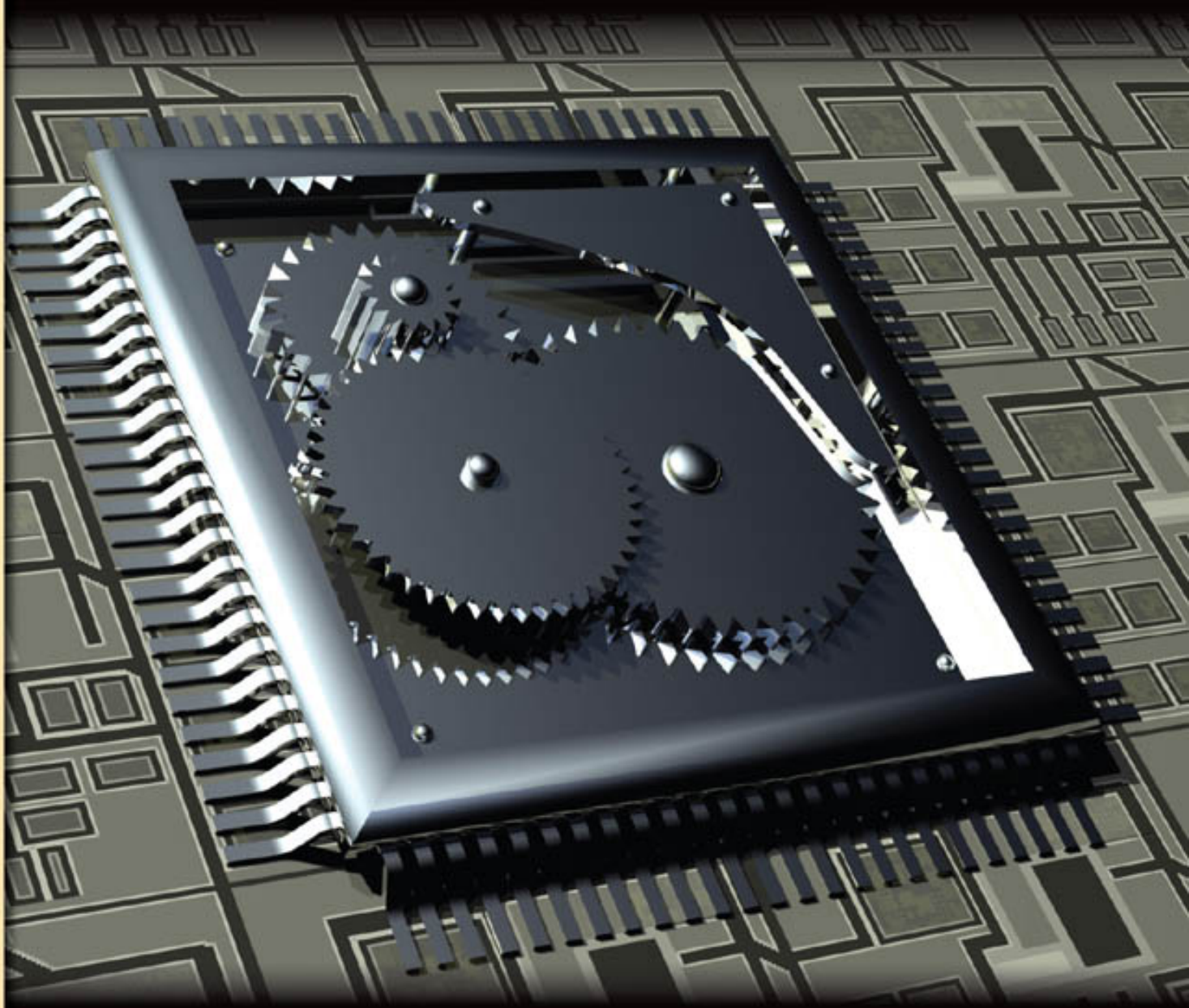
FUNDAMENTALS OF MECHATRONICS



M. JOUANEH

Fundamentals of Mechatronics

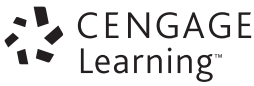
This page intentionally left blank

Fundamentals of Mechatronics

Musa Jouaneh
Department of Mechanical, Industrial, and Systems Engineering
University of Rhode Island

CENGAGE Learning™

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



Fundamentals of Mechatronics
Musa Jouaneh

Publisher, Global Engineering: Christopher M. Shortt

Acquisitions Editor: Swati Meherishi

Senior Developmental Editor: Hilda Gowans

Editorial Assistant: Tanya Altieri

Team Assistant: Carly Rizzo

Marketing Manager: Lauren Betsos

Media Editor: Chris Valentine

Director, Content and Media Production: Patricia M. Boies

Content Project Manager: Jennifer A. Ziegler

Production Service: RPK Editorial Services, Inc.

Copyeditor: Shelly Gerger-Knechtl

Proofreader: Harlan James

Indexer: Shelly Gerger-Knechtl

Compositor: MPS Limited, a Macmillan Company

Senior Art Director: Michelle Kunkler

Internal Designer: Juli Cook/Plan-IT_Publishing

Cover Designer: Andrew Adams/ 4065042 Canada Inc.

Cover Image: © Raimundas/Shutterstock

Rights Acquisitions Specialist: Sam Marshall

Text and Image Permissions Researcher: Kristiina Paul

Senior First Print Buyer: Doug Wilke

Library of Congress Control Number: 2011934121

ISBN-13: 978-1-111-56901-3
ISBN-10: 1-111-56901-0

Cengage Learning
200 First Stamford Place, Suite 400
Stamford, CT 06902
USA

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

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

For your course and learning solutions, visit **www.cengage.com/engineering**.

Purchase any of our products at your local college store or at our preferred online store **www.cengagebrain.com**.

Printed in the United States of America
1 2 3 4 5 6 7 13 12 11

To the LORD who has done wonderful things in my life
and to my lovely wife for her encouragement and support

ABOUT THE AUTHOR

Musa Jouaneh received his B.S. in Mechanical Engineering from the University of Louisiana, Lafayette in 1984 and then went on to the University of California at Berkeley where he received his M.Eng in 1986 and his Ph.D. in 1989. He is currently a Professor of Mechanical Engineering and Applied Mechanics at the University of Rhode Island. His teaching interests include Mechatronics, Robotics, Real-Time Monitoring and Control, and Engineering Mechanics. Professor Jouaneh has been the recipient of several awards including URI Outstanding Contributions to Intellectual Property Award (2004), The URI Foundation Teaching Excellence Award (2003), The 2002–2003 Edmund and Dorothy Marshall Faculty Excellence Award in Engineering, Albert E. Carlotti Faculty Excellence Award in Engineering (1993), and Graduate Fellowship, University of California at Berkeley (1984–1985). Dr. Jouaneh is a member of American Society of Mechanical Engineers (ASME) and a senior member of Institute of Electrical and Electronic Engineers (IEEE).

CONTENTS

PREFACE

Fundamentals of Mechatronics is designed to serve as a textbook for an undergraduate course in Mechatronics Systems Design. It has been written with the primary objective of covering both hardware and software aspects of mechatronics system design in a single text, providing a complete treatment of the subject matter. To design a complete mechatronics system, the student must not only learn about sensors, actuators, microcontrollers and other electronics, but must also understand how to design the software that interacts with these hardware elements. This book lays emphasis on a structured way for developing such software. Software concepts are applicable to both microcontrollers and PC-based systems. Software code examples are presented in C, MATLAB, and Visual Basic Express to appeal to a wide variety of students and instructors.

Content and Organization

Fundamentals of Mechatronics focuses on applications, modeling considerations, and relevant practical issues that arise in the selection and design of mechatronics components and systems. The textbook provides a comprehensive discussion of the use microcontrollers in control of mechatronics systems, using the PIC microcontroller as a vehicle for teaching. It also discusses software topics such as timing, task/states, graphical user interfaces, and Real-Time Operating Systems that are needed to implement control of mechatronics systems. Interfacing of microcontrollers/PCs with mechatronics components is covered, illustrating techniques such as asynchronous serial, synchronous serial, USB, and Ethernet. The book also includes descriptions of several simple-to-build experimental systems that instructors teaching the course can build and use in their courses.

The book is organized into 10 chapters and several appendices. Chapter 1 is an introductory chapter. Chapters 2 and 3 focus on circuits and electronics. Chapters 4–6 focus on microcontrollers, interfacing, and control software development. Chapters 7 and 8 focus on sensors and actuators, while Chapter 9 focuses on the basics of feedback control. The last chapter lists the details of three mechatronics projects.

The textbook uses several programming languages to illustrate key topics. Different programming platforms are presented to give the instructor a choice to select the programming language most suitable for their course objectives. MATLAB is used as tool for modeling and simulation as well as for illustrating timing and Graphical User Interfaces. Visual Basic Express is used for illustrating timing, task/state, and GUI development for code running on a PC. The C language is used for programming of PIC microcontrollers. The author does not expect the student to have any appreciable knowledge of programming except familiarity with basic programming concepts through a prior introductory course on MATLAB. An appendix that covers Visual Basic Express in included in the text. If the student is expected to develop PC-based applications that interact with code running on a microcontroller, then both the VBE and the C-language should be emphasized.

