking system. With a nonpreemptive switch, the currently running task gives up its control of the CPU voluntarily; that is, the task has been completed. With a preemptive switch, the kernel forces the currently running task to give up the CPU for a higher priority task.

Priorities: The kernel must run only one task even if more than one are ready. Thus a priority is assigned to each task. Of all the ready tasks, the highest priority task is picked. Priorities also allow the kernel to decide what tasks should interrupt other tasks.

Priority inversion: A priority inversion can occur if a higher priority task is blocked from running by a lower priority task. This can happen when two tasks share a resource.

Real-time system: It is difficult to state a definition for real time that all agree upon. In an embedded system, *real time* tends to designate an application or system in which the system performs its tasks on a time scale that the user considers to be short. For example, when pressing a key to effect an action, the user should get a response in a few tens of milliseconds; otherwise the user notices the delay and may become impatient and push the key again. In the context of this chapter, and when discussing real-time operating systems, a real-time system is one in which multiple tasks are executed to accomplish the objectives of the application. The tasks' executions are controlled by their priorities, by timing, or by events, and are overseen by the real-time operating system. The execution of the tasks is said to be *event driven.* Many system designers distinguish between *soft* and *hard real-time systems.*

Reentrant functions: A reentrant function is one that can be interrupted and then entered again without losing data from the first entry.

Resource: A resource is a process or I/O capability that is used by a task.

Safety critical: This term describes an RTOS certified by a certification body to be safe to use in system where human life is at stake.

Scalability: Scalability refers to how easily a small system can be expanded to a large system.

Semaphore: A semaphore is a mechanism that allows multiple tasks to share resources. When a task is using a resource, it sets a semaphore to signal other tasks that the resource is in use. The other resources must wait until the semaphore has cleared before they can acquire the resource.

Soft real time: In a soft real-time system, task execution is not on a strict time schedule; if a task is not completed precisely on schedule, no great disability occurs. Examples of soft real-time-systems include displays, printers, and other non-time-critical applications.

Starvation: Starvation can occur when a lower priority task is blocked from running by higher priority tasks that never relinquish the processor. Under these conditions, a task never receives a resource for which it is waiting.

Task scheduling: In a multitasking system, a variety of scheduling algorithms must be used in scheduling the tasks to run.

Task: A task is a program or function that uses the CPU and system resources. Tasks are written to be run independently of all other tasks, and multiple tasks are needed in an application. Tasks are also called threads.

Thread: A thread is a task or unit of program execution.

Time-Sharing Systems

Time-sharing systems are *multiuser* systems.

In the early to mid-1960s computers were large, expensive mainframes. Multiple users could use the computer simultaneously in *multiuser*, or *time-sharing*, systems. These systems allocated the CPU to each user in a time-multiplexed sequence. Figure 16-2 shows three users of a time-sharing system. Each user received the full resources of the CPU during its time slice and, unless there were many users, it seemed to each one that it had the computer full time.

We can see in Figure 16-2 that an operating system manages the switching between users at the beginning of each time slice. This is called task or context switching. Every time a user or task switches, the context is saved, to be restored the next time the task runs.

Figure 16-3 shows the architecture of a time-sharing or multiuser system. Each time the operating system receives an interrupt, it switches to the next user by saving the user's context

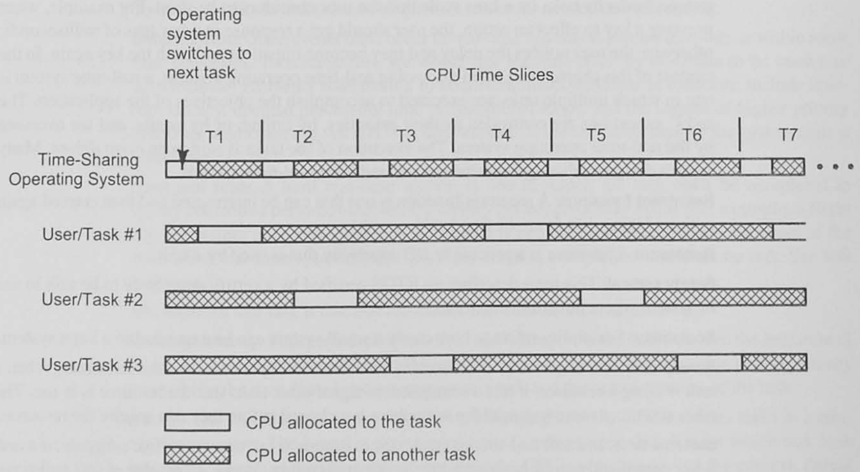

Operating system switches to next task
CPU Time Slices
T1
T2
T3
T4
T5
T6
T7
Time-Sharing Operating System
User/Task #1
User/Task #2
User/Task #3
CPU allocated to the task
CPU allocated to another task

Figure 16-2 Time-sharing system.

and, depending on how much main memory is installed, swaps the current program out to disk storage and brings in the next user's program. Users were often located remote from the mainframe computer, which they accessed through modems and dial-up or dedicated lines. Users who needed system I/O, say to print a program listing or program output, had to physically go to the computer center where the system was located.

Event-Driven, Real-Time System

The RTOS *kernel* manages the task switching.

The time-sharing system of Figure 16-2 is the grandparent of today's real-time systems. Instead of users of the system, we now have *tasks* that run when required by the application. Our multiuser system has become a *multitasking* system. Figure 16-4 shows a multitasking system that is *event driven*. Again, an operating system, the real-time operating system (RTOS), manages the task switching. Notice that the tasks do not necessarily run in sequence, nor are they allocated the same amount of CPU time.

The portion of the RTOS that manages the task switching is called the *kernel*. In Figure 16-4 the kernel prepares Task #1 to run (1, 2). At some point, either Task #1 finishes or an event occurs that requires Task #2 and the kernel prepares and runs it (3, 4). Following Task #2 the

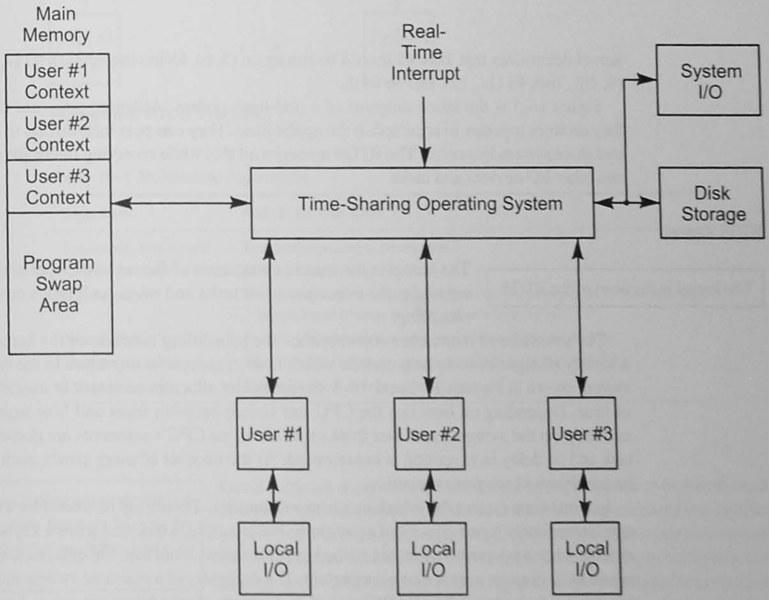


Figure 16-3 Time-sharing system architecture.

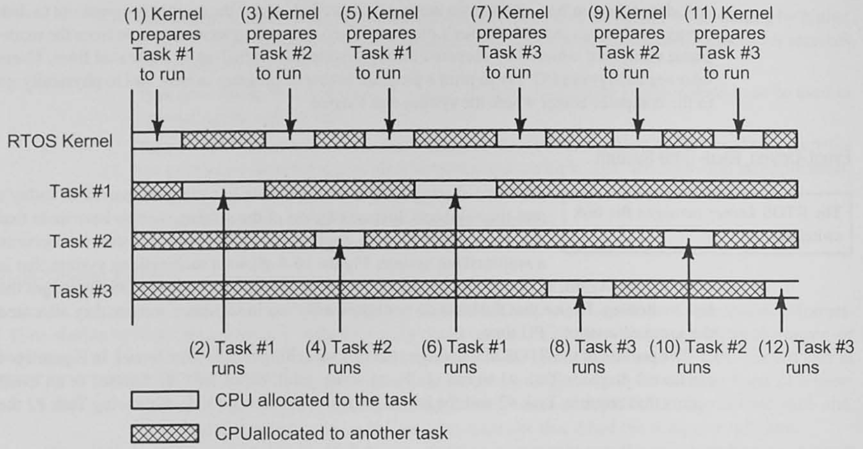

(1) Kernel prepares Task #1 to run
(3) Kernel prepares Task #2 to run
(5) Kernel prepares Task #1 to run
(7) Kernel prepares Task #3 to run
(9) Kernel prepares Task #2 to run
(11) Kernel prepares Task #3 to run
RTOS Kernel
Task #1
Task #2
Task #3
(2) Task #1 runs
(4) Task #2 runs
(6) Task #1 runs
(8) Task #3 runs
(10) Task #2 runs
(12) Task #3 runs
CPU allocated to the task
CPUallocated to another task

Figure 16-4 Event driven, multitasking system.

kernel determines that Task #1 should be run again (5, 6), following by Task #3 (7, 8), Task #2 (9, 10), Task #3 (11, 12), and so forth.

Figure 16-5 is the block diagram of a real-time system. Although tasks are independent, they do work together to accomplish the application. They can pass information to one another and share system resources. The RTOS manages all this while receiving interrupts from timers and other I/O devices and tasks.

RTOS Kernel

The kernel is the heart of the RTOS.

The kernel is the central component of the real-time operating system. It schedules the execution of all tasks and saves each task's context during switching.

The *scheduler* or *dispatcher* accomplishes the scheduling function of the kernel. There is a variety of algorithms to help choose which task to execute at any time. In the time-sharing system shown in Figures 16-2 and 16-3, the scheduler allocates each task or user an increment of time. Depending on how fast the CPU can switch between users and how many users are currently on the system, each user thinks that all of the CPU's resources are dedicated to that task and no delay in execution is experienced. As the number of users grows, each may experience delays in program execution.

In a real-time system the scheduling is more complex. There may be a need for a task to execute immediately based on a set of conditions. For example, a task that scans a keyboard should execute when a keypress is detected so the key information is not lost. Other tasks, such as updating an LCD display, may not be as important. The designers of a real-time system must consider priorities of the tasks, and the RTOS must allow a variety of scheduling processes. See Table 16-1 for a few of the many scheduling algorithms used in multitasking operating systems.

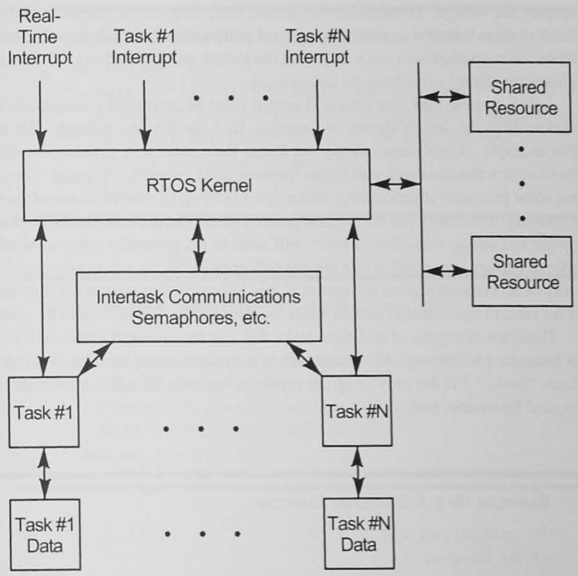


Figure 16-5 RTOS structure.

Table 16-1 Scheduling Algorithms

Algorithm	Task to Be Run Next
First-come, first-served	Task that requested to be run first
Shortest job first	Task with the shortest run time
Fixed-priority	The waiting task with the highest priority; priorities are set by the application programmer but may be changed as time goes by
Round robin	Each task is scheduled in sequence without regard to priority
Rate monotonic	Task with the highest execution rate given the highest priority
Deadline	Task with the closest completion time

Tasks

A *task* is an independent section of code that runs forever. It may be interrupted to allow other tasks to be run.

Good software design always calls for the program code to be partitioned, designed, and developed in modules that are as independent of one another as possible. For example, consider a program that allows a user to press one of two switches to select one of two input analog voltages to digitize and display on an LCD screen. Following the design principles we espoused in Chapter 3, we might develop at least one level of design as shown in Example 16-1. We could identify at least three functions, or tasks, the software must have to display the voltage. These could be `GetSwitch`, `DigitizeChannel`, and `UpDateDisplay`. Each of these functions could be developed independently and designed to transfer information from one to another. `GetSwitch` passes the switch number to `DigitizeChannel`, which then passes the digital value to `UpDateDisplay`.

Let us assume now that the LCD screen must be refreshed periodically so that it does not flicker. Will the design shown in Example 16-1 do this fast enough? We don't really know. For example, at this stage we do not know the conversion time of the A/D or whether the `GetSwitch` function will wait in the function until a switch is pressed. Thus, we see that there are some priorities of processing. Since `UpDateDisplay` needs to run often enough to prevent flickering, it probably has the highest priority of all. On the other hand, depending on how long `UpDateDisplay` runs, `GetSwitch` will need to act promptly when a switch is pressed to be able to capture the switch while we are still pressing it. `DigitizeChannel` is the lowest priority of all because it does not matter if it is interrupted to permit the display to be updated or if we need to read another switch while waiting for the conversion to be completed.

These are examples of real-time tasks that can be managed under an RTOS. As we can see in Example 16-2 through 16-4, each task is a separate, complete function that executes forever. Each "thinks" it is the only program running, but each transfers information to a data area to be used by another task.

Example 16-1 A/D Display Example

```
/* Initialize all I/O */
/* Do Forever */
/*   Get switches */
/*   If Switch 1 pressed
/*     Then */
/*       Digitize channel 1 */
/*       Update LCD
/*   ElseIf Switch 2 pressed
/*     Then */
/*       Digitize channel 2 */
/*       Update Display */
/*   Else do nothing */
/* End Do forever */
```

Example 16-2 Real-Time `GetSwitch` Task

```
/* Do Forever */
/*   Wait for a switch closure */
/*   Get which switch is closed */
/*   Set WhichSwitchClosed information */
/*
/* End Do forever */
```

Example 16-3 Real-Time `DigitizeChannel` Task

```
/* Do Forever */
/*   Wait until WhichSwitchClosed changes
/*   Start A/D conversion of the required channel */
/*   Wait for the conversion to be complete */
/*   Update CurrentADValue */
/* End Do forever */
```

Example 16-4 Real-Time `UpDateDisplay` Task

```
/* Do Forever */
/*   Put task to sleep to wait for 50 ms since the last
 *   display update */
/*   Convert CurrentADValue to DisplayValue */
/*   Send DisplayValue to LCD */
/* End Do forever */
```

Task Execution States

Each task in the real-time system may be in one of the following execution states:

1. **Sleeping:** A task may choose to delay its execution for a fixed time; during this time, it is said to be sleeping.
2. **Suspended:** A suspended task is not available for scheduling. The RTOS can suspend or resume a task.
3. **Blocked:** A task that is *blocked* is not running because it is waiting for some external event to occur. For example, it may be waiting for a time period to elapse or a shared resource to become available. When tasks are blocked, they usually have some timeout period specifying the maximum time they can remain blocked.
4. **Waiting or ready:** A task that is not blocked or suspended, and is not running because another task with higher priority is running, is in a *waiting* or *ready* state.
5. **Executing or running:** The task is using the CPU.

Task Switching

In the multitasking system shown in Figure 16-2, each user is allocated a unit of CPU time. A timer interrupt, as we described in Chapter 14, controls the task switching. This type of system is called a *round robin* system.

Another multitasking design strategy is called *event-driven* or *priority scheduling*. In this case, tasks are switched only when a task with higher priority needs service. Event-driven task switching can be *nonpreemptive* or *preemptive*.

In Figure 16-6a, a nonpreemptive kernel starts the low-priority task (1, 2) and then detects an interrupt that indicates that a high-priority task needs to run (3). The low-priority task completes its execution (4) and then relinquishes control of the CPU. The kernel does a context switch (5) and allows the high-priority task to run (6).

Figure 16-6b shows a preemptive kernel. As before, the low-priority task is running (1, 2) and an interrupt signals the need for the higher priority task (3). Now the kernel suspends the low-priority task (4), performs the context switch, and allows the high-priority task to run (5). When that task is completed, the kernel allows the low-priority task to resume (6, 7).

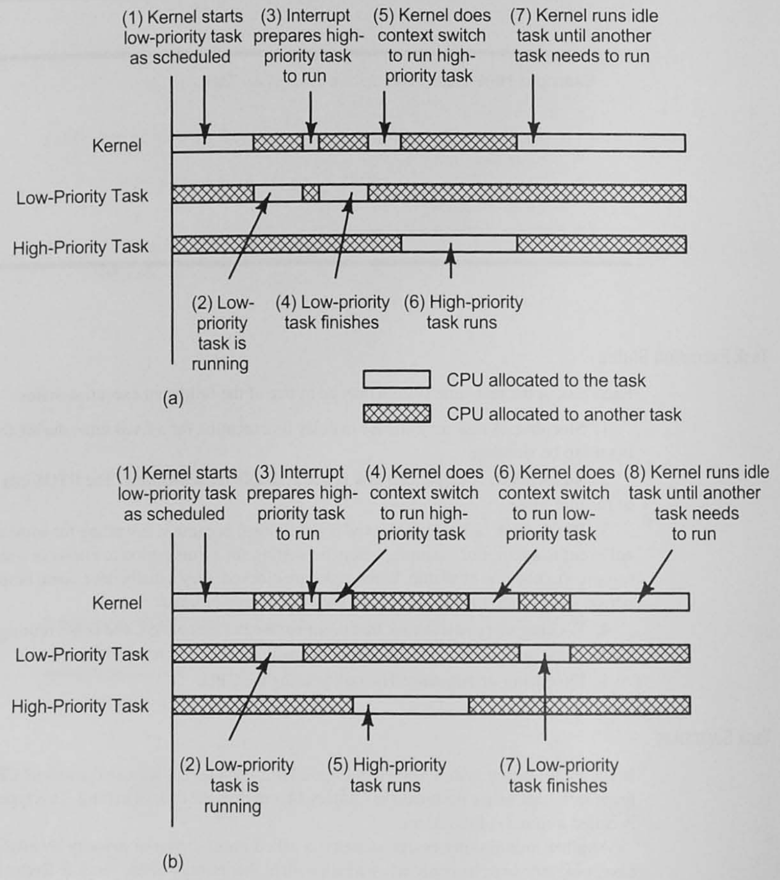

(1) Kernel starts low-priority task as scheduled
(3) Interrupt prepares high-priority task to run
(5) Kernel does context switch to run high-priority task
(7) Kernel runs idle task until another task needs to run
Kernel
Low-Priority Task
High-Priority Task
(2) Low-priority task is running
(4) Low-priority task finishes
(6) High-priority task runs
CPU allocated to the task
CPU allocated to another task
(a)
(1) Kernel starts low-priority task as scheduled
(3) Interrupt prepares high-priority task to run
(4) Kernel does context switch to run high-priority task
(6) Kernel does context switch to run low-priority task
(8) Kernel runs idle task until another task needs to run
Kernel
Low-Priority Task
High-Priority Task
(2) Low-priority task is running
(5) High-priority task runs
(7) Low-priority task finishes
(b)

Figure 16-6 Task switching. (a) Nonpreemptive. (b) Preemptive.

The Context Switch

The *context switch* saves the current task's CPU registers.

Each time a task is switched, either into or out of execution, the machine's context must be restored or saved. Figure 16-7 shows the situation facing the designer of the context switching portion of the RTOS kernel. For each task that executes, the CPU contains a program counter, a stack pointer, and registers with values specific for that task. When a task switch is made, these values must be saved and then restored when the task runs again. An interrupt often signals a task switch; as we know, the program counter, and in some processors all other registers, are saved on the stack. This arrangement saves the interrupted task's context, and the RTOS then switches to the next task by first restoring its context and then starting it up again.

Each task must have an area of RAM for a *task control block* or *process control block*, as shown in Figure 16-8. The task state is the execution state of the task (running, waiting, etc.) The priority establishes the task in the pecking order of all the tasks in the system. The stack pointer contains the current value of the task's stack pointer, and the timing parameter contains timing information needed to schedule the task.

Let us say Task #2 is running, with its program counter pointing to the next instruction to be executed, its registers holding with values that are being used, and the stack pointer pointing to the last (or next) used location on its stack. Now assume an interrupt that transfers

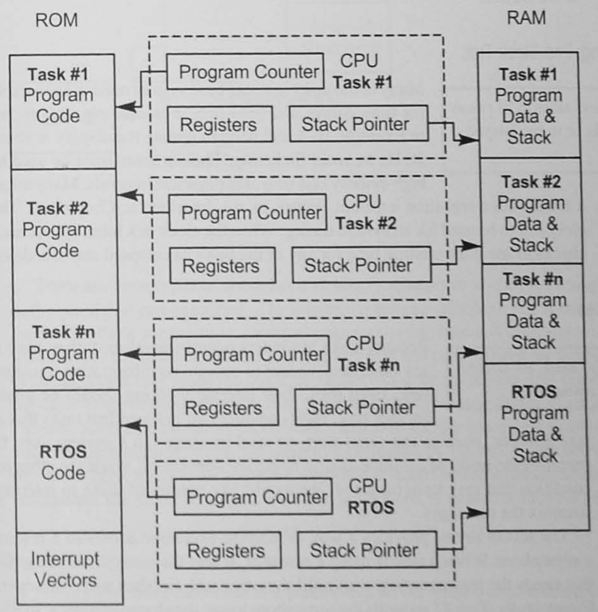


Figure 16-7 RTOS context switching.

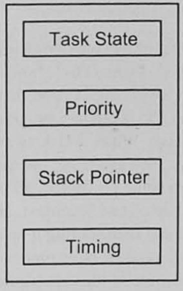


Figure 16-8 RTOS task control block.

control to the kernel to do a task and context switch if needed. The program counter and the registers are pushed onto Task #2's stack, saving its context. The kernel saves Task #2's stack pointer in the task control block and then determines the next task to be switched in and run. It loads the stack pointer register from that task's control block and executes a return from interrupt instruction, which reloads the new task's context from its stack, where it had been saved earlier.

Real-Time Timing: The Clock Tick

A *clock tick* allows tasks to be run at specific intervals or to be delayed a specific time.

Many of the tasks in a real-time system need to keep track of time intervals for time delays and time-outs. For example, the `UpDateDisplay` task in Example 16-4 will need to update the display at something over 20 to 30 Hz to avoid flickering. Thus, a timer must be used to ensure that this high-priority task runs at appropriate intervals. Many microcontrollers have a timer with a real-time interrupt feature, as we described in Chapter 10. This can generate a *clock tick* to be used for all system timing. When the clock tick interrupt occurs, the RTOS kernel checks to see if the waiting time for any of the tasks has elapsed and schedules them to run.

Sharing Resources

A semaphore is used to control access to shared resources.

Because tasks are independently developed, they have no knowledge of any other task, except to operate on information transferred from another task. Even then, their internal working should be totally transparent to another task. Now consider two independent tasks that must share a system resource, such as the serial port, to send messages to a system user. One task may be reporting the result of a routine analog-to-digital conversion, while the other reports some error condition that may have occurred. We would like neither of these to interrupt the other or to intermix the messages.

The RTOS kernel provides a way to manage exclusive access to a resource by providing a *semaphore*. When a task is using a resource, it sets the semaphore to exclude others. A task that needs the resource must wait until the other task finishes and releases the semaphore. In Figure 16-9a, Task #1 requests the semaphore to use the shared resource. In Figure 16-9b, it has been granted access, and the semaphore is reset to block Task #2's access to the resource.

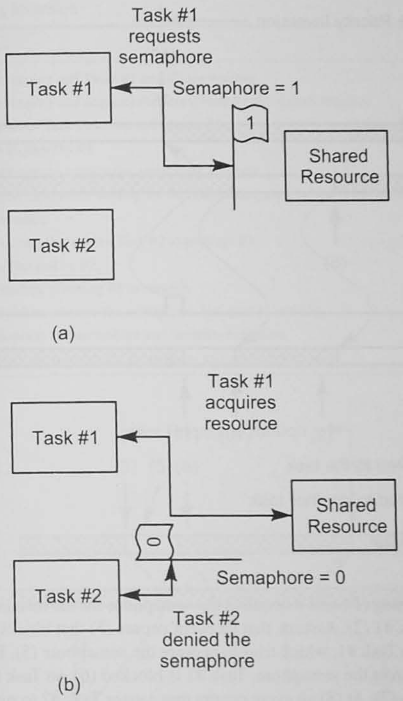


Figure 16-9 (a) Semaphore requested by Task #1. (b) Semaphore denied to Task #2.

There are three types of semaphore. A *binary* semaphore is used for a single resource, such as the serial port just described . The resource is either available or not. A *counting* semaphore is used when there are multiple, identical resources to be shared. A counting semaphore could be initialized to the number of blocks of memory that tasks can request for temporary storage. For example, a task may request a block of memory to store successive A/D values before calculating the average of one channel of data. Another task may need a similar block for another analog channel. Each time a memory block is allocated to a task, the counting semaphore is decremented and then incremented, when the memory is released by the task. If the semaphore has decremented to zero, a task must wait until a block is released before it can continue. The third type of semaphore, the *mutex* or *mutual exclusion* semaphore, is used to reduce *priority inversions*.

Priority Inversions

A priority inversion can occur in a preemptive RTOS when a shared resource is being used by a task and a higher priority task wishes to use the resource. In this scenario, (Figure 16-10), assume that Task #1 is the highest and Task #3 the lowest priority. Assume that Task #3 is

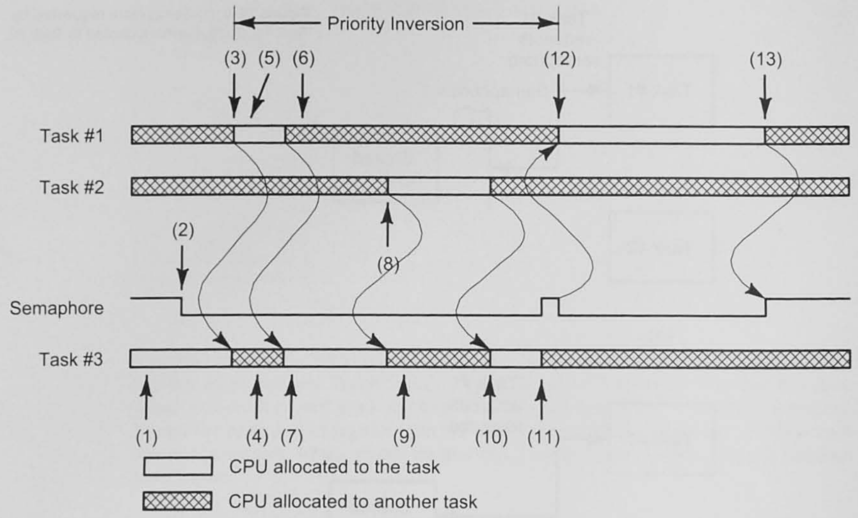


Figure 16-10 Priority inversion.