Course Outlines

Although the intended market for this text is junior/senior-level undergraduate students, the text could also be used in an advanced undergraduate/beginning graduate-level course with focus on control software. For a junior/senior level

undergraduate course, some of the sections in Chapter 4 (Microcontrollers), Chapter 5 (Data Acquisition and Microcontroller/PC Interfacing), and Chapter 6 (Control Software) could be skipped. For an advanced undergraduate level/beginning graduate-level course with focus on control software, the course could just focus on the contents of Chapters 1, 4–6, 9–10, and selected topics from the remaining chapters. In the undergraduate mechatronics course (MCE433) that the author teaches at the University of Rhode Island (URI), he covers Chapter 1, most of the material in Chapters 2, 3, and 4, selected topics from Chapter 5, most of Chapter 6, and selected topics from Chapter 7–10, in addition to the appendix on VBE.

Supplements and Ancillaries

A separate laboratory book called *Laboratory Exercises in Mechatronics* is available for purchase through Cengage Learning. *Laboratory Exercises in Mechatronics* details a number of laboratory exercises and projects to facilitate guided hands-on experience with many of the topics covered in this text.

Instructors Solutions Manuals for both *Fundamentals of Mechatronics* and *Laboratory Exercises in Mechatronics* are available from the publisher on request. To request access to the solutions manuals and additional course materials, please visit www.cengagebrain.com. At the cengagebrain.com home page, search for the ISBN of your title (from the back cover of your book) using the search box at the top of the page. This will take you to the product page where these resources can be found.

Acknowledgements

I would like to acknowledge the many students who were enrolled in the mechatronics class at the University of Rhode Island and who had provided useful suggestions and comments which shaped the current manuscript. These include Michael Andrews, Scott Carlson, Anthony Digiulio, William Fanning III, Andrew Krytiun, Daniel Ouellete, Paul Schumacher, and Andrew Wild.

I would also like to acknowledge James Byrnes, the electronic technician in the Mechanical, Industrial, and Systems Engineering Department at URI who has helped in building and wiring some of the circuits used in this book.

I am also thankful to several of my colleagues at the University of Rhode Island who provided valuable comments. These include Professors William Palm and Godi Fisher.

I also wish to acknowledge the valuable comments and suggestions of the manuscript reviewers:

Alan A. Barhorst, Texas Tech University

Jordan M. Berg, Texas Tech University

William W. Clark, University of Pittsburgh

Burford Furman, San Jose State University

Hector Gutierrez, Florida Institute of Technology

Steve Hung, Clemson University

Marcia K. O'Malley, Rice University

Thanks to Swati Meherishi, Acquisitions Editor, and Hilda Gowans, Senior Developmental Editor, of Cengage Learning, for their help in bringing this book to fruition.

The manuscript of this book was class tested at the Pennsylvania State University, State College. The author and the publisher are grateful for the support extended by the instructors at Penn State. We also thank the many students who used the manuscript and provided useful comments.

Musa Jouaneh
Kingston, Rhode Island

This page intentionally left blank

CHAPTER 1

Introduction to Mechatronics

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- Explain what is a mechatronic system
- List the components of a mechatronic system
- Give examples of real-world mechatronic systems
- Give an overview of the topics covered in the text

1.1 INTRODUCTION

Mechatronics is the field of study concerned with the design, selection, analysis, and control of systems that combine mechanical elements with electronic components, including computers and/or microcontrollers. Mechatronics topics involve elements from mechanical engineering, electrical engineering, and computer science, and the subject matter is directly related to advancements in computer technology. The **term 'mechatronics'** was coined by Yasakawa Electric Company [1] to refer to the use of electronics in mechanical control (i.e., '**mecha**' from mechanical engineering and '**tronics**' from electrical or electronic engineering). Auslander, et al. [2] have defined mechatronics as the application of complex decision-making to the operation of physical systems. This definition removes the specific technology to be used to perform the operation from the definition.

A block diagram of a typical mechatronic system is shown in Figure 1.1. A mechatronic system has at its core a **mechanical system** which needs to be commanded or controlled. Such a system could be a vehicle braking system, a positioning table, an oven, or an assembly machine. The controller needs information about the state of the system. This information is obtained from variety of **sensors,** such as those that give proximity, velocity, temperature, or displacement information. In many cases, the signals produced by the sensors are not in a form ready to be read by the controller and need some **signal conditioning operations** performed on them. The conditioned, sensed signals are then converted to a digital form (if not already in that form) and presented to the controller.

The **controller** is the 'mind' of the mechatronic system, which processes user commands and sensed signals to generate command signals to be sent to the actuators in the system. The **user commands** are obtained from a variety of devices, including command buttons, graphical user interfaces (GUIs), touch screens, or pads. In some cases, the command signals are sent to the actuators without utilizing any feedback information from the sensors. This is called open-loop operation, and

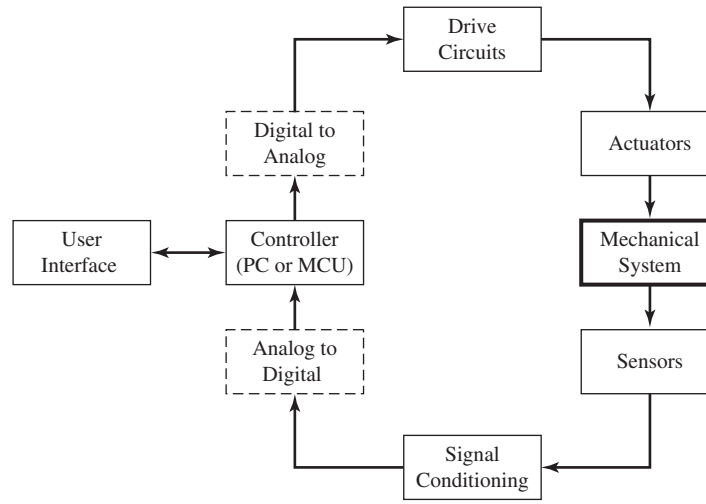


Figure 1.1
Typical components of a mechatronics system

for it to work, this requires a good calibration between the input and output of the system with minimal disturbances. The more common mode of operation is the closed-loop mode in which the command signals sent to the actuators utilize the feedback information from the sensors. This mode of operation does not require calibration information, and it is much better suited for handling disturbances and noise.

In many cases, the command signals to the actuators are first converted from a digital to an analog form. Amplifiers implemented in the form of **drive circuits** also can be used to amplify the command signals sent to the actuators. The **actuator** is the mechanism that converts electrical signals into useful mechanical motion or action. The choice of the controller for the mechatronic system depends on many factors, including cost, size, ease of development, and transportability. Many mechatronic systems use personal computers (PCs) with data acquisition capabilities for implementation. Examples include control of manufacturing processes such as welding, cutting, and assembly. A significant number of controllers for a mechatronic system are implemented using a microcontroller unit (MCU), which is a single-chip device that includes a processor, memory, and input-output devices on the same chip. Microcontrollers often are used for control of many consumer devices, including toys, hand-held electronic devices, and vehicle safety systems. Control systems that use MCUs often are referred to as **embedded control systems**.

The control system for a mechatronic system can be classified as either a discrete-event control system or a feedback control system. In a **discrete-event system,** the controller controls the execution of a sequence of events, while in a **feedback control system**, the controller controls one or more variables using feedback sensors and feedback control laws. Almost all realistic systems involve a combination of the two. This textbook will discuss these two classes in detail.

A mechatronic system integrates mechanical components, electronic components, and software implemented either on a PC or MCU to produce a flexible and intelligent system that performs the complex processing of signals and data. In many cases, a mechatronic system can be used to improve the performance of a system beyond what can be achieved using manual means. An example includes the speed control of rotating equipment. In some cases, a mechatronic system is the only means by which that system can operate (such as the control of magnetic bearings and in nano-positioning control applications).

1.2 Examples of Mechatronic Systems

Modern society depends on mechatronic-based systems for its conveniences and luxurious standard of living. From intelligent appliances to safety features in cars (such as air bags and anti-lock brakes), mechatronic systems are widely used in everyday life. The availability of low-cost, compact, and powerful processors in the form of MCUs accelerated the widespread use of mechatronic systems. An example is the use of embedded controllers to control many of the devices in a vehicle. A list of such applications is shown in Table 1.1.

Table 1.1

Listing of sample applications of mechatronic systems in vehicles

Application Area		
Safety	**Comfort**	**Power Train**
• Airbag system	• Door locks	• Engine controls
• Anti-lock breaking system	• Keyless entry system	• Fuel pump controls
• Daytime running light	• Heating system controls	• Fuel sensing controls
• Electronic stability controls	• Seat positioning controls	• Gearbox controls

To further illustrate mechatronic systems, we will discuss four available systems: an industrial robot, a mobile robot, a flatbed scanner, and a parking-garage gate.

Industrial Robots Robots, whether of the fixed type (such as industrial robots) or of the mobile type, are good examples of mechatronic systems. Figure 1.2 shows an industrial robot arm. A **robot** is a mechanical device that can be programmed to perform a wide variety of applications. The main components of a robot system are the controller and the mechanical arm. The controller handles several operations, including the user interface, programming, and control of the arm. The mechanical arm consists of several mechanical links that are connected at joints. An actuator is used to drive each link, and each actuator has a feedback sensor to indicate the location of the link. A multi-link robot is a complicated device that requires coordination of the motion of the links. This job is done by the control software, which processes information from the desired motion of the arm, and the feedback sensors, which send commands to the actuators or the servomotors to perform the desired task. To enable a robot to handle variation in the environment in which it operates, additional sensors are normally used (such as vision and proximity).

Figure 1.2

Industrial robot
(© Balonicci/ Shutterstock.com)

Mobile Robots **Mobile robots** are currently being used in a wide diversity of applications. Whether vacuum cleaning, assisting soldiers in combat operations, or delivering food and medicine in hospitals, their use is increasing. Similar to their fixed counterparts, a mobile robot consists of a number of modules that are commanded by a controller. Due to their operation in unstructured environments, mobile robots rely heavily on sensors to guide them in navigation and to avoid obstacles. Examples of sensors used by mobile robots include ultrasonic proximity sensors, vision sensors, and global positioning system sensors. An example of a mobile robot is the Roomba® vacuum-cleaning robot (see Figure 1.3) made by iRobot® Corporation. The Roomba has a cylindrical shape, two wheel modules, and a sensor to detect obstacles. The Roomba has all of the main components of a mechatronic system: actuators (wheel modules), sensors (target and dirt), and a controller.

Figure 1.3

Roomba® vacuum-cleaning robot
(iRobot Corporation, Bedford, MA)

Figure 1.4

A flatbed scanner
(© CreativeAct-Technology series/Alamy)

Scanner A **scanner** (see Figure 1.4) is a device that captures an image of a document and converts it into a format suitable for electronic storage. The main components of a scanner include the scanning head, the transport device, the controller, and the control software. The controller commands the transport device which carries the scanner head. The transport device uses a stepper motor and a system of gears and belts to move the scanning head in precise steps. After each step, the transport device stops, and a scan is sampled. The scanning head involves some form of a line camera that measures the reflectivity of a scanned line. The scanned line is brought to the scan sensor through a system of mirrors and lenses. The output of the scanning head is processed by the control software to create a map of the scanned document. This map is further analyzed to reveal all of the features in the document and to filter any noise signals from the captured data. The control software sequences the operation of the scanner and communicates with the PC. When the scanning job is completed, the scanned image is then transferred to a PC using a USB or a parallel-port connection. This mechatronic system involves all of the elements of a typical control system: sensor, actuator, and controller. It is also an example of a discrete-event system.

Figure 1.5

Parking gate
(© BigPileStock/Alamy)

Parking Gate A **parking garage gate** (see Figure 1.5) is another example of a mechatronic system that involves a number of elements. The system has an electric motor to raise and lower the gate arm. It also has a proximity sensor to prevent the gate from striking people and vehicles. In addition, it has a microcontroller in which software is used to run the gate in different operating modes. Typically, a parking-garage gate operates as follows: The user presses a button to get a ticket or swipes a card in a card scanner. Once the ticket is picked up by the user or the card is validated, the gate arm rotates upward. The gate arm remains in a raised position until the vehicle has completely cleared the gate, at which point the gate drops down. The operation of each stage of this system is dependent on sensor feedback and timing information. The controller for this system cycles between the different operating stages each time a vehicle needs to enter the parking garage.

The previous examples illustrate a wide range of mechatronic systems. With the increased use of automation systems in manufacturing and the integration of mobile robots in many applications, the study of mechatronics will further increase. It should be noted that mechatronics encompasses many **enabling technologies** that are key to the design, operation, and control of modern, smart systems. These technologies include signal processing, system interfacing, sensor integration, drive technology, actuation systems, software programming, and motion-control systems.

1.3 Overview of Text

This book covers topics that are needed to design, analyze, and implement a complete mechatronic system. The text is organized into ten chapters and several appendices and covers all of the areas related to the mechatronic components shown in Figure 1.1. **Chapter 2** covers the basics of analog circuits and components. Virtually every mechatronic device has some form of an electric circuit, and thus understanding and analyzing electrical circuits is important in mechatronics. **Chapter 3** discusses the operation of semiconductor electronic devices (such as diodes, thyristors, and transistors) that are used in many circuits and devices for switching or amplification purposes. It also covers digital circuits. In many situations, the current and voltage capabilities of interface devices available on PCs and MCUs are not adequate to operate real devices (such as motors and heaters) and semiconductor electronic components (such as transistors) are needed. **Chapter 4** discusses the use and programming of microcontrollers in detail. The objective of this chapter is to give the reader complete coverage of the features and capabilities of a typical microcontroller. Unlike combinational and sequential circuits, microcontrollers and microprocessors offer a flexible but complex method to implement control logic.

Chapter 5 discusses techniques to interface a processor to the outside world using different interface devices (such as analog-to-digital converters, digital-to-analog converters, digital input/output ports, asynchronous and synchronous serial ports, Internet, and USB). **Chapter 6** focuses on software development issues when using a microcontroller (and to a less extent, a PC) as the controller in a mechatronic system. Some of these issues include how to incorporate time into a control program, how to structure the operation and control of physical systems into tasks and states, and how to write control code that is suitable for real-time implementation. A software-based control system offers flexibility over a hardware-based one, since the controller structure and control logic can be changed by simply changing the code in the program. **Chapter 7** focuses on sensors. Sensors are vital components of mechatronic systems, since they provide the feedback information that enables automated systems to function. A sensor is an element that produces an output in response to changes in physical quantity (such as temperature, force, or displacement). **Chapter 8** discusses actuators, which are the key components of all mechanized equipment. An actuator is a device that converts energy to mechanical motion. This chapter focuses on electrically powered actuators, which are commonly used in mechatronic systems. **Chapter 9** covers the basics of feedback control systems. The objective is to illustrate to the reader the design, simulation, and implementation of basic feedback control systems. **Chapter 10** discusses several experimental systems that are suitable for extended or final project topics. These projects are intended to illustrate the integration of the various topics covered in the text. The text has also several appendices. **Appendix A** gives a detailed overview of Visual Basic Express. **Appendix B** covers basics of system response, while **Appendix C** discusses MATLAB simulation of dynamic systems. **Appendix D** has a list of 7-bit ASCII codes.

Questions

1.1 What is mechatronics?

1.2 What are the elements of a mechatronic system?

1.3 How are mechatronic systems implemented?

Problems

P1.1 Perform research on mechatronic systems that are used in vehicles. Identify the type of sensors and/or actuators that are used in the following systems.

a. Air bag

b. Door locks

c. Powered side mirrors

P1.2 Research and identify all the mechatronic components used in the following devices.

a. Modern washing machine

b. Servo-driven industrial robot

c. Automated entry door

CHAPTER 2

Analog Circuits and Components

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- Explain the characteristics of basic circuit components
- Explain the different types of switches
- Perform circuit analysis using Kirchhoff's voltage law (KVL) and Kirchhoff's current law (KCL)
- Determine an equivalent circuit for a given two-terminal circuit
- Explain the concept of impedance and loading effects
- Determine the RMS current and voltage in an AC circuit
- Determine the components of power in an AC circuit
- Analyze different op-amp circuits
- Explain proper grounding techniques
- Explain the function of a solenoid
- Use a relay as an interface element

2.1 Introduction

Virtually every mechatronic device has some form of an electric circuit, and thus, understanding and analyzing electrical circuits is important in mechatronics. A **circuit** is defined as a closed path through a series of electronic components in which a current flows through. Electric circuits can be of the analog or digital type. In **analog** circuits, the voltage is continuous and can have any value over a specified range, while in a **digital** circuit, the voltage signal is usually represented by just two different levels (such as 0 and 5 volts (V)). Analog circuits are more sensitive to noise or disturbances than digital circuits. In an analog circuit, any noise in the circuit is translated into changes in the analog signal or a loss of information, while in a digital circuit, small disturbances have no effect. As long the signal stays within a specified range in a digital circuit, it represents the same information. Digital circuits are used to perform logic operations using hardware instead of software.

Two basic quantities in electrical circuits are voltage or electric potential, and current. **Voltage** refers to the capability of driving a stream of electrons through a circuit, similar to the concept of a force in a mechanical system, while **current** is a measure of the flow of charge in the circuit. The unit of measurement for current is Amperes (A). The time integral of the current is defined as **electrical charge** and has units of Coulombs (C). When the voltage or current does not change its value with respect to time in a circuit, it is called a direct current (DC) circuit. On the

other hand, when the voltage or current varies sinusoidaly with time it is called an alternating current (AC) circuit.

This chapter covers the basics of analog circuits and components. Digital circuits are covered in the next chapter. For further reading, see [3-6].

2.2 Analog Circuit Elements

Circuit elements include power sources (such as a power supply or a battery), switches to open and close the circuit, and circuit components (such as resistors and capacitors). A schematic of an electrical circuit is shown in Figure 2.1. The figure shows a closed loop where a conducting element (such as a copper wire) connects a voltage source (or a current source) to load elements on the circuit.