A *priority inversion* occurs when a lower priority task prevents a higher priority task from running.

running (1), and it acquires the semaphore for the resource it shares with Task #1 (2). Assume that an event occurs (3) that blocks Task #3 (4) and runs Task #1, which tries to acquire the semaphore (5). Because Task #3 controls the semaphore, Task #1 is blocked (6), so Task #3 resumes running (7). At (8) an event occurs that causes Task #2 to run. Because Task #2 is higher priority than Task #3, and does require the shared resource, Task #3 is blocked (9) until Task #2 finishes (10). Finally, Task #3 is allowed to complete when it releases the semaphore (11). Task #1 can now run to completion (12, 13). During the time Task #1 was blocked by the *lower* priority Task #3, Task #3's priority was effectively inverted to the same priority as Task #1. This is called *priority inversion*. Note that although Task #3's priority seems to be higher than Task #1's, it remains lower than Task #2's because Task #2 blocks Task #3 at (8). Table 16-2 summarizes the priority inversion case.

Figure 16-11 shows the same scenario, but now a mutual exclusion (mutex) semaphore with priority inheritance is used. When two tasks with different priorities need the same resource, the mutex semaphore is, again, a binary semaphore; when the higher priority task requests the resource, however, the lower priority task's priority is raised above that of higher priority task so that it continues to run to complete its use of the resource. In this way, Task #2's execution is delayed until both Task #3 and Task #1 complete. See Table 16-3.

Passing Information

As we saw in Examples 16-2 through 16-4, tasks are self-contained functions that run forever. Yet we must have some way to pass information between tasks. The `GetSwitch`

Table 16-2 Priority Inversion

	Task
(1)	Task #3 is running and Tasks #1 and #2 are waiting.
(2)	Task #3 is running and acquires the semaphore for the shared resource.
(3)	An event causes Task #1 to run and blocks #3.
(4)	Task #3 is blocked by #1.
(5)	Task #1 attempts to acquire the semaphore and fails.
(6)	Task #1 is blocked while waiting for the semaphore, even though it has higher priority than Task #3. This allows #3 to resume.
(7)	Task #3 resumes.
(8)	An event occurs that causes Task #2 to preempt #3.
(9)	Task #3 is blocked by #2.
(10)	Task #2 finishes, allowing #3 to resume.
(11)	Task #3 finishes, releases the semaphore, and goes to waiting.
(12)	Task #1 acquires the semaphore and resumes execution.
(13)	Task #1 finishes, releases the semaphore, and goes to waiting.

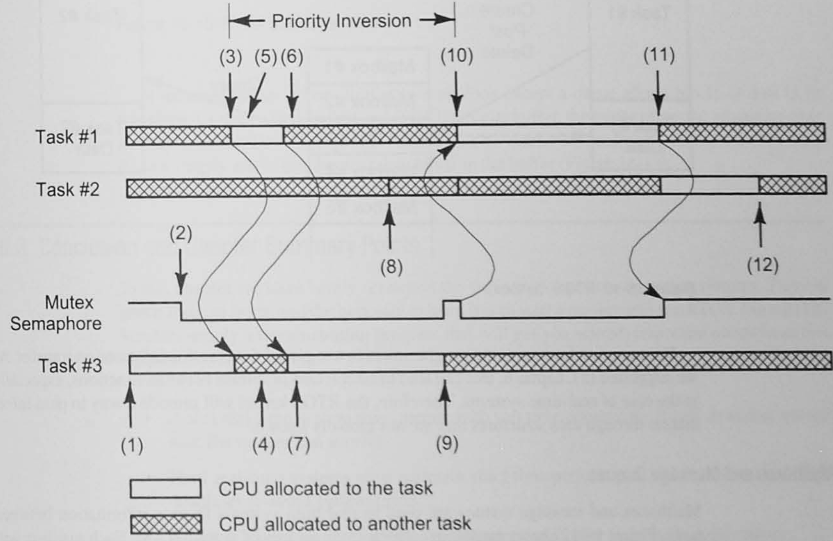


Figure 16-11 Priority inversion with priority inheritance.

Global data should not be used to pass information between tasks.

task must indicate that a switch closure has been made, and it mus pass the switch that was activated to the `DigitizeChannel` task s that it can convert the correct analog channel. `DigitizeChanne` then passes this digital value to the `UpdateDisplay` task.

Table 16-3 Priority Inversion with Priority Inheritance

	Task
(1)	Task #3 is running and Tasks #1 and #2 are waiting.
(2)	Task #3 is running and acquires the semaphore for the shared resource.
(3)	An event causes Task #1 to run and blocks #3.
(4)	Task #3 is blocked by #1.
(5)	Task #1 attempts to acquire the semaphore and fails.
(6)	Task #1 is blocked while waiting for the semaphore, even though it has higher priority than Task #3. This allows #3 to resume.
(7)	Task #3's priority is raised to be higher than #1, and #3 resumes execution.
(8)	An event occurs that requires Task #2, but because #3 now has the higher priority, #2 is blocked and #3 continues.
(9)	Task #3 finishes, releases the semaphore, and goes to waiting.
(10)	Task #1 acquires the semaphore and resumes execution. Task #2 remains blocked because #1 has higher priority.
(11)	Task #1 finishes, releases the semaphore, and goes to waiting, allowing #2 to execute.
(12)	Task #2 finishes and goes to waiting.

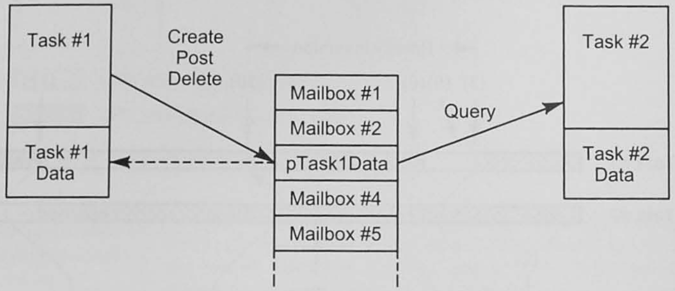


Figure 16-12 RTOS mailbox.

It is tempting for beginning programmers to use global data for this information transfer. As we suggested in Chapter 6, this can lead to interaction problems between functions, especially in the case of real-time systems. Therefore, the RTOS kernel will provide a way to pass information through data structures that are not globally known.

Mailboxes and Message Queues

Mailboxes and message queues are used in real-time systems to pass information between tasks. Figure 16-12 shows mailboxes, which are in an area of system RAM. Each mailbox will contain a pointer to data in a task. The task *creates* the mailbox, and the kernel allocates storage for the mailbox and returns a pointer to the mailbox to the task. The task can then *post* the pointer to its data (`pTask1Data`), which may be any data structure, in the mailbox with system function calls. The receiving task can query the mailbox to get the data pointer. The operating system allows handshaking between the two tasks and usually has a time-out feature so that if Task #2 is trying to query the mailbox for the data and Task #1 has not posted it yet, Task #2 will not be held up indefinitely.

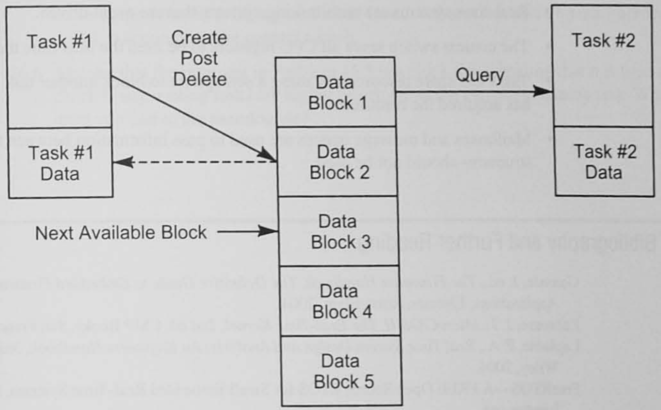


Figure 16-13 RTOS message queue.

A message queue is very similar to a mailbox except a queue allows blocks of data to be transferred. Often a first-in first-out (FIFO) circular buffer, the queue offers the advantage over a mailbox in that Task #2 might get behind Task #1 when #2 is reading #1's data. If it does, Task #1 can simply ask to use the next data block in the buffer (Figure 16-13).

16.3 Conclusion and Chapter Summary Points

In this chapter we have barely scratched the surface of real-time operating systems. There is much more to learn, and the best way to learn it is to start a project using an RTOS. Most RTOS vendors supply a demonstration program that will get you started. Important points from this chapter are the following.

- A soft real-time system may operate with soft time constraints. If task deadlines are not met, the system can survive.
- Hard real-time systems must maintain strict time performance and must meet task deadlines to avoid system faults.
- A task is a function that executes independently of other tasks in the application.
- An application consists of multiple tasks.
- A task may be sleeping, suspended, blocked, waiting, or running.
- The RTOS kernel provides task switching.
- Time-sharing systems share the CPU equally with all users and are the antecedents of today's real-time systems.

- Real-time systems are multitasking systems that are event driven.
- The context switch saves all CPU registers to be used the next time the task runs.
- Tasks can share resources by using a semaphore to block another task when one task has acquired the resource.
- Mailboxes and message queues are used to pass information between tasks; global data structures should not be used.

16.4 Bibliography and Further Reading

Ganssle, J. ed., *The Firmware Handbook, The Definitive Guide to Embedded Firmware Design and Applications*, Elsevier, Amsterdam, 2004.
Labrosse, J. J., *MicroC/OS-II, The Real-Time Kernel*, 2nd ed. CMP Books, San Francisco, 2002.
Laplante, P. A., *Real Time System Design and Analysis: An Engineers Handbook*, 3rd ed. Hoboken, NJ, Wiley, 2004.
FreeRTOS—A FREE Open Source RTOS for Small Embedded Real-Time Systems, http://www.freertos.org.

16.5 Problems

Explore

16.1 Use the documentation supplied with your RTOS, to make a table showing the system calls and the place in your documentation that gives details for the following functions:

a. task management functions
b. time management functions
c. semaphore management functions
d. mailbox management functions
e. message queue management functions

16.2 Does the operating system you are studying offer preemptive or nonpreemptive scheduling?

Stimulate

16.3 List at least five soft real-time system applications.

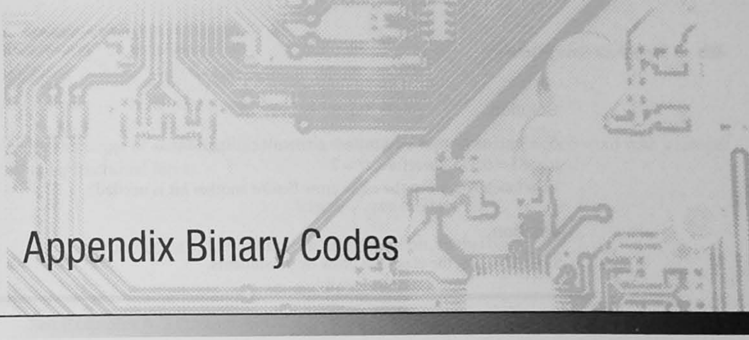
16.4 List at least five hard real-time system applications.

Challenge

16.5 Assume that an interrupt has just occurred, signaling a task switch. Your operating system maintains a pointer to the currently executing task's control block (Figure 16-8), `pCurrentTCB`, and uses a function `GetNextTCB()` to request a pointer to the next

task's control block. Write a pseudocode function that would run on your microcontroller to accomplish the context switch.

16.6 Assume that the interrupt in Problem 16.5 is a clock tick signaling that it is time to check if any waiting tasks are higher priority than the currently running task. What must you add to the pseudocode?



Appendix Binary Codes

A.1 Binary Codes Review

Coding is a two-part process consisting of encoding and decoding. Encoding means converting information into a form that can be used in the microcontroller, generally into a binary code. Decoding allows us to convert the coded information back to its original form. Whenever we choose a binary code, we consider the following.

The type of information to be encoded: Is the information numerical? Are there negative and positive numbers? Are there fractional or just integer numbers? If the information is not numerical, is there a standard code to be used? How much information is there? What is its range of values? To what resolution do we need to know and encode the information?

The number of bits needed to represent the information: The number of bits needed depends on the amount of information to be encoded and the resolution to which we need to know the information.

$$\text{Number of bits} \geq \log_2(\text{number of information elements})$$

or

$$\text{Number of bits} \geq \log_2 \frac{\text{full-scale value}}{\text{resolution value}}$$

When we know the number of bits required, we can calculate the number of code words available.

$$\text{Number of code words} = 2^{\text{number of bits}}$$

See Examples A-1 and A-2.

Example A-1

A binary code is needed to identify each of the 83 students in a class. How many bits are required

Solution

$N \geq \log_2$ (number of information elements) $\geq \log_2 (83)$
$\log_2 83 = 6.375$; therefore, $N = 7$
How much larger can the class grow before another bit is needed?

Solution

Since $2^7 = 128$, the class can grow by 45 students.

Example A-2

A binary code is needed to encode an analog voltage converted to a digital value by an analog-to-digital converter. The maximum voltage is 5.0 V and the resolution required is 0.01 V. How many bits are required?

Solution

$N \geq \log_2$ (full-scale value/resolution required) $= \log_2 (5.0/0.01) = 8.9$; therefore, $N = 9$.

Binary Codes for Numerical Information

We use several codes for numerical information. The five that are most important to microcontroller users are (1) unsigned binary, (2) signed/magnitude, (3) ones' complement, (4) two's complement, and (5) binary coded decimal.

Unsigned Binary Code

The unsigned binary code is a positive weighted code; each bit in the code word has a weight (or value) according to its position. Each digit is assigned a position starting at the binary point with zero, increasing to the left, and decreasing to the right. The weight of each position is the base raised to the power of the digit position. The left-most bit is the *most significant* bit (MSB) and the right-most bit the *least significant* (LSB). See Table A-1.

Unsigned binary codes are used for positive numerical information.

The unsigned binary code uses all positive weights and represents only positive information. The number of bits, and therefore the number of codes, determines how much information can be encoded. In a code word with p integer and q fractional bits, the number of codes is

Table A-1 Binary Word Bit Positions

Bits	…	b_3	b_2	b_1	b_0	.	b_{-1}	b_{-2}	b_{-3}	…
Bit position	…	3	2	1	0	.	–1	–2	–3	…
Bit weight	…	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	…
Weights	…	8	4	2	1	.	0.5	0.25	0.125	…
	MSB					Binary Point				LSB

$$\text{Number of codes} = 2^{p+q}$$

The range of numerical information that can be represented in a code word with p intege and q fractional bits is

$$\text{Range} = \text{zero to } 2^p - 2^{-q}$$

The resolution is the value of the least significant bit. In this case,

$$\text{Resolution} = 2^{-q}$$

See Examples A-3 through A-5.

The unsigned binary code can represent only positive information, but there are severa other codes used for negative information. The three used most commonly in the microcor troller world are the signed/magnitude, radix-1-complement (ones'-complement), and radix complement (two's-complement) codes.

Example A-3

An unsigned binary code has four integer bits ($p = 4$) and two fractional bits ($q = 2$). How mar codes are there? What is the range of numbers that can be encoded? What is the smallest nur ber that can be encoded?

Solution

The number of codes is $2^{p+q} = 2^6 = 64$.
The range of numbers is from 0 to $2^p - 2^{-q} = 16 - 0.25 = 15.75$.
The smallest number that can be encoded is the resolution $= 2^{-q} = 2^{-4} = 0.25$.

Example A-4

What are the weights of each of the bits p through w in an unsigned binary code word pqrst.uv

Solution

Code:	p	q	r	s	t	.	u	v	w
Weights:	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
	16	8	4	2	1	.	0.5	0.25	0.125

Example A-5

How many codes are there, what is the range of numbers that can be represented, and wh the resolution of an unsigned binary code word pqrst.uvw?

Solution

Number of codes = $2^8 = 256$
Range = 0 to $2^5 - 2^{-3}$ = 0 to 31.875
Resolution = $2^{-3} = 0.125$

Signed/Magnitude Binary Code

Signed/magnitude, ones'-complement, and two's-complement binary codes are used for positive and negative numbers.

The *signed/magnitude binary code* is similar to our decimal number system. The decimal code word for "plus twelve" is written +12 or just 12. "Minus twelve" is encoded –12. Two additional symbols, + and –, are added to the front of the digits used for the magnitude. These symbols double the number of code words to be able to represent both positive and negative numbers. Notice that there are two codes for zero, +0 and –0. By convention, we never use the code for minus zero.

In the binary system, an additional bit to encode the sign is added to the binary digits encoding the magnitude. A zero is used for positive numbers and a one for negative. Table A-2 shows the layout for a signed/magnitude binary code. Example A-6 shows a 7-bit binary code with one bit used as a sign and 6 bits to encode the magnitude.

The range of information that can be represented with p integer bits (including the sign bit) plus q fractional bits is

$$-(2^{p-1} - 2^{-q}) \text{ to } +(2^{p-1} - 2^{-q})$$

For Example A-6, the range is –15.75 to +15.75. Again, there is a code for plus and minus zero. See Example A-7.

Example A-6 Signed/Magnitude Binary Code Examples

0	1	0	1	1	.	1	1	=	+11.75
1	1	0	1	1	.	1	1	=	–11.75
Sign code		Magnitude code							

The left-most bit is the sign bit, and the magnitude is encoded with a 6-bit unsigned binary code.

Table A-2 Signed/Magnitude Code Bits

Bits	b_{p-1}	...	b_2	b_1	b_0	.	b_{-1}	b_{-2}	...	b_{-q}
Bit position	p-1	...	2	1	0	.	–1	–2	...	–q
Bit weight	Sign	...	2^2	2^1	2^0	.	2^{-1}	2^{-2}	...	2^{-q}
Weights	0 = +	...	4	2	1	.	0.5	0.25	...	
	1 = –									

Example A-7

How many codes are there, what is the range of numbers that can be represented, and what is the resolution of a signed/magnitude binary code word pqrst.uvw, where p is the sign bit?

Solution

Number of codes = $2^8 = 256$ (but two are used for zero)
Range = $-(2^4 - 2^{-3})$ to $(2^4 - 2^{-3})$ = –15.875 to 15.875
Resolution = $2^{-3} = 0.125$

Ones'-Complement Code

The definition of the *radix-1* or *ones'-complement* of a number X is

$$\text{Ones' complement} = 2^p - X - 2^{-q}$$

where p is the number of integer bits and q the number of fractional bits.

Example A-8 shows how to form the ones'-complement code for ±6.25. The left-most bi is an *indicator* (called the *sign bit*) for the sign of the number, with 0 representing positive an 1 negative. The range and the resolution of the ones'-complement code are the same as th signed/magnitude code and, again, there are two codes for zero. The ones'-complement cod is not a weighted code.

Example A-8

Find the ones'-complement code for –6.25, assuming a code of the form pqrst.uvw.

Solution

Find the unsigned binary code for +6.25 and add a sign bit in the most significant bit positio Then, to find the code for –6.25 complement all bits.

6.25			1	1	0	.	0	1	0	Unsigned binary code for 6.25
+6.25	=	0	1	1	0	.	0	1	0	Ones'-complement binary code for +6.25
–6.25	=	1	0	0	1	.	1	0	1	Ones'-complement binary code for –6.25

Two's-Complement Code

Two's-complement binary codes are used for negative numbers in microcontroller systems.

In the binary number system, the radix complement is the *twc complement* binary code. The definition of p-integer bit, twc complement of number X is

$$\text{Two's complement} = 2^p - X$$

Table A-3 Two's-Complement Code Bits

Bits, 0 or 1	b_{p-1}	…	b_2	b_1	b_0	.	b_{-1}	b_{-2}	…	b_{-q}
Bit position	p–1	…	2	1	0	.	–1	–2	…	–q
Bit weight	-2^{p-1}	…	2^2	2^1	2^0	.	2^{-1}	2^{-2}	…	2^{-q}
Weights	0 = +	…	4	2	1	.	0.5	0.25	…	
	1 = –									

This is a *negatively* weighted code because the most significant bit has a negative weight, as shown in Table A-3 and Examples A-9 and A-10.

There is only one code for zero in the two's-complement scheme. The code word used for minus zero in the signed/magnitude code is used for the most negative number. We can see this by looking at the range of the two's-complement binary code. For a number with p integer and q fractional bits, the range is

$$\text{Range} = -(2^{p-1})\ to + (2^{p-1} - 2^{-q})$$

The range of the binary number in Example A-9 is –8.00 to +7.875. The resolution is 2^{-q} = 0.125. See Examples A-9 through A-11.

Example A-9

Show the weights of a two's-complement binary number, 1 0 1 1 . 0 1 1.

Solution

1 0 1 1 . 0 1 1 = $1 \times (-2^3) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = -8 + 2 + 1 + 0.25 + 0.125 = -4.625$

Example A-10

What are the weights of each of the bits in a two's-complement binary code word pqrst.uvw?

Solution

Code:	p	q	r	s	t	.	u	v	w
Weights:	-2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
	–16	8	4	2	1	.	0.5	0.25	0.125

Example A-11

How many codes are there, what is the range of numbers that can be represented, and what is the resolution of a two's-complement binary code word pqrst.uvw?

Solution

Number of codes = 2^8 = 256
Range = $-(2^4)$ to $(2^4 - 2^{-3})$ = –16.000 to 15.875
Resolution = 2^{-3} = 0.125

The Sign of the Number

In the signed/magnitude, ones'-complement, and two's-complement codes, the most significant gives the sign of the number, although the sign bit for signed/magnitude code could be placed a where in the code word. In two's-complement codes the sign bit carries a negative weight. In sign magnitude and ones'-complement codes the sign bit does not carry a weight; it indicates the sig

Finding the Code for the Negative

In decimal, when we want the code for the negative of a number, we "take the negative of it' simply changing the sign. For a signed/magnitude binary code, the same is true. The sign b *complemented* to change a positive to a negative and vice versa. See Example A-6.

The ones'-complement code for a negative number is found by complementing each of bits in the code for the positive number. This process is called *ones'-complementing*, or *complementing* the bits. See Examples A-8 and A-12.

Finding the code for the negative in a two's-complement number system involves an e step. We find the code for the negative by *taking the two's complement* of the code for positive. This is analogous to "taking the negative" of a signed/magnitude code. The tv complement of any number is found as follows:

$$\text{Two's-complement code} = \text{Ones'-Complement code} + 2^{-q}$$

Taking the two's complement to find the negative is a three-step process:

1. Find the two's-complement code for the positive number.
2. Complement each of the bits (one's complement).
3. Add one to the least significant bit position.

This procedure is shown in Examples A-13 through A-15.

Example A-12 Ones'-Complement Binary Code Example

0 1 0 1 1 . 1 1 = +11.75

Complement each bit to find the code for –11.75.
1 0 1 0 0 . 0 0 = –11.75

Example A-13 Taking the Two's Complement

3.25	=	0	0	1	1	.	0	1	0	0	Two's complement code for +3.25
Ones' complement	=	1	1	0	0	.	1	0	1	1	
Add 2^{-4}		0	0	0	0	.	0	0	0	1	
–3.25	=	1	1	0	0	.	1	1	0	0	Two's complement code for –3.25

Example A-14

Find the two's-complement binary code for –6.25 assuming a code of the form pqrst.uvw.

Solution

+6.25	=	0	0	1	1	0	.	0	1	0	Two's-complement code for +6.25
Ones' complement	=	1	1	0	0	1	.	1	0	1	
Add 2^{-3}		0	0	0	0	0	.	0	0	1	
–6.25	=	1	1	0	0	1	.	1	1	0	Two's-complement code for –6.25

Example A-15

Take the two's complement of the code for –6.25 to find the code for +6.25.

Solution

–6.25	=	1	1	0	0	1	.	1	1	0
Ones' complement	=	0	0	1	1	0	.	0	0	1
Add 2^{-3}		0	0	0	0	0	.	0	0	1
+6.25	=	0	0	1	1	0	.	0	1	0

Binary Coded Decimal

A 4-bit, unsigned binary code is sometimes used to encode the ten decimal digits 0–9. This *natural binary coded decimal* is used so frequently that it is usually just called *binary coded decimal* or *BCD*. Table A-4, presented shortly, shows the natural BCD code. Because only 4 bits are used for each of the decimal digits, it is convenient to pack two BCD digits into one 8-bit byte. See Example A-16.

Example A-16 Packed BCD

Use an 8-bit packed BCD code to, give the code for the decimal numbers 23, 45, 99.

Solution

A packed BCD code has 4 bits for each decimal digit in one-half of each byte. The most s nificant nibble has the most significant digit's code.

23	=	0	0	1	0	0	0	1	1
45	=	0	1	0	0	0	1	0	1
99	=	1	0	0	1	1	0	0	1

Hexadecimal Codes

The hexadecimal, or base-16, number system is shorthand for strings of binary digits. Like BCD code, the 16 hexadecimal digits, 0–9, A–F, are encoded by using an unsigned binary c The hexadecimal digits and their binary codes will be shown in Table A-4. See Example A-

Example A-17 Hexadecimal Codes

Covert the binary number 1 0 1 1 0 1 0 1 to hexadecimal.

Solution

Start with the four least significant bits (0 1 0 1) = 5; the most significant four bits (1 0 1 1) The hexadecimal number is B5.

You Have to Know the Code

If given a binary number and asked what it means, you cannot answer unless you know code is being used. Table A-4 shows the different information that is decoded from a 4-bit word by using the different codes covered in this section.

Binary Arithmetic

Unsigned Binary Arithmetic

Adding and subtracting unsigned binary numbers are done just as we add and subtra magnitudes of decimal numbers. In each case we keep track of carries into or borrows the next-most-significant digit position. See Examples A-18 and A-19.

Table A-4 Four-Bit Binary Code Comparison

Code Word	Unsigned Binary	Ones' Compl.	Two's Compl.	Signed/Mag.	BCD	Hex
0000	0	0	0	0	0	0
0001	1	1	1	1	1	1
0010	2	2	2	2	2	2
0011	3	3	3	3	3	3
0100	4	4	4	4	4	4
0101	5	5	5	5	5	5
0110	6	6	6	6	6	6
0111	7	7	7	7	7	7
1000	8	–7	–8	–0	8	8
1001	9	–6	–7	–1	9	9
1010	10	–5	–6	–2	NA	A
1011	11	–4	–5	–3	NA	B
1100	12	–3	–4	–4	NA	C
1101	13	–2	–3	–5	NA	D
1110	14	–1	–2	–6	NA	E
1111	15	–0	–1	–7	NA	F
Range	0–15	–7 – +7	–8 – +7	–7 – +7	0–9	0–F

Example A-18 Unsigned Binary Addition

Add the 4-bit binary codes for 6 and 3.

Solution

```
Carries  1  1  0
  6   =  0  1  1  0
 +3   =  0  0  1  1
 ----------------------
  9   =  1  0  0  1
```

Example A-19 Unsigned Binary Subtraction

Subtract the 4-bit binary code 3 from 6.

Solution

```
                      6      0  1  1  0
                     –3  =   0  0  1  1
                             ----------
                             0  1  0  1
Partial difference       =   0  1  1
Borrows                      ----------
Difference            3  =   0  0  1  1
```

Overflow

An *overflow* is an error condition that occurs when the result of an arithmetic operation cannot be represented by the number of bits available.

An overflow occurs if the result of adding or subtracting (or multiplying or dividing) two numbers is a number outside the allowable range. In Example A-20 we try to use a 4-bit unsigned binary code to add 9 and 10. The expected result, 19, requires 5 bits. The carry bit out of the most significant bit indicates an overflow of the available bits. In microcontroller systems, this error in unsigned arithmetic is detected with a special flag called the carry flag.