Figure 2.1

A schematic of an electrical circuit

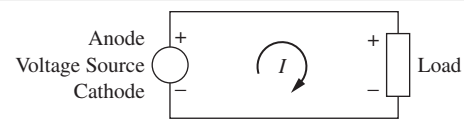


Figure 2.1 also shows the direction of conventional current flow from the **anode** (positive terminal) of the power source to the **cathode** (negative terminal). The conventional current flow direction is the opposite of the direction of actual electron flow in the circuit but was kept due to Benjamin Franklin, who thought that electrical current is due to the motion of positively charged particles.

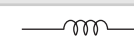
Circuit components can be of the passive type, which require no external power to operate (such as resistors and capacitors), or active components which require power to operate (such as operational amplifiers). We will focus on the three basic passive circuit elements in this section, namely, the resistor, the capacitor, and the inductor. Table 2.1 shows the electrical symbols for these elements. The table also shows the symbols for two energy sources that are normally represented in circuits. These include an ideal voltage source, and an ideal current source. These sources are considered ideal because they do not have any internal resistance, capacitance, or inductance.

The **resistor** is an element that dissipates energy. The constitutive relation for an ideal resistor is given by **Ohm's law**, in which the voltage drop across the resistor is linearly related to the current through the resistor, or

$$V = IR \tag{2.1}$$

Table 2.1

Symbols of basic circuit elements

Element	Reference	Circuit Symbol
Resistor	R	
Capacitor	C	
Inductor	L	
Ideal Voltage Source	V	
Ideal Current Source	I	

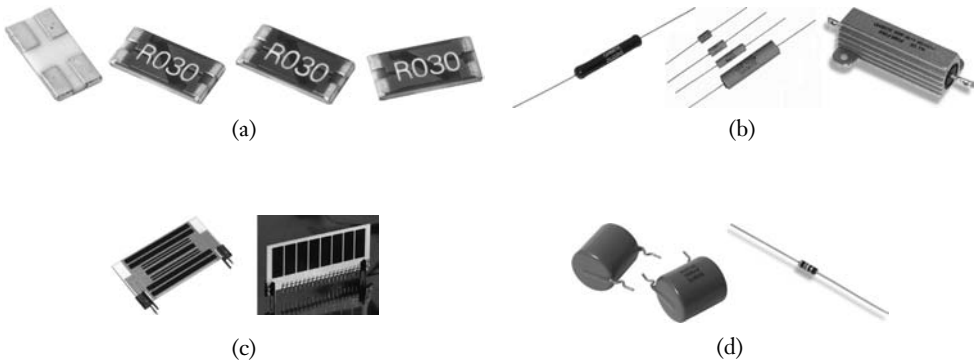


Figure 2.2

Resistor types
(a) surface mount,
(b) wire wound,
(c) thick film, and
(d) carbon composition

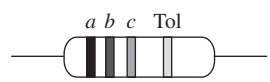
(Courtesy of Ohmite Mfg. Co., Arlington Heights, IL)

The resistance is measured in units of ohms (Ω). Resistors can be either of the fixed type or variable. Fixed-type resistors are made in made in a variety of forms including surface mount, wire wound, thick film, and carbon composition (see Figure 2.2).

Typical fixed-type, low-wattage resistors are made of molded carbon composition and have a resistance that ranges from a few ohms to about 20 MΩ. These resistors have a cylindrical shape, and have sizes that increase with the power rating of the resistor. Typical power rating is 1/4 to 1 Watt (W). The resistance can be read from the color code printed on the resistors. Typically, four color bands are shown on the resistor, as shown in Figure 2.3, but resistors with five or six color bands are also available. For four color bands, the left three bands give the resistor value, while the fourth band gives the resistance tolerance (Tol). The resistance is given by the formula

Figure 2.3

Resistor color bands



$$R = ab \times 10^{c}(\mp\%\text{Tol}) \tag{2.2}$$

where the *a* band is the value of the tens digit, the *b* band is the value of the ones digit, the *c* band is the base-10 exponent power value, and the Tol band gives the tolerance or expected percentage variation in the resistor value. Table 2.2 gives these values.

Table 2.2

Resistor bands color code

a, b, c	Value	Tol Band	Value
Black	0	Silver	10%
Brown	1	Gold	5%
Red	2	Brown	1%
Orange	3	Red	2%
Yellow	4	Green	0.5%
Green	5	Blue	0.25%
Blue	6	Violet	0.1%
Violet	7	Gray	0.05%
Gray	8		
White	9		

As an example, a resistor whose bands are colored brown, black, orange, and silver has a resistance of 10k +/−10% ohms. The resistance of real resistor is not actually constant, but it increases with temperature.

Commercial resistors are available in certain preferred values. These values are dependent on the resistance tolerance. For example, preferred resistor values include 1 Ω, 100 Ω, 10 kΩ, and 1 MΩ; 22 Ω, 2.2 kΩ, and 220 kΩ; or 16 Ω, 1.6 kΩ, and 160 kΩ.

Variable-type resistors include rheostats and potentiometers. **Rheostats** are two-terminal resistors, while **potentiometers** are three-terminal resistors. They can be of the linear or rotary type, and the resistance between the terminals is changed as the position of the wiper terminal is changed.

Unlike a resistor, a **capacitor** is an energy storage element. The constitutive relation for a capacitor is

$$\frac{dV}{dt} = \frac{1}{C} I \tag{2.3}$$

where C is the capacitance in Farads (F). Small capacitors are typically of the ceramic type, which can be used in both AC and DC circuits. These capacitors have capacitance that is less than 0.1 micro-Farads (μF). Capacitors with large capacitance (up to several thousand micro Farads) are of the electrolytic type. These are used only in DC circuits, and their leads are polarized. One characteristic of capacitors is the leakage current, which is the current that flows between the capacitor plates when a voltage is applied across the plates of the capacitor. This current leads to the loss of charge over time from the capacitor. This current, however, is typically small, unless the capacitor is of the electrolytic type. Similar to resistors, capacitors are also available as fixed or variable type.

An **inductor** is also an energy storage element. Inductive elements in practice include solenoids and motors. The constitutive relation for an inductor is

$$\frac{dI}{dt} = \frac{1}{L} V \tag{2.4}$$

where L is the inductance, and it is measured in units of Henry (H). Small-sized inductors are of the molded type, and they have inductance that varies from submirco to several thousand microHenry (μH).

2.3 Mechanical Switches

Mechanical switches are devices that make or break contact in electrical circuits. There are a variety of mechanical switches available, including toggle, push-button, rocker, slide, and others (see Figure 2.4).

Toggle switches are specified in terms of their number of poles and throws. **Poles** refer to the number of circuits that can be completed by the same switching action, while **throws** refer to the number of individual contacts for each pole. Figure 2.5 shows four different configurations of toggle switches. In Figure 2.5(a), a single-pole, single-throw (SPST) switch is shown, which is the configuration of basic switches (such as on-off switches and mechanical contact limit switches). In Figure 2.5(b), a single-pole, double-throw (SPDT) switch is shown. The Figure 2.5(b) configuration is commonly used in the residential wiring of rooms that have two switches to operate a light fixture, and Figure 2.6 shows an example of such a circuit which uses two SPDT switches. Note that the SPDT switch is commonly known as a 'three-way switch.' Figure 2.5(c) shows a double-pole,

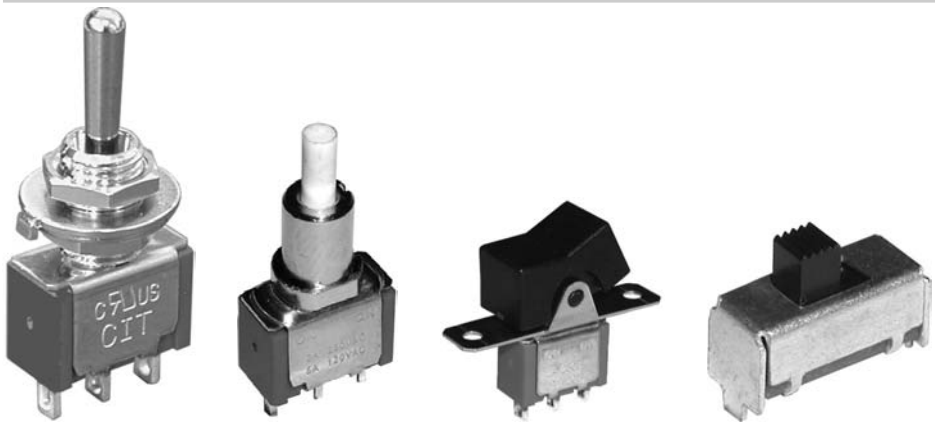


Figure 2.4

Mechanical switches (a) toggle, (b) push-button, (c) rocker, and (d) slide

(Courtesy of CIT Relay & Switch, Minneapolis, MN)

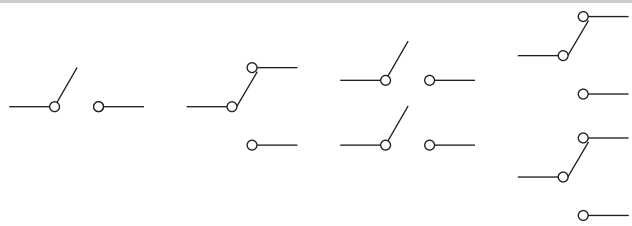


Figure 2.5

Different configurations of toggle switches

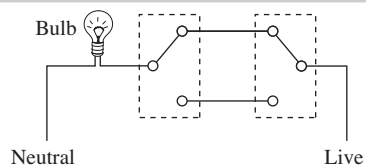


Figure 2.6

Wiring circuit for a light bulb using two SPDT switches

single-throw (DPST) switch, which is equivalent to two SPST switches controlled by a single mechanism. Figure 2.5(d) shows a double-pole, double-throw (DPDT) switch configuration. This configuration is commonly used in the construction or electromechanical relays (to be discussed in the next section).