Example A-20 Unsigned Overflow

Add the 4-bit binary codes for 9 and 10. The result must be 4 bits also.

Solution

```
Carry = overflow   1  0  0  0
   + 9       =        1  0  0  1
   +10       =        1  0  1  0
  -----               ----------
   + 3?               0  0  1  1
```

Two's-Complement Binary Arithmetic

The beauty of using the two's-complement code for signed numbers is that the hardware to do addition and subtraction is the same as the hardware for unsigned binary coded arithmetic. Further, one can easily subtract two numbers by adding the two's complement of the subtrahend to the minuend. This is shown in Example A-21 for a 6-bit, two's-complement code and in Example A-22 for an 8-bit code.

A microcontroller sets a bit called the *two's-complement overflow flag* when overflow occurs in two's-complement arithmetic.

A two's-complement overflow occurs when the result of an addition or subtraction is outside the allowable range of numbers for the number of bits available. When two's-complement numbers are added or subtracted, a carry out of the most significant bit position does not indicate an overflow as it does in unsigned binary arithmetic. See Example A-23.

There are various algorithms for detecting a two's-complement overflow. One of the easiest to understand is the following:

A two's-complement overflow occurs if adding or subtracting two numbers of the same sign yields a result with a different sign.

Two's-complement overflow cannot occur when one is adding or subtracting two numbers of opposite sign.

Example A-21 Subtraction by the Addition of the Two's Complement

Binary Subtraction									Subtraction by Adding the Two's Complement							
+5	=	0	0	0	1	0	1		+5	=	0	0	0	1	0	1
−3	=	0	0	0	0	1	1		+(−3)	=	1	1	1	1	0	1
+2	=	0	0	0	0	1	0		+2	=	0	0	0	0	1	0

Example A-22

Use 8-bit, two's-complement binary codes (5 integer and 3 fractional bits) to compute 8.75 − 10.5.

Solution

8.75	=	0 1 0 0 0 . 1 1 0	Two's-complement binary code for 8.75
10.50	=	0 1 0 1 0 . 1 0 0	Two's-complement binary code for 10.5
−10.50	=	1 0 1 0 1 . 1 0 0	Two's-complement binary code for −10.5

Therefore

8.75	=	0 1 0 0 0 . 1 1 0	Two's-complement binary code for 8.75
+ −10.50	=	1 0 1 0 1 . 1 0 0	Two's-complement binary code for −10.5
−1.75	=	1 1 1 1 0 . 0 1 0	

The result is negative. To find the magnitude of the result, take the two's complement.

0 0 0 0 1 . 1 1 0 = 1.75

Example A-23 Two's-Complement Overflow

Add the 4-bit, two's-complement numbers +6 and +3 and detect if an overflow occurs. The result is to be 4 bits.

Solution

Carry does not = overflow		0	1	1	0	
+6	=		0	1	1	0
+3	=		0	0	1	1
−7?	=		1	0	0	1

Two's-complement overflow has occurred because the sign of the result is different from the sign of the two numbers.

BCD Arithmetic

BCD numbers can be added, but the binary result must be adjusted with a *decimal adjust for addition* instruction to achieve the correct result.

As shown earlier, BCD codes encode the 10 decimal digits, each with 4 bits. Often two 4-bit BCD digits are packed into a single 8-bit byte for convenient storage. The microcontroller can add these bytes like any normal, binary addition; when the data are BCD numbers, however, a special adjustment must be added to correct the binary result to BCD. Consider adding 34_{10} to 29_{10}, where 34 and 29 are encoded in BCD.

+34	=	0 0 1 1 0 1 0 0
+29	=	0 0 1 0 1 0 0 1
5D?		0 1 0 1 1 1 0 1
+ Adjustment		0 0 0 0 0 1 1 0
+64		0 1 1 0 0 1 0 0

The result, $5D_{16}$, achieved by the microcontroller's *binary* arithmetic add instruction, is not cor- rect in either binary ($5D_{16} = 93_{10}$) or BCD (1101 is not a valid BCD digit). The microcontroller has a special instruction that, if executed immediately after adding the two BCD numbers automatically adds an adjustment to correct the result to BCD. This instruction is often callec *decimal adjust for addition*.

Binary Codes for Non numerical Information

Sometimes encoded (or decoded) information is not a number. A common example is th alphanumeric information sent from a keyboard to a computer or from a computer to a display Codes used for this application are called unweighted codes because, unlike the numerica codes, there is not a weight associated with a bit's position. To find out what a code means you must look it up in a table.

The ASCII Code

The *American Standard Code for Information Interchange* (*ASCII*) is used to encode alpha numeric information: for example, keys on a keyboard or letters displayed on a terminal. Th ASCII codes for alphanumeric information are shown in Table A-5. See Example A-24. Th two left-most columns (MS digit = 0 and 1) are control codes that have been defined for seria data communications. These are shown in Table A-6.

Example A-24 ASCII Code

Find the ASCII code for the letter H.

Solution

H is in the MS digit column '4' and the LS digit row '8'. Thus $H = 48_{16}$.

Table A-5 ASCII 7-Bit Codes for Alphanumeric Characters

	MS Digit							
LS Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	Xa	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

A.2 Problems

Explore

A-1 Use the ASCII code to encode your name. [a]

A-2 Decode the ASCII message 44 65 73 69 67 6e 69 6e 67 20 77 69 74 68 20 6d 69 63 72 6f 70 72 6f 63 65 73 73 6f 72 73 20 69 73 20 46 55 4e 21. [a]

A-3 Give the decimal value of the following binary code words assuming (i) unsigned binary, (ii) two's-complement binary, and (iii) signed/magnitude codes. [a]

a. 10101010
b. 01010101
c. 11001100
d. 00110011
e. 10000000
f. 01111111

A-4 Find the two's-complement binary code for the following decimal numbers: [a]

a. 26
b. –26
c. 32.125
d. –32.125

Table A-6 ASCII Control Codes

00	NUL	Null	Character with all zeros
01	SOH	Start of header	Used at the beginning of a sequence of characters that constitutes a machine-readable address of routing information; the header is terminated by the STX character
02	STX	Start of text	Character that precedes a sequence of characters to be treated as an entity; STX may be used to terminate a sequence of characters started by SOH
03	ETX	End of text	Character used to terminate an STX sequence of characters
04	EOT	End of transmission	Indicates the conclusion of a transmission
05	ENQ	Enquiry	Used as a request for a response from a remote station
06	ACK	Acknowledge	Character transmitted by a receiver as an affirmative response
07	BEL	Bell	Character used to control an alarm or attention device
08	BS	Back space	Controls the movement of the printing mechanism back one space
09	HT	Horizontal tab	Controls the movement of the printing mechanism to the next predefined tab position
0A	LF	Line feed	Moves the printing mechanism to the next line; in some systems this may be interpreted as a "New Line" (NL), where the print mechanism moves to the beginning of the next line
0B	VT	Vertical tab	Controls the movement of the printing mechanism to the next predefined printing line position
0C	FF	Form feed	Moves the printing mechanism to the start of the next page
0D	CR	Carriage return	Moves the printing mechanism to the start of the line
0E	SO	Shift out	Indicates that the code combinations following are outside the character set of the standard ASCII table until a Shift In character is received
0F	SI	Shift in	Indicates that the code characters following are to be interpreted according to the standard ASCII table
10	DLE	Data Link Escape	Changes the meaning of a limited number of following characters. DLE is usually terminated by a Shift In character
11	DC1	Device controls	Characters used to control ancillary devices associated with data processing
12	DC2		
13	DC3		
14	DC4		
15	NAK	Negative acknowledge	Transmitted by a receiver as a negative response to the sender
16	SYN	Synchronous idle	Character used by a synchronous transmission system in the absence of any other characters to maintain synchronism between the transmitter and receiver
17	ETB	End of transmission block	Used to indicate the end of a block of data
18	CAN	Cancel	Indicates that the data with which it is sent is in error or is to be disregarded
19	EM	End of medium	Sent with data to represent the physical end of the medium
1A	SUB	Substitute	Character that may be substituted for a character that is invalid or in error
1B	ESC	Escape	Control character intended to provide code extension; it is usually a prefix affecting the interpretation of a limited number of contiguously following characters
1C	FS	File separator	Information separators that may be used within data
1D	GS	Group separator	
1E	RS	Record separator	
1F	US	Unit separator	

A-5 Find the decimal equivalent of the following two's-complement numbers: [a]

a. 0101101.1
b. 1010010.1
c. 1000
d. 1010.1101

A-6 A 6-bit, two's-complement binary code is to be used for integer numbers. What is the range of information? What is the resolution? How many codes are there? [a]

A-7 How many bits are required to encode the decimal number 238 by means of a BCD code? How many by means of an unsigned binary code? How many by means of a two's-complement binary code?

A-8 Find the binary code words for the following hexadecimal numbers: [a]

a. BEEF
b. FEED
c. C0FFEE
d. F00D

A-9 Find the hexadecimal code words for the following binary code words: [a]

a. 01011010
b. 11110101
c. 110101
d. 101

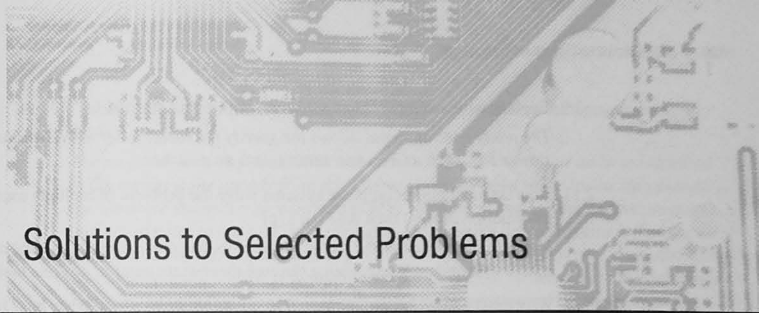
Stimulate

A-10 Prove that two's-complement overflow cannot occur when two numbers of different signs are added. [a]

Challenge

A-11 For 4-bit-number, two's-complement addition, choose four examples to demonstrate the following: [a, b]

a. Addition with no carry out and no two's-complement overflow.
b. Addition with no carry out and two's-complement overflow.
c. Addition with carry out and no two's-complement overflow.
d. Addition with carry out and two's-complement overflow.



Solutions to Selected Problems

Solutions to Chapter 2 Problems

2.1 What is the difference between an assembler and a compiler? [a, c]

An assembler converts an assembly language program consisting of operations and their operands into the machine language (1s and 0s) for the microcontroller.

A compiler converts a high-level language, such as C, first into the assembly language needed for the line of C code, and then into the machine language for the microcontroller.

2.3 What is a microcontroller memory map? [a]

A graphic representation showing what kind of memory is located in what address space.

2.5 What is the purpose of the program counter? [a]

The program counter points to the location in memory from which the CPU is fetching instructions.

2.7 Give short answers to the following: [a, g]

a. What is a data bus?
A parallel, bidirectional, binary information pathway with multiple sources and destinations.

c. How is an information source, such as a set of switches, interfaced to a data bus?
With three-state gates whose enable is controlled by an Address_OK signal and a READ control signal.

e. Give the sequence of events that occur when a CPU does an input (or read) cycle.
CPU puts address on the address bus
Address decoder generates ADR_OK
CPU asserts READ control signal
Input device puts data on the data bus
CPU reads the data
CPU deasserts READ control signal

2.9 Explain why a computer has ready or wait control signals. [c]

The ready control signal allows the speedy microcontroller to be synchronized with slower I/O, such as a human setting data on switches.

2.11 How do most microcomputer systems solve the problem of multiple sources of information present on a data bus? [g]

Multiple sources can exist as long as addressing and address decoding are used to enable one and only one source through three-state gates at any one time.

2.13 Why must a latch be used to interface an output device to the data bus? [a, c]

The data bus is active all the time, with data flowing to and from memory and I/O devices. To be able to output specific data to an output device at a specific time, a latch must be used that is clocked by the write control signal and the correct address.

2.15 A microcontroller memory map shows 16 Kbyte of Flash EEPROM (ROM) in memory space 0xC000-0xFFFF and 1 Kbyte of RAM in memory space 0x1000–0x13FF. [c, k]

a. Give a range of addresses (in hex) suitable for locating code:

0xC000–0xFFFF

b. Give a range of addresses (in hex) suitable for allocating variable data storage.

0x10000–0x13FF

2.17 Design an instruction decoder as shown in Figure 2-13 using AND, OR, and inverter gates to decode the 3-bit opcodes and produce a control signal asserted by each of the operations given in Table 2-5.

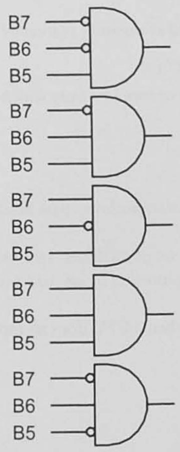


Figure S–2-17.

2.19 Describe the instruction execution cycle of a move-immediate instruction shown in Table 2-11. [c, e]

The program counter points to the instruction to be executed; this address is applied to the memory; the opcode from this address is transferred to the instruction decoder; the sequence controller recognizes the MVI instruction and increments the program counter to point to the next byte (the data); the memory data addresses is transferred to the memory address register and applied to the memory; the sequence controller generates timed control signals required to transfer data from the memory to the destination register; the program counter is incremented to the next instruction to be executed.

2.21 Draw a timing diagram relative to the system CPU clock shown in Figure P_2–21, which includes the address and data buses, R/W_L, and the read control signal (READ_L = active low) and shows a read cycle. [a]

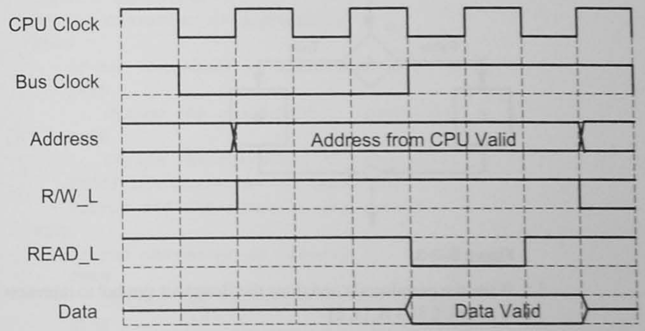


Figure S–2-21.

2.23 A CPU generates a bus clock and R/W_L signal during a read cycle as shown in Figure 2–18. Give a logic equation or show a logic diagram expressing the logic required for the READ_L control signal.

READ_L = ADDRESS_OK and R/W_L

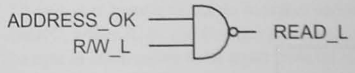


Figure S–2-23.

Solutions to Chapter 3 Problems

3.1 List at least five principles of top-down design. [a, c]

Understand the problem completely; design in levels; ensure correctness at each level; postpone details; successively refine your design; design without using a programming language.

3.3 Write the pseudocode and draw the flowchart symbol to represent the decision IF A is TRUE THEN B ELSE C. [a, c]

```
IF A
THEN
    Begin B
    . . .
    End B
ELSE
    Begin C
    . . .
    End C
ENDIF A
```

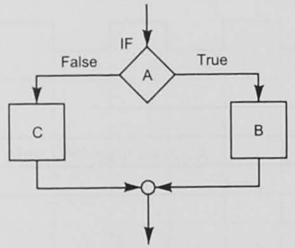


Figure S–3-3.

3.5 Write the pseudocode and draw the flowchart symbol to represent the repetition WHILE A is TRUE DO B. [a, c]

```
WHILE A
DO
    Begin B
    . . .
    End B
ENDO
ENDWHILE A DO
```

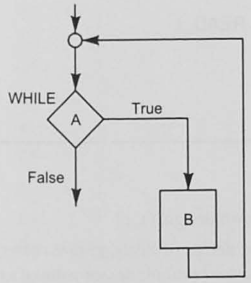


Figure S–3-5.

3.7 Use structured flowcharts or pseudocode to write a design that will implement the following problem description: [c]

Prompt for and input a character from a user at the keyboard.
If the character is alphabetic and is uppercase, change it to lowercase and output it to the screen.
If the character is alphabetic and is lowercase, change it to uppercase and output it to the screen.
If the character is numeric, output it with no change.
If it is any other character, beep the bell.
Repeat this process until an ESC character is typed by the user.

```
DO
    Output a prompt
    Input a character
    IF the character is alphabetic
    THEN
        IF the character is uppercase
        THEN
            Change the character to lowercase
        ELSE
            Change the character to uppercase
        ENDIF the character is uppercase
        Output the character to the screen
    ELSE
        IF the character is numeric
        THEN
            Output the character to the screen
        ELSE
            Output a bell to the screen
        ENDIF it is numeric
    ENDIF the character is alphabetic
ENDO
WHILE The character is not an ESC
```

3.9 Use structured pseudocode to give a design that will accomplish the following: [c]

A user is to input a character to select one of three processes. Valid characters are A, B, and C, where A, B, and C select processes A, B, or C, respectively. Process A requires a byte of information to be input from an A/D converter, which it then converts to a integer decimal number in the range of 0 to 5 and displays it on the screen. Processes B and C are not defined at this stage. Prompts and error messages are to be displayed. You do not have to give details of the decimal conversion required in Process A.

```
Prompt user for character A, B, or C
Get the character
IF the character is A
THEN
    Get input from the A/D
    Convert it to an integer decimal number in the range of 0 to 5
    Display the value on the screen
```

```
ELSE
    IF the character is B
    THEN
        Do process B
    ELSE
        IF the character is C
        THEN
            DO process C
        ELSE
            Print error message that user did not enter A, B, or C
        ENDIF the character is C
    ENDIF the character is B
ENDIF the character is A
```

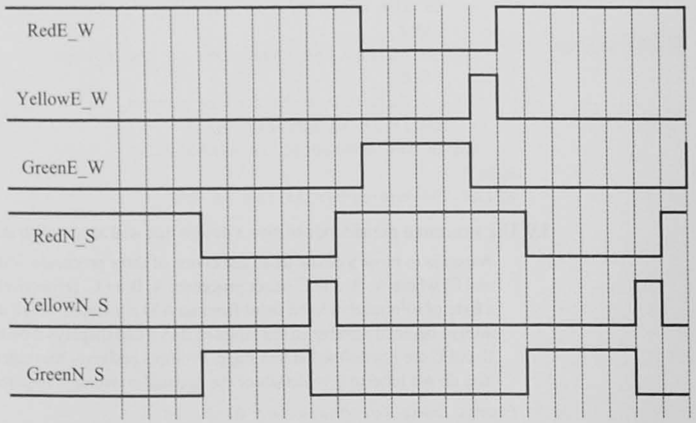
3.11 Design a traffic light controller: [c]

Imagine an intersection with north/south and east/west streets. There are to be six traffic light signals:

RedE_W, YellowE_W, GreenE_W

RedN_S, YellowN_S, GreenN_S

Assume the time elements in the table below are 10 seconds and that a timer delay is available as a function or subroutine. Give the pseudocode structured design for the light controller.



```
Start with GreenN_S coming on:
Initial conditions: Turn RedE_W on and RedN_S on. All
  others off.
DO
   Turn RedN_S off and GreenN_S and RedE_W on
   Wait 40 seconds
   Turn YellowN_S and RedE_W on and GreenN_S off
   Wait 10 seconds
   Turn RedN_S and RedE_W on and YellowN_S off
   Wait 10 seconds
   Turn RedE_W off and GreenE_W and RedN_S on
   Wait 40 seconds
   Turn YellowE_W and RedN_S on and GreenE_W off
   Wait 10 seconds
   Turn RedE_W and RedN_S on and YellowE_W off
   Wait 10 seconds
FOREVER
```

Solutions to Chapter 4 Problems

4.1 List the CPU registers available in the microcontroller you are studying.

4.3 In Example 4-34, what is the decimal result of the two's-complement binary addition? [a]

70_{10}.

4.5 What is the meaning of sign bit = 1 when unsigned binary coded numbers are added? [a]

Merely that the most significant bit is 1.

4.7 What is the meaning of carry bit = 1 when unsigned binary coded numbers are added? [a]

An overflow has occurred.

4.9 What is the meaning of zero bit = 1 when unsigned binary coded numbers are added? [a]

The result is zero.

4.11 What is the meaning of two's-complement overflow bit = 1 when unsigned binary coded numbers are added? [a]

It has no meaning in unsigned binary code arithmetic.

4.15 Do the following 8-bit binary additions and for each case give the expected result in the carry, zero, sign and overflow flags.

a. 1010 0011 +0011 1011 1101 1110 C=0, Z=0, S=1, OV=0	b. 1111 1111 +0000 0001 0000 0000 C=1, Z=1, S=0, OV=0	c. 0111 0001 +0100 0000 1011 0001 C=0, Z=0, S=1, OV=1
d. 1010 0010 +1000 0000 0010 0010 C=1, Z=0, S=0, OV=1	e. 0111 1111 +1000 0000 1111 1111 C=0, Z=0, S=1, OV=0	f. 1010 1010 +0101 0101 1111 1111 C=0, Z=0, S=1. OV=0

4.17 For problem 4.15, assume that the binary numbers are in two's-complement binary code. Show the equivalent decimal arithmetic operations and indicate if overflow has occurred. [a]

a. −93 + 59 = −34 No overflow	b. −1 + 1 = 0 No Overflow	c. 113 + 64 = −79 Overflow
d. −94 + −128 = 34 Overflow	e. 127 + −128 = −1 No overflow	f. −86 + 85 = −1 No overflow

4.19 For the multibyte addition shown in Example 4-2, state what kind of instruction you would expect the microcontroller to have to be able to do this. [e]

An add-with-carry instruction is needed.

Solutions to Chapter 5 Problems

5.1 List the addressing modes available in the CPU you are studying.

5.3 A microcontroller is to be used in an embedded system with the following memory map:

0x0000 0x1FFF	ROM
0x2000 0x7FFF	None
0x8000 0xFEFF	RAM
0xFF00 0xFFFF	ROM

a. In what memory addresses must code and constant data be located? [c]

0x000–0x1FFF or 0xFF00–0xFFFF

b. In what memory addresses must variable data and storage be located [c]

0x8000–0xFEFF

5.5 Name at least five ways to address an operand. [a]

register; indexed; memory indirect; register indirect; direct; immediate; relative

5.7 What are the names of the addressing modes that form the effective address from a constant and the contents of a register? [a]

Based; indexed

5.9 To increase the memory address space in a computer system, one must (a) increase the number of data lines, (b) increase the number of read and write control bits going to the memory, (c) increase the number of address lines. [a]

(c) increase the number of address lines

5.11 A register indirect address instruction (a) has the address of the operand in the instruction, (b) has the address of the operand in a register, (c) uses the program counter to calculate the offset address of the operand. [a]

(b) has the address of the operand in a register

5.13 Assume you are designing a CPU that is to have a 20-bit address bus with each memory location containing 16 bits. A base page is defined that has 1024 locations. Assume that memory indirect addressing using base page addresses is the ONLY kind of memory addressing this CPU has. How many bits in the instruction must be allocated for a memory reference instruction? [c]

10 bits to address the 1024 locations in the base page

Solutions to Chapter 6 Problems

6.11 In Example 6-1 a constant defined by an equate is used to initialize a register with a constant value in line 50 and a constant stored in ROM memory is used to initialize a register in line 58. Comment on these two assembly language programming techniques. Which is better? [a]

Because you are initializing a register with a constant known at the time you are writing the program (assembly time), there is no need to allocate and use a memory location to do this. Use the equate for these constants.

6.21 Use the principles of structured programming to write structured pseudocode (do not write assembly language code) for the following problem statement: [c]

The program is the prompt for and will accept a two-digit hexadecimal number from a user typing characters on the keyboard. These are to be converted to an 8-bit binary number and displayed on the LEDs. After a one-second delay, the complement of the byte is to be displayed on the LEDs for one second. After this delay, the LEDs are to be turned off and the process repeated starting at the prompt. The program is to continue until the user types two zeros ("00").

Your design should follow the principles of top-down design, and you may postpone consideration of such details as how to convert the two input characters to binary, and the details of the prompt and how it is to be printed.

```
; Initialize stack pointer
; Initialize I/O
; Enable LEDs
; DO
;   Prompt for an input
;   Get two characters from the keyboard
;   Convert to binary
;   Output to LEDs
;   Delay 1 second
;   Complement the data
;   Output to LEDs
;   Delay 1 second
;   Blank LEDs
; ENDDO
; WHILE User has not entered "00"
```

Solutions to Chapter 7 Problems

7.9 Write a C function to search a null-terminated string of characters for a specific substring and to return the address of the start of the substring. The input to the subroutine is to be the starting address of the string to be searched, the starting address of the substring to be searched for, and the number of characters in the substring. If the substring is found, return the address of the first character in the search string; otherwise return an address of 0x0000. [c]

```
/*****************************************************************
 * unsigned char* str_search( unsigned char* input_str, unsigned
  char* search_str, unsigned int search_len );
*****************************************************************/
char* str_search( char* input_str, char* search_str, int
  search_len ){
char *temp_start_str, *temp1, *temp2;
int i;
/*****************************************************************/
 temp_start_str = 0;
 temp1 = input_str;
 temp2 = search_str;
 while (*temp1 != 0){
   if ( *temp1 == *temp2 ) {
     /* Found the start of a match */
     temp_start_str = temp1;  /* Save the start of the string */
     i = 1;
     ++temp1;
     ++temp2;
     while ((*temp1 == *temp2) && (i <= search_len)){
       ++i;
       ++temp1;
       ++temp2;
     }
     if ( i == search_len ){
       /* The search string has been found */
       return( temp_start_str );
     } else {
       temp2 = search_str;
     }

   }
   ++temp1;
 }

 return( search_str );

 }
```

7.11 Write a program for your microcontroller in C and then in assembly (or vice versa) to find the largest of thirty-two 8-bit unsigned numbers in 32 successive memory locations. Place the answer in the next available location. [c]

```
/*****************************************************************
 * Problem 7.11
 * Find the largest unsigned char byte in a 32-byte buffer
 * Put the result in the next memory location following the
 * buffer.
 ****************************************************************/
 unsigned char DATA[32], result;
 ****************************************************************/
void main(void) {
int i;
char temp1;
/****************************************************************/
  /* Initialize the data buffer with some test data */
  temp1 = 0xF0;
  for (i = 0; i < 32; ++i ){
    DATA[i] = temp1++;
  }
  result = 0;  /* initialize the result = smallest */
  for (i = 0; i < 32; ++i){
    if (DATA[i] > result) result = DATA[i];
  }
  for(;;) {} /* wait forever */
}
```

7.13 There are 4 bytes of data in variable data array DATA[0]–DATA[1]. Write a program in C to count the number of 1s in these 4 bytes. Place the result in NUM_ONES. [c]

```
/*****************************************************************
 * Problem 7.13
 * Count the number of 1's in a 4 byte, (2 word) buffer.
 ****************************************************************/
/* Test data */
int DATA[2] = {
  0xffff, 0x0000
};
int NUM_ONES;
/****************************************************************/
void main(void) {
int i, j;
unsigned int temp1, temp2;
/****************************************************************/
  NUM_ONES= 0;  /* initialize the result = 0 */
  for (i = 0; i < 2; ++i){
    temp1 = DATA[i];
    temp2 = 0x01;    /* Scan pattern 0x01, 0x02, 0x04 etc. */
    /* Scan each of the words ANDing with a shifting bit pattern */
    for (j = 0; j < 16; ++j){
      if ((temp1 & temp2) > 0) {
        /* Have found a 1 bit */
        ++NUM_ONES;
      }
      /* Shift the scanner */
      temp2 = temp2*2;
    }
  }
  for(;;) {} /* wait forever */
}
```

Solutions to Chapter 8 Problems

8.1 Explain why 0x20 is exclusive-ORed with the input character in Example 8-4 to change the case. [a, g]

The ASCII codes for the alphabetic characters differ only in bit 5. For example, the ASCII code for 'A' is 0x41 and for 'a' is 0x61. Toggling bit 5, which is accomplished by exclusive-ORing the code with 0x20, changes the case.