A DPDT switch which is internally wired for polarity reversal applications is commonly called 'a four-way switch' (see Figure 2.7). Such a switch has only four wires coming out of it (instead of six) and can be inserted between two SPDT switches to enable wiring of a single light bulb using three switches (see Problem 2.3).

Toggle switches are known as 'break before make' type, which means that the switch pole never connects to both terminals in SPDT or DPDT switch configuration. **Push-button switches** have the symbol shown in Figure 2.8. They can be either of two types: normally open (NO) or normally closed (NC). Normally open or normally closed refer to the state of the switch before it is activated. Push-button switches are widely used as reset switches and doorbell switches.

One disadvantage of mechanical switches is **switch bouncing**. Since the switch arm is typically a small flexible element, the opening and closing of mechanical switches causes the switch to bounce a number of times before settling at its desired state. Figure 2.9 shows a typical pattern in closing a switch. Note that each of the contacts during the bouncing interval, which is typically about 15 to 25 ms long, may register by a processor as separate switch action unless means were incorporated to address this issue. The most common approach to solve this problem is to

Figure 2.7

DPDT switch wired as 'a four-way switch'

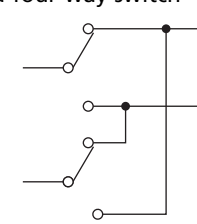


Figure 2.8

Push-button switch (a) normally open and (b) normally closed

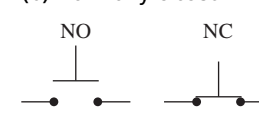
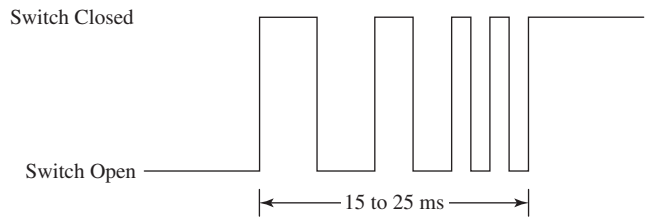


Figure 2.9
Switch bounce pattern for switch closure

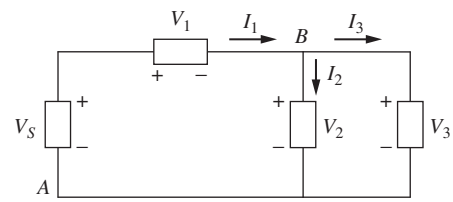


provide for each switch a debouncing circuit that makes use of flip-flop circuit elements (to be discussed in Chapter 3).

2.4 Circuit Analysis

A typical analog circuit is shown in Figure 2.10. The objective of circuit analysis is to determine the voltage and current at any point in the circuit. This is done with the aid of two laws: Kirchhoff's voltage law (KVL) and Kirchhoff's current law (KCL).

Figure 2.10
A typical electric circuit



Kirchhoff's voltage law states that the algebraic sum of the voltage drops and rises around any closed path in a circuit is zero. In equation form, it is stated as

$$\sum_{i=1}^{i=N} V_i = 0 \tag{2.5}$$

where N is the number of elements in the selected path. To illustrate this, consider the left loop of the circuit shown in Figure 2.10. Starting at any point in the circuit (such as point A), and going clockwise, we get

$$V_s - V_1 - V_2 = 0 \tag{2.6}$$

In getting the Equation (2.6), potential rises are considered positive, and potential drops are considered negative. As we go around the loop, the potential of the voltage source V_s rises (goes from $-$ to $+$), and the potential across the second and third elements drops (goes from $+$ to $-$). We would get an equivalent expression if we went counterclockwise around the loop. Here the sign of the potential drops and rises for the three elements would be opposite, but the final form is equivalent.

Kirchhoff's current law states that sum of the current into a node is zero, or in equation form:

$$\sum_{i=1}^{i=N} I_i = 0 \tag{2.7}$$

With reference to Figure 2.10, KCL gives the following relationship between the currents at node B in the circuit.

$$I_1 - I_2 - I_3 = 0 \tag{2.8}$$

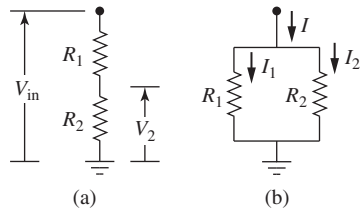


Figure 2.11

(a) Voltage dividing circuit and (b) current dividing circuit

where the current is considered positive if it goes into the node, and is negative if it leaves the node. Example 2.1 illustrates the use of KVL and KCL.

When two resistors are connected in series in a circuit, as in Figure 2.11(a), it is called a **voltage dividing circuit** because the voltage is divided among the two resistors, with the voltage drop across each resistor being proportional to the resistance of each resistor. For example, the output voltage across resistor R_2 is given by

$$V_2 = \frac{R_2}{R_1 + R_2} V_{in} \tag{2.9}$$

Similarly, when two resistors are connected in parallel in a circuit, as in Figure 2.11(b), it is called a **current dividing circuit**. The current through resistor R_2 is given by

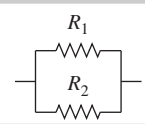
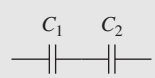
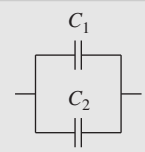
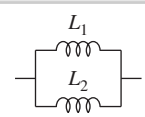
$$I_2 = \frac{R_1}{R_1 + R_2} I \tag{2.10}$$

Note that the current through R_2 is the product of the input current I and the resistance of the other resistor R_1 divided by the sum of the resistance of the two resistors in the circuit.

In many circuits, we could have several elements in serial or parallel configuration. Table 2.3 gives the total resistance, capacitance, and inductance for serial and parallel combinations of these elements. The total values expressions can be derived by using the constitutive relation for the element in question and by applying KVL or KCL.

Table 2.3

Total resistance, capacitance, and inductance

Element	Series Connection	Parallel Connection
Resistor		
	$R_T = R_1 + R_2$	$1/R_T = 1/R_1 + 1/R_2$
Capacitor		
	$1/C_T = 1/C_1 + 1/C_2$	$C_T = C_1 + C_2$
Inductor		
	$L_T = L_1 + L_2$	$L_T = 1/L_1 + 1/L_2$

Example 2.1 Application of KVL and KCL

For the circuit shown in Figure 2.12, determine the voltage drop across each of the three resistors as well as the current through each one of them.

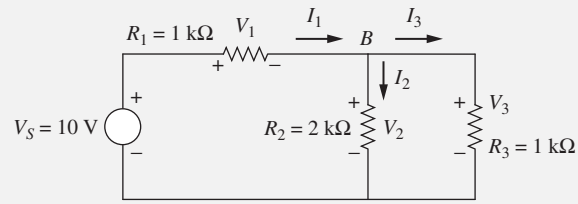


Figure 2.12

Solution:

Applying KCL at node *B*, we get

$$I_1 = I_2 + I_3 \tag{1}$$

Applying KVL to the left loop gives

$$10 = V_1 + V_2 \tag{2}$$

From Ohm's law applied to each resistor and using Equation (2), we get

$$I_1 = (V_S - V_2)/R_1,\ I_2 = V_2/R_2, \text{ and } I_3 = V_3/R_3 \tag{3}$$

From KVL applied to the right loop, we get

$$V_3 - V_2 = 0 \tag{4}$$

Substituting the expressions in Equations (3) and (4) into Equation (1), we get

$$(10 - V_2)/R_1 = V_2/R_2 + V_2/R_3$$

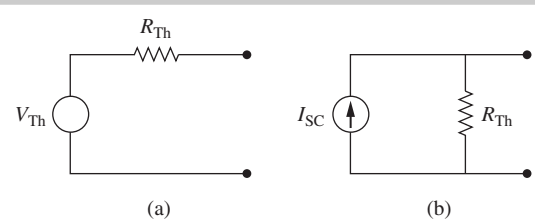
Solving for V_2, we get $V_2 = 4$ V. Substituting this result into Equations (2) and (4) gives $V_1 = 6$ V, $V_2 = V_3 = 4$ V, $I_1 = 6$ mA, $I_2 = 2$ mA, and $I_3 = 4$ mA.