8.3 Assume that main( ) in Example 8-4 is a test jig to test the `change_case` function. Comment on the thoroughness of the testing. What would you do to make the testing better and more rigorous? [b]

The test string tests only a few characters. A better test would be to initialize an array with all 128 ASCII codes and check to see that only the alphabetic characters are changed by the `change_case` *function.*

Solutions to Chapter 9 Problems

9.3 Which type of I/O addressing, separate I/O or memory mapped, requires a control signal called "I/O request" to access I/O devices? [a]

Separate I/O.

9.5 Describe the advantages of the three-state gate over the open-collector gate when the application entails multiple sources on a data bus. [a, g]

The three-state gate has an active pull-up and thus does not require an external resistor pull-up. The three-state enable also allows the input of the gate to be any logic level. With the open-collector gate, the logic must ensure the output is high when the gate is to be disabled. In addition, it requires an external pull-up resistor.

9.7 Why are three-state gates used in an input interface? [a]

When a three-state gate output is disabled, it presents a high impedance to the bus. This allows multiple sources to be connected to the bus.

9.9 In a parallel output operation, how is the synchronization of the data transfer between CPU and a data latch consisting of eight D-type flip-flops accomplished? [a]

The CPU places the address of the data latch onto the address bus followed by the data to be output. It then asserts the WR control signal and the IORQ signal, if needed.

9.11 Discuss the consequences of a CPU designer's decision to implement memory-mapped I/O instead of separate I/O. What does it mean to the CPU designer, and what does it mean to you, the system designer using the CPU? [e]

To the CPU designer, memory-mapped I/O results in a simpler design for the sequence controller. Separate I/O instructions do not need to be provided because any memory reference instruction can access I/O. For the system designer using the CPU, memory-mapped I/O means that the address decoders needed for each I/O device must decode the full address bus instead of a smaller address, as generally used in separate I/O. Also, less memory will be available for program and data use.

9.13 A 74HC138 decoder has the following address bits assigned to its inputs:

ADR	74HC138 Pin
A7	A2
A6	A1
A5	A0
A4	E1
A3	E2_L
A2	E3_L

A1 and A0 are don't cares.

Assume an 8-bit address and make a table similar to Table 9-4 showing what address each output responds to. [b]

Table S-9-13 Reduced Address Decoding

Address Bits								
A7	A6	A5	A4	A3	A2	A1 A0		
Decoder Inputs								
A2	A1	A0	E3	E2_L	E1_L	Not Used by Decoder	Valid Hex Addresses	Decoder Output
0	0	0	1	0	0	00 to 11	0x10–0x13	O0_L
0	0	1	1	0	0	00 to 11	0x30–0x33	O1_L
0	1	0	1	0	0	00 to 11	0x50–0x53	O2_L
0	1	1	1	0	0	00 to 11	0x70–0x73	O3_L
1	0	0	1	0	0	00 to 11	0x90–0x93	O4_L
1	0	1	1	0	0	00 to 11	0xB0–0xB3	O5_L
1	1	0	1	0	0	00 to 11	0xD0–0xD3	O6_L
1	1	1	1	0	0	00 to 11	0xF0–0xF3	O7_L

9.15 Example 9-5 shows a segment of code that waits until a status bit asserted by the output device is ready to accept new data before writing the data to the port. What can go wrong with this arrangement, and what might you do to make it better? [b, c]

If the status bit is never asserted, the program hangs up. While this may be OK, depending on the application, it might be better to have a timer that will let you continue with the rest of the program. Another alternative is to use an interrupt.

Solutions to Chapter 10 Problems

10.1 List five possible applications for interrupts. [a]

Fire sensors; counting; analog conversion completed; timing; generating waveforms

10.3 Why, in most processors with interrupts, are further interrupts disabled when the processor reaches the interrupt service routine? [a, k]

To give the programmer control over further interrupts and to keep one interrupt from interrupting another.

10.5 Name two methods by which a CPU can determine which of several devices has generated an interrupt. [a]

Polled and vectored interrupts.

10.7 What are polled interrupts? [a]

Interrupts in which the CPU must do a sequence of I/O reads (in a program, this is called polling) to determine which device generated the interrupt request.

10.9 Define "interrupt latency." [a]

The time between the interrupt request and the start of execution of the interrupt service routine.

10.11 Give at least two components of interrupt latency. [a]

Time taken to finish the current instruction; time taken to save the system registers; time taken to fetch the vector or poll the interrupting devices.

10.17 For a processor with 10 interrupting devices, which type of architecture, polled or vectored, provides the faster transfer of control to the interrupt service routine for a specific interrupt? [a]

Vectored interrupts.

10.19 What must be done to solve the problem of two devices generating simultaneous interrupts in a system with polled interrupts? [a]

Poll the devices in the order of priority.

10.21 A real-time interrupt generator as shown in Figure 10–12 is driven by an 8 MHz clock. A programmable divider is followed by a 10-bit counter to generate overflow interrupts. [b, c]

a. How should the programmable divider be set to generate interrupts at approximately once every millisecond?
For the 10-bit counter to overflow every 1 ms, its clock must be (1024 counts/10^{-3} s) = 1.024×10^6 Hz. Therefore the programmable divider should be set at (8×10^6 Hz)/(1.024×10^6 Hz) = 7.812. The nearest integer divider is 8.

10.23 Design the hardware for an input interrupting device in a polled interrupt system. Assume an 8-bit switch register for data, a one-bit status register for an "I did it" bit, and a push-button switch to generate a wired-OR IRQ_L signal. The status register and switch register are each to occupy an address in the 8-bit I/O address space. Assume separate I/O with control signals READ_L and WRITE_L. [c]

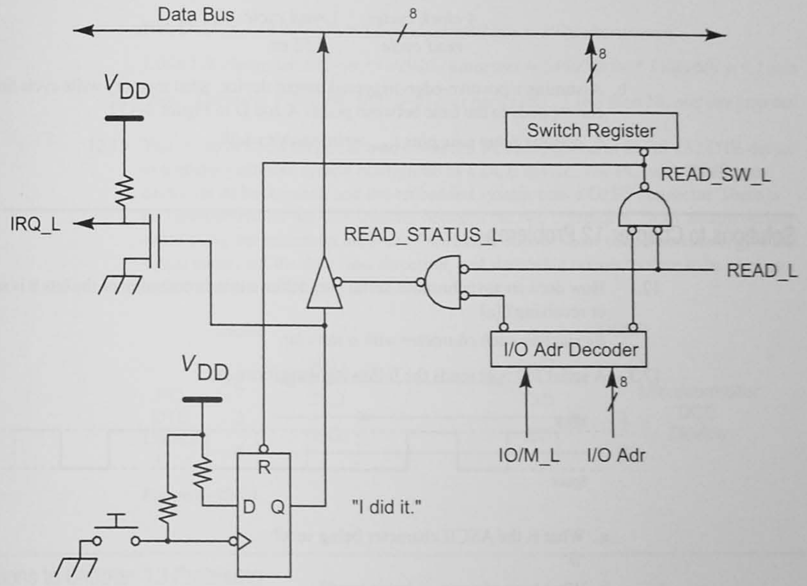


Figure S-10–23.

Solutions to Chapter 11 Problems

11.3 A CPU reads from the data bus 150 ns after it has supplied the address to the address bus. Which memory access time specification would be best to use for RAM memory in this system? Justify your decision in terms of cost and system reliability. [c]

(a) 10 ns; (b) 110 ns; (c) 150 ns; (d) 200 ns;

(b) Although it appears that 150 ns would do the job, there is no margin of error to account for uncontrolled variables such as operation over temperature extremes and variable propagation delays due to board layout. Therefore, 110 ns memory will be the best choice. This allows for propagation delays to occur and still have the memory contents available on the data bus when the CPU reads it.

11.5 Compare the memory write cycles shown in Figures 11–10a and 2–17. For the memory timing shown in Table 11-2, answer the following.

a. What is the maximum CPU clock frequency that would be allowed?

The address from the CPU is valid for four CPU clock cycles = 55 ns minimum. The maximum CPU clock frequency is

$$\frac{4\ clock\ cycles}{read\ cycle} \times \frac{1\ read\ cycle}{55\ ns} = 72.72\ MHz$$

b. Assuming a positive-edge-triggered output device, what memory write cycle time corresponds to the time between points A and D in Figure 2–17?

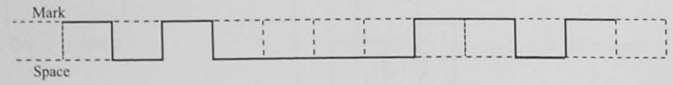
t_{A_S} address setup time plus t_{MWE} write enable width

Solutions to Chapter 12 Problems

12.1 How does an asynchronous serial port achieve synchronization of the bits it is sending or receiving? [a]

By starting each character with a start bit.

12.3 A serial I/O port sends the following waveform: [a]



a. What is the ASCII character being sent?

'a'

b. What type of parity is being used?

odd parity

12.5 To initiate a serial data transfer, a UART first [a]

b. *sends the start bit.*

12.7 How many bits per second (baud) is a serial port sending when the character rate is 120 characters per second? Assume ASCII characters with even parity. [a]

1200 bits/s

12.11 You are to define a serial cable to connect two PCs configured as RS 232 DTE devices. Each PC has a DE9P connector on its back panel. The software used in each PC for file transfer uses hardware (RTS/CTS) flow control. Draw an appropriate cable using the minimum number of wires. Be sure to show each connection, give the signal name, tell the data flow direction, and state what connectors are to be used on each end of the cable. [c]

See Figure 12-4. Use DE9S connectors on each end of the cable.

12.15 Why is the RS-232 voltage specification for mark and space logic levels used for serial communications voltage levels instead of TTL? [a]

To have increased noise margin.

12.17 An SCI is transmitting data at 19.2 kbaud. The format is seven data bits, even parity, one stop bit. How long does it take to send a document that is one megabyte long? [a]

*19,200 bits/s * 1 character/10 bits = 1920 characters/s*

*1,048,576 characters/Mbyte * 1s/1920 characters = 546s/Mbyte * 1 min/60s = 9.1 min*

(Note: There are 7 data bits (bits 6–0) plus parity (bit 7), one start bit, and one stop bit.

12.19 You are to define a serial cable to connect a PCs configured as an RS-232 DTE device to a microcontroller system configured as a DCE device. The PC has a DE9P connector on its back panel, and the embedded system uses a DE9S connector. There is no flow control for the data transfer between the two computers. Draw an appropriate cable using the minimum number of wires. Be sure to show each connection, give the signal name, tell the data flow direction, and state what connectors are to be used on each end of the cable. [c]

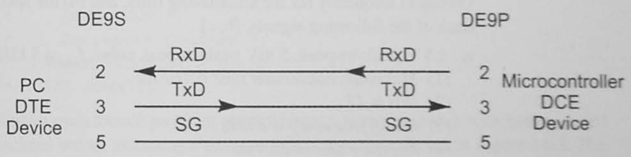


Figure 5–12-19.

Solutions to Chapter 13 Problems

13.1 Briefly explain the following terms: aperture time, conversion time, aliasing, Nyquist frequency. [a]

Aperture time: The time the A/D or sample-and-hold is "looking" at analog input signal.

Conversion time: The time between START_CONVERT and END_OF_CONVERT.

Aliasing: An effect caused by frequencies in the digitized signal that are greater than twice the sampling frequency.

Nyquist frequency: The maximum frequency that can be in the input signal without aliasing.

13.3 How does a successive approximation A/D converter work? [a]

It tests each bit in succession starting at the most significant bit. This bit drives a digital-to-analog converter, which generates a signal for one side of a comparator. The comparator compares the input analog signal with the output of the D/A. If the input is higher, the bit remains set; otherwise it is reset. Each successive bit is tested in this way.

13.5 How does a flash converter work? [a]

A flash converter consists of 2^N–1 comparators. The output code is produce in a flash in this way.

13.7 The A/D converter conversion time is 100 μs. What is the maximum frequency that can be digitized without the occurrence of aliasing? [b, c]

$$f_{max} = \frac{1}{2 \times 100\,\mu s} = 5\ kHz$$

13.9 An A/D converter is to digitize a 10 V, full-scale signal to a resolution of 1 part in 1024. [b, c]

a. How many bits are required?

10 bits

c. What is the accuracy when a 1 V signal is digitized?

$$Accuracy = \frac{9.77mV}{1V} \times 100\% = 0.977\%$$

13.11 For an A/D converter, specify (1) maximum conversion time, (2) number of bits, (3) cutoff frequency for the antialiasing filter, and (4) the aperture time to digitize each of the following signals. [b, c]

a. ±5 V, peak-to-peak, 5 mV peak-to-peak noise, f_{max} = 3 kHz

(1) Maximum conversion time = 167 μs
(2) Bits = 11
(3) Cutoff frequency = 3 kHz
(4) Aperture time = 25.9 ns

c. ±1 V, peak-to-peak, 5 mV peak-to-peak noise, f_{max} = 1 kHz

(1) Maximum conversion time = 500 μs
(2) Bits = 9
(3) Cutoff frequency = 1 kHz
(4) Aperture time = 0.31 μs

Solutions to Chapter 14 Problems

14.3 Describe how to use the timer/counter circuit in Figure 14-2 to generate a 10 kHz square wave. Assume a counter clock frequency of 8 MHz. [c]

An interrupt is needed every 50 μs. One option would be to set the programmable divider to divide by 8 and then to initialize the counter with –50. The counter would then overflow, generating the interrupt and automatically reloading the initialization value.

14.5 Assuming a 2 MHz counter clock frequency, what is the period of the external signal in Figure 14-5b? [b]

The interval between two successive rising edges is 0x2476–0x0EFF = 0x1577 = 5495_{10}.

Period = 5495/2 × 0^6 = 2.748 ms

14.7 A timing circuit is needed that can generate a time delay longer than 2^{16} counter clock cycles of the timer/counter comparison circuit shown in Figure 14-3. Use pseudocode to describe a strategy to do this. [c]

Divide the required delay ΔT into N smaller intervals, and calculate the number of counter clock cycles for this interval, say M, so that M is <2^{16}.

$$M = \frac{\Delta T}{N} \times clock\ frequency$$

```
/* Initialize the timer for output compare operation */
/*  */
/* Delay Delta T */
/*   Read 16-bit counter */
/*   Add M */
/*   Store in 16-bit comparator */
/*   Reset COF */
/*   Initialize Counter = N */
/*   DO */
/*     Wait for COF to be set (delay M cycles) */
/*     Add M to 16-bit comparator */
/*     Reset COF */
/*     Decrement Counter */
/*   WHILE (Counter > 0) */
/*   END DOWHILE */
```

14.9 Write a pseudocode program to implement a real-time clock with binary coded decimal output assuming a timer output compare as shown in Figure 14-3. The clock is to display hh:mm in 24-hour format. [c]

```
/* Initialize timer for output compare operation */
/* Initialize all clock variables to 0 */
/*   h_tens = 0 */
/*   h_ones = 0 */
/*   m_tens = 0 */
/*   m_ones = 0 */
/* DO */
/*   Delay one minute */
/*   Increment m_ones */
/*   IF m_ones = 10 THEN */
/*     /* m_ones = 0 */
/*     /* Increment m_tens */
/*   ENDIF m_ones = 10 */
/*   IF m_tens = 6 THEN */
/*     /* m_tens = 0 */
/*     /* Increment h_ones */
/*   ENDIF m_tens = 6 */
/*   IF h_ones = 10 THEN */
/*     /* h_ones = 0 */
/*     /* Increment h_tens */
/*   ENDIF h_ones = 10 */
/*   IF h_ones = 4 and h_tens = 2 have reached 24:00 THEN */
/*     /* h_tens = 0 */
/*     /* h_ones = 0 */
/*   ENDIF h_ones = 4 and h_tens = 2 */
/*   Update the clock display */
/* FOREVER */
```

14.11 Assume you are to use the timer in Figure 14-2 generate a 50 Hz square wave. Assume that the bus clock is 8 MHz and the programmable divider factors are 1, 2, 4, 8, 16, or 32. Write a pseudocode design that will allow you to output required square wave. [c]

With an 8 MHz clock and the programmable divider set to divide by 32, the counter clock is 250 kHz. To generate a delay of 10 ms (half the period of 50 Hz), a 2500 clock cycle delay is needed.

```
/* Initialize the timer for automatic reload operation
 * and set the programmable divider to divide by 32 */
/* Initialize the 8/16-bit register with -2500 */
/* Reset the TOF */
/* DO */
/*   Wait for the TOF */
/*   Toggle the output bit */
/*   Reset the TOF */
/* FOREVER */
```

Solutions to Chapter 15 Problems

15.1 Design an output circuit with eight LEDs connected to a port on your microcontroller. The LEDs are to be on when bits in a byte stored in location DATA1 are 1s. Show the hardware and software required. [c]

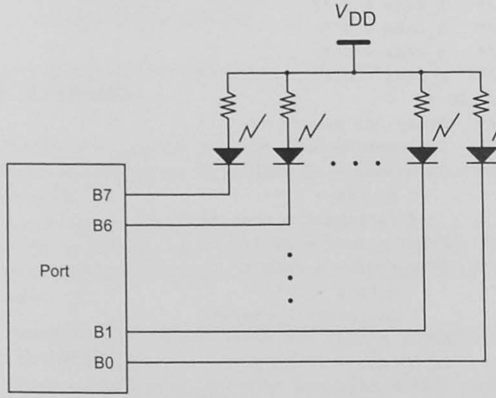


Figure S-15-1.

```
/* Get DATA1 */
/* Complement it */
/* Output it to the port */
```

15.3 A mythical microprocessor has two 8-bit output ports (P and Q) and two 8-bit input ports (R and S). Assume that a set of eight switches is connected to Port S and a set of eight LEDs is connected to Port P. Describe (a diagram would be nice) how you would use these resources (plus any others you would like: more switches, buffers, latches, etc.) to implement a scheme that would allow you to input data from the switches only after the user has completed entering new data, and then to display the 8-bit data on the LEDs. The hardware is to be as simple and cheap as possible. Describe how your system will input data from the switches and output to the LEDs. [c, k]

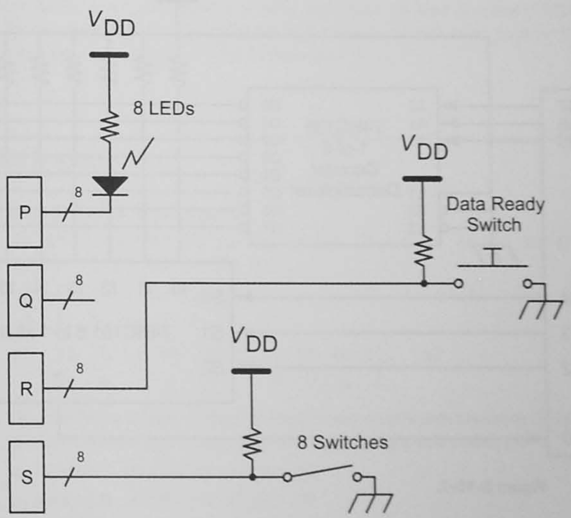


Figure S-15-3.

Eight LEDs are connected to Port P, which must be initialized as an output port. An output of zero to a bit on Port P illuminates the LED. Eight switches are connected to Port S. A data ready switch is connected to a bit (e.g., bit 7) on Port R.

15.5 An eight-digit LED display is multiplexed, with each digit being refreshed at 100 Hz by an interrupt service routine. The ISR changes the display to the next digit and requires 8 μs to refresh each dgit. [b]

a. If the interrupt service routine is started by an interrupt from the timer system, what interrupt rate would allow us to refresh each digit in the display at 100 Hz?

With eight digits being refreshed at 100 Hz, the interrupt frequency must be 800 Hz, or 0.00125 seconds per interrupt.

b. *What percentage of the processor's time is spent refreshing the eight-digit display?*

The total time between interrupts is 1,250 μs and the time to refresh is 8 μs. Therefore, 8/1250 = 0.64%

15.7 Use a 74HC138 1-of-8 decoder and a 74HC151 8-to-1 multiplexer to design a keyboard scanner that will scan an 8 × 8 keyboard matrix. Show your hardware, and give a software scanning algorithm to scan the keyboard and return a 6-bit keycode. [c]

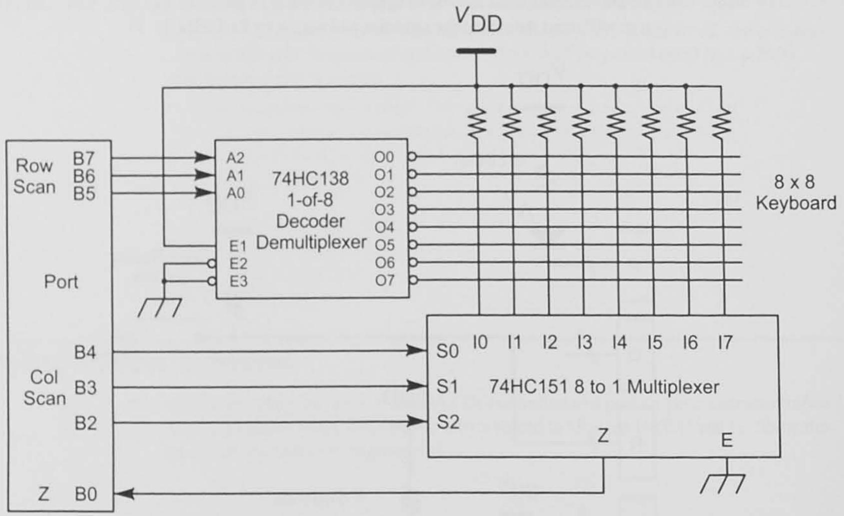


Figure S-15-7.

A key switch at each of the 64 locations connects a row line, driven by the 74HC138 decoder, to a column line, which is input to the 74HC151 multiplexer. With no keys pressed, each of the columns is high. When a key is pressed, and when that row line is asserted low, the associated column is low.

```
/* FOR Row_Scan = 0 to 7 */
/*   FOR Col_Scan = 0 to 7 */
/*     IF Z = 0 THEN Break */
/*   ENDFOR Col_Scan = 0 to 7 */
/*  IF Z = 0 THEN Break */
/* ENDFOR Row_Scan = 0 to 7 */
/* Look Up the keycode based on the present Row_Scan and
 * Col_Scan */
```

Solutions to Chapter 16 Problems

16.3 List at least five soft real-time system applications.

LCD display driver, CD music player, keyboard driver, airline reservation system, printer driver, cell phone, etc.

16.5 Assume that an interrupt has just occurred signaling a task switch. Your operating system maintains a pointer to the currently executing task's control block (Figure 16–8), `pCurrentTCB`, and uses a function `GetNextTCB()` to request a pointer to the next task's control block. Write a pseudocode function that would run on your microcontroller to accomplish the context switch.

```
/* Save current task's stack pointer in the current TCB */
/* Use GetNextTCB() to retrieve the pointer to the next task's TCB */
/* Execute return-from-interrupt */
```

Solutions to Appendix Problems

A-2. Decode the ASCII message

```
44 65 73 69 67 6e 69 6e 67 20 77 69 74 68 20 6d 69 63 72
6f 70 72 6f 63 65 73 73
 D  e  s  i  g  n  i  n  g     w  i  t  h     m  i  c  r
 o  p  r  o  c  e  s  s
6f  72 73 20 69 73 20 46 55 4e 21. [a]
 o  r  s     i  s     F  U  N  !
```

A-3. Give the decimal value of the following binary code words assuming (i) unsigned-binary, (ii) two's complement binary, and (iii) signed/magnitude codes. [a]

a. 10101010 *(i) 170 (ii) –86 (iii) –42*

c. 11001100 *(i) 204 (ii) –52 (iii) –76*

e. 10000000 *(i) 128 (ii) –128 (iii) –0*

A-5. Find the decimal equivalent of the following two's complement numbers: [a]

a. 0101101.1 *77.5*

c. 1000 *–8*

A-7. How many bits are required to encode the decimal number 238 using a BCD code? How many using an unsigned-binary code? How many using a two's complement binary code?

BCD = 12 bits; Unsigned Binary = 8 bits; Two's complement = 9 bits

A-9. Find the hexadecimal code words for the following binary code words: [a]

a. 01011010 *5A*

c. 110101 *35*

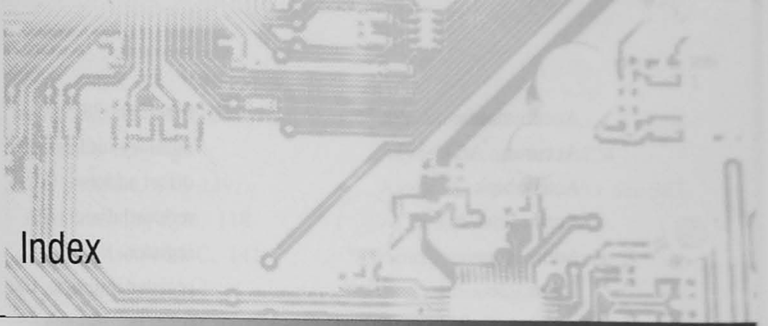
A-11. For four-bit number two's complement addition, choose four examples to demonstrate the following: [a, b]

a. Addition with no carry out and no two's complement overflow.

0111 + 1000 = 1111 no carry, now overflow

c. Addition with carry out and no two's complement overflow.

1111 + 1111 = 1110 with carry, no overflow



Index

Microcontrollers and Microcomputers: Principles of Software and Hardware Engineering, Second Edition, is an ideal introductory text for an embedded system or microcontroller course. While most texts discuss only one specific microcontroller, this book offers a unique approach by covering the common ground among all microcontrollers in one volume.

Since the text does not focus on a particular processor, it can be used with processor-specific material—such as manufacturer's data sheets and reference manuals—or with texts, including author Fredrick M. Cady's *Software and Hardware Engineering: Motorola M68HC11* or *Software and Hardware Engineering: Motorola M68HC12*. Now fully updated, the second edition covers the fundamental operation of standard microcontroller features, including parallel and serial I/O interfaces, interrupts, analog-to-digital conversion, and timers, focusing on the electrical interfaces as needed. It devotes one chapter to showing how a variety of devices can be used, and emphasizes C program software development, design, and debugging.

About the Author

Fredrick M. Cady is Emeritus Professor of Electrical and Computer Engineering at Montana State University. He has been honored with several teaching awards, including MSU Bozeman Mortar Board Professor of the Month, MSU Alumni–Chamber of Commerce Award for Excellence, and the Phi Kappa Phi Anna Krueger Fridley Award for Distinguished Teaching.

ISBN 978-0-19-537161-1
90000
9 780195 371611