2.5 Equivalent Circuits

For any two-terminal circuit or network that has only resistive elements (or any type of elements if impedance (see next section) is used instead of resistance), the circuit can be simplified into one of two forms. These are the Thevenin equivalent circuit or the Norton equivalent circuit. These simplified forms allow us to focus on a specific portion of a network by replacing the remaining network with an equivalent circuit. The **Thevenin equivalent circuit**, as in Figure 2.13(a) consists

Figure 2.13

(a) Thevenin equivalent circuit and (b) Norton equivalent circuit



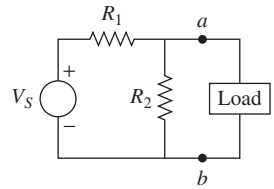
of an ideal voltage source (V_{TH}) connected in series with a resister (R_{TH}). The value of V_{TH} is the open-circuit voltage of the original circuit at the terminals. R_{TH} is defined as the ratio of the open-circuit voltage V_{TH} to the short-circuit current (I_{SC}), where the short-circuit current is the current that would flow through the terminals if the terminals were short circuited. R_{TH} alternatively can be found by determining the equivalent resistance at the terminals when the voltage sources are shorted and the current sources are replaced with open ones.

The **Norton equivalent circuit**, as in Figure 2.13(b) consists of an ideal current source I_{SC} connected in a parallel with a resistor R_{TH}.

As an illustration, consider the circuit shown in Figure 2.14. We would like to replace the circuit to the left of the nodes *a* and *b* with an equivalent Thevenin circuit. The open-circuit voltage of the left circuit at the terminals *a* and *b* is simply the voltage across the R_2 resistor when the load is disconnected. This voltage is given by

Figure 2.14

Circuit to be replaced with a Thevenin equivalent circuit



$$V_{TH} = \frac{R_2}{R_1 + R_2} V_S \tag{2.11}$$

To determine R_{TH}, we need first to find the short-circuit current I_{SC}. This is the current that would flow through the terminals when the terminals *a* and *b* are shorted with the portion of the circuit to the right of the *a* and *b* terminals removed. This current is simply

$$I_{SC} = \frac{V_S}{R_1} \tag{2.12}$$

since no current will flow through R_2. Thus R_{TH} is

$$R_{TH} = \frac{V_{TH}}{I_{SC}} = \frac{R_1 R_2}{R_1 + R_2} \tag{2.13}$$

Alternatively, we can determine R_{TH} by determining the resistance at the *a* and *b* terminals when the supply voltage is short circuited. In this case, the two resistors R_1 and R_2 act as two resistors in parallel, and the resistance at the terminals is the effective resistance of these two resistors. Thus,

$$R_{TH} = R_{Eff} = \frac{R_1 R_2}{R_1 + R_2} \tag{2.14}$$

This concept is further illustrated in Example 2.2.

Example 2.2 Thevenin Equivalent Circuit

Determine the Thevenin equivalent circuit for the circuit shown in Figure 2.15. Let $V_S = 8$ V, $R_1 = 2\ \Omega$, and $R_2 = R_3 = 4\ \Omega$.

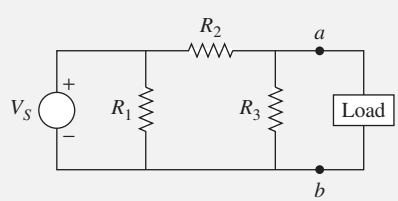


Figure 2.15

Solution:

We start by determining the open-circuit voltage at the terminals *a* and *b* when the load is removed. This is the same as the voltage drop across the R_3 resistor. Note that with no connection between terminals *a* and *b*, the resistors R_2 and R_3 are in series, and the two of them are in parallel with the R_1 resistor. The voltage drop across R_2 and R_3 is the same as the voltage across the R_1 resistor or 8 V. Using the voltage dividing rule, the voltage across R_3 is

$$V_{TH} = V_{R_3} = \frac{4}{4+4}8 = 4 \text{ V}$$

To determine R_{TH}, we find the total resistance of this circuit at the terminals *a* and *b* when V_S is short circuited. In this case, the R_1 resistance does not come to play, and the resistance of this network is the parallel combination of the resistors R_2 and R_3, which is 2 Ω. Thus, R_{TH} is 2 Ω. The Thevenin equivalent circuit is shown in Figure 2.16.

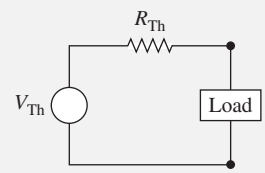


Figure 2.16

Note if the load resistance is 8 Ω, then the current through the load is simply

$$I_L = \frac{V_{TH}}{R_{TH} + R_L} = \frac{4}{2+8} = 0.4 \text{ A}$$

The reader should verify that the same value of current would have been obtained if the original circuit was analyzed.

2.6 Impedance

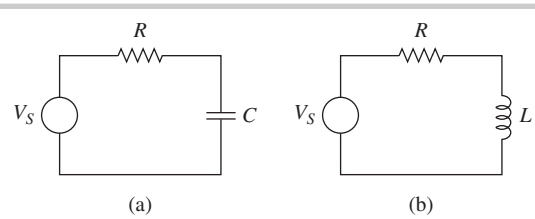
The concept of impedance is useful in analyzing loading effects that occur when measurement devices are connected to circuits. Impedance is a generalization of the concept of resistance. The impedance of a circuit that has only resistive elements is simply the resistance of the circuit. We define impedance for a two-terminal element as the ratio of the voltage to the current in that element or

$$Z = \frac{V}{I} \tag{2.15}$$

For circuits that contain other than resistive elements, the impedance is a function of the AC excitation frequency. We will obtain the impedance of different types of circuits under sinusoidal AC excitation. This allows us to see the effect of frequency on impedance. We will start with the RC circuit shown in Figure 2.17(a).

Figure 2.17

(a) RC circuit and (b) RL circuit



Applying KVL, the relationship between the supply voltage and the current through the circuit is given by

$$V_S(t) = Ri(t) + \frac{1}{C}\int i(t)\,dt \tag{2.16}$$

Using the Laplace Transform, we can convert Equation (2.16) to an algebraic equation in the Laplace variable s as

$$V_S(s) = Ri(s) + \frac{1}{Cs}i(s) \tag{2.17}$$

Solving for the ratio of $V_S(s)$ to $i(s)$, we obtain

$$\frac{V_S(s)}{i(s)} = \left[R + \frac{1}{Cs}\right] \tag{2.18}$$

Substituting $s = j\omega$ where ω is the angular frequency and $j = \sqrt{-1}$ to obtain this ratio as function of ω, we obtain

$$Z = \frac{V_S(j\omega)}{i(j\omega)} = \left[R + \frac{1}{j\omega C}\right] \tag{2.19}$$

Equation (2.19) shows that the impedance of an RC circuit is a complex quantity that is a function of the excitation frequency ω. If the resistor in the above circuit was removed, then the impedance due to the capacitor is

$$Z = \frac{1}{j\omega C} = -\frac{1}{\omega C}j \tag{2.20}$$

Equation (2.20) shows that the impedance of a capacitor is infinite with DC excitation, and approaches zero as the frequency goes to infinity. In the same fashion, we can see that the impedance of a resistor is independent of the excitation frequency and just equal to R. Using a similar approach to that used with the RC circuit, the impedance of the RL circuit shown in Figure 2.17(b) can be obtained as

$$Z = \frac{V_S(j\omega)}{i(j\omega)} = [R + j\omega L] \tag{2.21}$$

The impedance due to the inductor is simply given by

$$Z = \frac{V_S(j\omega)}{i(j\omega)} = j\omega L \tag{2.22}$$

Similar to the capacitor, Equation (2.22) shows that impedance of an inductor is a complex quantity that depends on the frequency ω. For DC excitation, an inductor has zero impedance and acts as a short circuit, but as the frequency approaches infinity, its impedance goes to infinity and acts as open circuit.

Because impedance can be a complex quantity, we can express impedance as

$$Z = R + jX \tag{2.23}$$

where the real part of Z is defined as the resistance R, and the imaginary part of Z is defined as the **reactance** X. Thus, the reactance of a capacitor is

$$X_C = -\frac{1}{\omega C} \tag{2.24}$$

and that of an inductor is

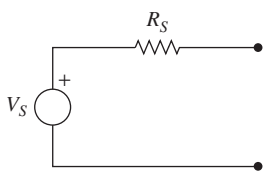
$$X_L = \omega L \tag{2.25}$$

Note that similar to resistors, the total impedance of several elements arranged in series is the sum of the individual impedances of the elements.

Measurement devices such as voltmeters and oscilloscopes are not ideal. They have finite input impedances that could affect the value of the measured quantity. Similarly, amplifiers are not ideal devices, but have finite input impedances that could affect the output of the amplifier. Also, power supply sources are not ideal and have small output impedances. When any of these devices is interfaced to a circuit, they create **loading effects** which are explained below. In general, it is desirable for a voltage source to have a very small output impedance and for a measurement device or amplifier to have a very large input impedance.

Figure 2.18

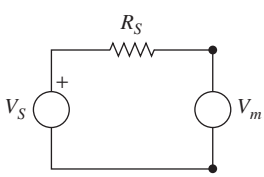
Voltage source



To illustrate loading effects, assume we have a voltage source V_S connected in series with a resistance R_S, as shown in Figure 2.18. This voltage source can be the output of a sensor or the output of a real power supply. Now assume that the value of this voltage will be measured by a multimeter or an oscilloscope. If we assume an ideal meter, then the meter has an infinite input impedance and will draw no current. Such an arrangement is shown in Figure 2.19. Because the 'ideal' voltmeter draws no current, the voltage measured by the ideal voltmeter will be the open-circuit voltage of the voltage source or V_S.

Figure 2.19

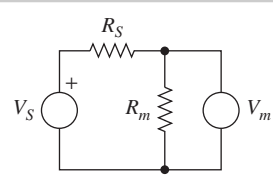
Measuring using ideal voltmeter



Now assume that the ideal voltmeter is replaced with a real-voltmeter that has finite impedance. Such meter can be represented as an ideal voltmeter in parallel with the voltmeter finite impedance (R_m). When this meter is connected to the voltage source, (see Figure 2.20), the output of the voltmeter will be the voltage drop across the R_m resistor.

Figure 2.20

Measuring using a real voltmeter



This value is given by the expression:

$$V_m = \frac{R_m}{R_m + R_S} V_S \tag{2.26}$$

If R_m is much larger than R_S, then the ratio $\frac{R_m}{R_m + R_S}$ is almost equal to 1, and V_m will be close to V_S. If however, R_m is comparable to R_S, then the measured voltage by the voltmeter will be significantly different from V_S. Multimeters and

oscilloscopes have large impedance (1 MΩ or higher), and power supplies have small R_S resistance (less than 1 Ω), so the loading effect is negligible. However, if a voltmeter is used to measure the voltage in a circuit with large resistance, then error due to loading effects could be significant. Loading effects are illustrated in Example 2.3.

Example 2.3 Loading Effects

A sensor with an output voltage of 1 V and a series internal resistance of 1 kΩ is connected to an amplifier that has a gain of 10 V/V and an internal resistance of 5 kΩ. Determine the output of the amplifier due to this sensor input.

Solution:

If we treat the amplifier as an ideal amplifier, then the amplifier output is simply the voltage applied to the amplifier times the amplifier gain or $1 \times 10 = 10$ V.

However, the amplifier internal resistance acts as a voltage divider with the sensor internal resistance (a circuit similar to that shown in Figure 2.20). The voltage drop across the amplifier internal resistance is:

$$V_{\text{in}} = \frac{R_A}{R_A + R_S} V_S = \frac{5}{5 + 1} 1 = 0.83 \text{ V}$$

$$\text{Thus, } V_{\text{out}} = \text{Gain} \times V_{\text{in}} = 8.3 \text{ V}$$

Hence, due to overloading, the amplifier output deviates from the ideal output by more than 16%. Obviously, this error can be minimized by using an amplifier with high input impedance. Operational amplifiers are such type of amplifiers. Note that if the ratio of the amplifier impedance to the sensor impedance is more than 100:1, then the error due to loading effects is less than 1%.

When signals are transmitted between devices that are interfaced together, it is important to ensure that the impedances of the different devices are properly matched. If the impedances are not matched, then a high-impedance input device can reflect back some of the input signal contents that are produced by the low-impedance output device. As an example, a function generator is a low-impedance device that has a 50 Ω output impedance. If a function generator needs to be interfaced to a high-impedance device, then we can match the output impedance of the function generator by inserting a 50 Ω resistor in parallel with the high-impedance device. This is shown in Figure 2.21. The effective resistance of the inserted resistor and the high-impedance circuit is almost equal to the output impedance of the function generator. **Impedance matching** is important in many applications including the transmission of audio signals between the audio amplifier and the speakers, and in transmitting signals from ultrasonic transducers through cables.

In many cases, we also would like to deliver the maximum power to a circuit from a supply source. To achieve this, the impedances of the supply circuit and the

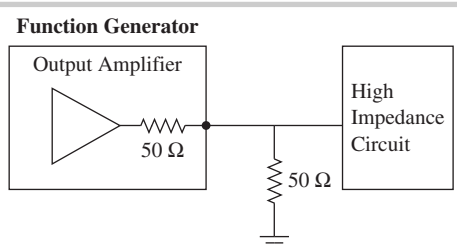


Figure 2.21

Signal connection for impedance matching

load circuit need to be matched. As an illustration, the power delivered to a resistive load with a resistance R_L from a power supply with an output voltage of V_O, and an output impedance R_S is given by the following expression (see Figure 2.20 for a similar circuit).

$$\text{Power} = \frac{V_L^2}{R_L} = \frac{R_L^2}{(R_L + R_S)^2} V_0^2 \times \frac{1}{R_L} = \frac{R_L}{(R_L + R_S)^2} V_0^2 \tag{2.27}$$

It can be easily shown that, if Equation (2.27) was differentiated with respect to R_L and set equal to zero, then the maximum power is obtained when

$$R_L = R_S \tag{2.28}$$

2.7 AC Signals

While DC voltages are common in battery-powered devices and laboratory setups, AC voltages are used in power transmission and operation of industrial and residential equipment such as compressors and kitchen appliances. AC voltage signals have the advantage that they are more efficient to transmit over long distances. When an AC voltage signal, such as a sinusoidal voltage signal, is applied to a circuit, the voltage in the circuit will also be sinusoidal with a frequency the same as the applied frequency. Two sinusoidal voltage signals are shown in Figure 2.22. The solid signal is defined by

$$V = V_0 \sin(\omega t + \theta) \tag{2.29}$$

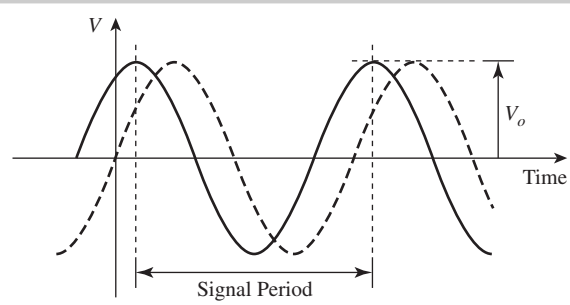
while the dashed signal is defined by

$$V = V_0 \sin(\omega t) \tag{2.30}$$

where V_O is the amplitude of the signal, and ω is the angular frequency in units of radians/second. Theta (θ) is defined as the phase angle and is a measure of the lead/lag in the signal. A positive phase angle (such as that for the solid signal) means that the signal is ahead or leads the dashed signal. The lead time is given by (θ/ω).

Figure 2.22

Two sinusoidal voltage signals



Note that the circular frequency ω in radians/second (rad/s) and the cycle frequency f in Hertz (Hz) are related by $\omega = 2\pi f$, and that the cycle frequency is the inverse of the signal period T.

For AC circuits, the magnitude of the voltage or current is specified by using the amplitude, the peak-to-peak value, or the root mean square (RMS) value. The **RMS voltage and current** are defined by

$$V_{\text{RMS}} = \sqrt{1/T \int_0^T v^2 \, dt} \tag{2.31}$$

and

$$I_{RMS} = \sqrt{1/T \int_0^T i^2 \, dt} \tag{2.32}$$

If the signal is sinusoidal, then the RMS voltage is simply equal to 0.707 V_O, and the RMS current is 0.707 I_O, where V_O and I_O are the amplitude of the sinusoidal voltage and current signals, respectively. Note that multimeters measure the RMS and not the amplitude value when they measure AC voltages and currents. Note also that 110 or 220 V supply is the RMS value of the voltage signal.

In resistive networks, both the current and the voltage will have the same phase angle. However, in circuits that have capacitive and inductive elements, the voltage and current will be out of phase. This is due to the fact that the impedance of these elements has an imaginary or j-component. In an inductor, the voltage will lead the current by 90° (the reactance is positive for an inductor), while in a capacitor; the voltage lags the current by 90° (the reactance is negative for a capacitor).

2.8 Power in Circuits

Power is defined as the rate of doing work. Power is an important specification for electrical components as it defines the rated capability for these components. In electrical circuits, the instantaneous power in an element at any point of time is defined as the product of the voltage and current through that element or

$$P(t) = V(t)\, I(t) \tag{2.33}$$

If the voltage and current do not change with time as in DC circuits, then the instantaneous power and average power are the same. For a resistor in a DC-circuit, the power can also be written as

$$P(t) = P_{avg} = VI = \frac{V^2}{R} = I^2 R \tag{2.34}$$

However, in AC circuits the voltage and current vary with time, and we need to distinguish between instantaneous power and average power. For an AC circuit with voltage $V(t) = V_O \sin(\omega t + \theta_V)$ and current $I(t) = I_O \sin(\omega t + \theta_I)$, the **instantaneous power** is given as

$$P(t) = V_O\, I_O \sin(\omega t + \theta_V) \sin(\omega t + \theta_I) \tag{2.35}$$

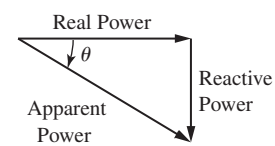
And the **average power** over one period is obtained as

$$P_{avg} = V_{RMS}\, I_{RMS} \cos(\theta) = \frac{V_O}{\sqrt{2}} \frac{I_O}{\sqrt{2}} \cos(\theta) \tag{2.36}$$

where θ is the difference between the voltage and current phase angles, i.e., $(\theta = \theta_V - \theta_I)$. The term $\cos(\theta)$ in Equation (2.36) is called the **power factor** and is a measure of the presence of reactive components (capacitors and inductors in the circuit). For a purely resistive network, the power factor is 1, and for a purely inductive network, the power factor is 0. Power factor is also defined as the ratio of the real (or useful) power to the apparent power (see Figure 2.23). A component of the apparent power is the reactive (or nonworking) power that is exchanged in capacitive and inductive components. Power factor is an important specification for devices with reactive components (such as AC-powered motors), as it defines how much of the supplied power is converted to real or useful power. Power supply

Figure 2.23

Real and apparent power



companies charge industrial and commercial customers fees for operating devices with low-power factor, since a low-power factor means that these devices draw larger currents that result in bigger power distribution lines. Example 2.4 illustrates the computation of power in AC circuits.

Example 2.4 Power in AC circuits

A load with an impedance of 500 + 600 jΩ is connected to a 110 V, 60 Hz source. Determine the power factor and the power absorbed by the load.

Solution:

The power factor can be determined from the angle that the load reactance makes with the load resistance. Using Equation (2.23), θ is given by

$$\theta = \tan^{-1}(X/R) = \tan^{-1}(600/500) = 50.2^{\circ}$$

But the power factor is cos(θ) or cos(50.2°) = 0.640

From Equation (2.15), the current through the load is given by

$$I = \frac{V}{Z} = \frac{110}{\sqrt{500^2 + 600^2}} = 0.141 \text{ A}$$

Using Equation (2.36), the absorbed or real power is

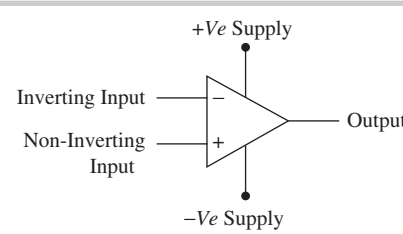
$$\text{Power} = V_{RMS}\, I_{RMS} \cos(\theta) = 110 \times 0.141 \times 0.640 = 9.93 \text{ W}$$

2.9 Operational Amplifiers

Operational amplifiers (op-amps) are analog circuit components that require power to operate. They are widely used in amplification and signal-conditioning circuits. The symbol for an op-amp is shown in Figure 2.24. The symbol is a triangle, with two leads drawn on one side of the triangle, and the third lead is drawn at the apex opposite to that side. One lead is defined as the inverting input (−), the other lead is defined as the non-inverting input (+), and the third lead is the output. The voltages at these two inputs and at the op-amp output are referenced to the ground. Figure 2.24 also shows the connections for the positive and negative supply voltages, although these connections are normally omitted when an op-amp is drawn in a circuit. The supply voltage is typically ±15 V. There are two other connections to the op-amp (called the balance or null offset) that permit adjustment of the op-amp output, but they are typically not shown.

Figure 2.24

Symbol and connections for an op-amp



Commercially, op-amps are available in a variety of forms. A common form is the single op-amp in the form of an 8-pin integrated circuit (IC), an example of which is the LM741 chip from National Semiconductor. The pin-layout of this chip is shown in Figure 2.25(a). Note that there is no connection to pin 8, and the positive and negative supply voltages are connected at pins 7 and 4, respectively. Another form is the dual op-amps on a single 8-pin package, and the pin layout for this form is different than that of single op-amp IC. Many vendors manufacture op-amp ICs, and they are available in other chip numbers such as the LF411 chip that is also available from National Semiconductor. An op-amp is constructed from a number of components including transistors, diodes, capacitors, and resistors.

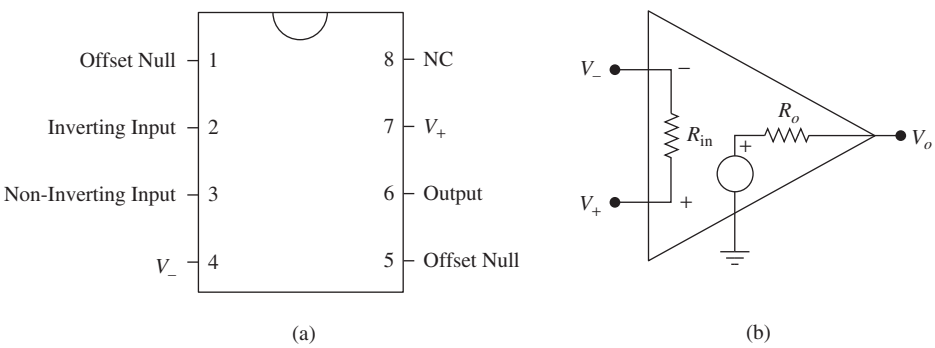


Figure 2.25

(a) Pin layout for the LM741 and (b) model of an ideal op-amp

An **ideal op-amp** can be modeled as shown in Figure 2.25(b). The inputs to the op-amp can be thought to be connected internally by a high-impedance resistor R_{in}. The value of this resistance is high enough (more than 1 MΩ), such that for ideal behavior, we can assume that no current flows between the V_- and V_+ input terminals. The output of the op-amp is modeled as a voltage source connected to a low impedance resistor R_o (less than 100 Ω) in series. The voltage output is proportional to the difference between the input voltages, i.e.,

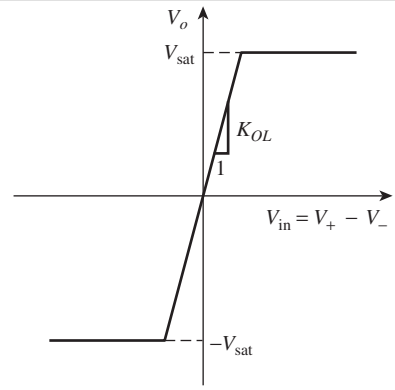
$$V_o = K_{OL} \times (V_+ - V_-) \tag{2.37}$$

where K_{OL} is the open-loop gain of the op-amp. The open-loop gain of the op-amp is usually very high (10^5 to 10^6), so a very small voltage difference between the two inputs results in a saturation of the output. For example, if the gain is 10^6, and the saturation voltage is 10 V, then the op-amp will saturate if the voltage difference between the input leads exceeds 10 μV. Since the op-amp output is finite, but the op-amp has a very large gain, we assume that $V_+ = V_-$. The assumption that $V_+ = V_-$ along with the assumption that no current flows into the input terminals are the two basic rules that are used to analyze ideal op-amp circuits.

It should be noted that the **saturation voltage** of an op-amp is a function of the supply voltage for the op-amp and it is slightly smaller than it. For example, at supply voltage of ±15 V, the saturation voltage is about ±13 V. The open-loop input output relationship for an op-amp is shown in Figure 2.26. In most cases, however, op-amps are not used in open-loop configuration but are used with a feedback loop between the output voltage lead and the inverting input lead. The closed-loop gain is much smaller than the open-loop gain, but the feedback provides more stable operating characteristics.

Figure 2.26

Open-loop input/output relationship for an op-amp

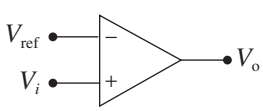


Note that an op-amp gives a zero output if the two input voltages are the same. This is called the common-mode rejection property of the op-amp. In reality, the output will not be exactly zero, but one can use the null offset terminals on the op-amp to adjust this output. Op-amps have good frequency response characteristics, and their bandwidth exceeds 1 MHz.

Op-amps can perform various operations such as comparison, amplification, inversion, summation, integration, differentiation, or filtering. The particular operation depends on how the op-amp is wired and what external components are connected to the op-amp. We will discuss below some of these operations assuming ideal behavior. In most cases, the real-behavior closely follows the ideal behavior.

Figure 2.27

Comparator op-amp circuit



2.9.1 Comparator Op-Amp

A comparator is used to compare two voltage signals, and switch the output to $+V_{sat}$ if one of the signals is larger than the other, and to $-V_{sat}$ otherwise, where V_{sat} is the saturated output of the op-amp. The circuit for an op-amp operating as a comparator is shown in Figure 2.27. Here the op-amp is operating in open-loop, which means there is no feedback from the op-amp output to the input. The input voltage V_i is connected to the non-inverting input (+), and the reference voltage V_{ref} is connected to the inverting input (−). The comparator output V_o is then

$$V_o = \begin{cases} V_{sat}, V_i > V_{ref} \\ -V_{sat}, V_i < V_{ref} \end{cases} \quad (2.38)$$

A comparator can be used, as an example, in situations where it is needed to set an output on if a sensor input exceeds a certain value. Microcontrollers such as the PIC16F690 (discussed in Chapter 4) have a built-in comparator feature.

2.9.2 Inverting Op-Amp

The inverting op-amp circuit is shown in Figure 2.28 which has a feedback loop between the op-amp output and the inverting input (−). An input voltage V_i is applied to the inverting input through a resistor R_1, and the non-inverting input (+) is grounded. Since the non-inverting input is connected to ground,

$$V_- = V_+ = 0 \quad (2.39)$$

The current I_1 is equal to I_2 because virtually no current flows between the inverting and the non-inverting inputs. The current I_1 is equal to

$$I_1 = \frac{V_i - V_-}{R_1} = \frac{V_i}{R_1} \quad (2.40)$$

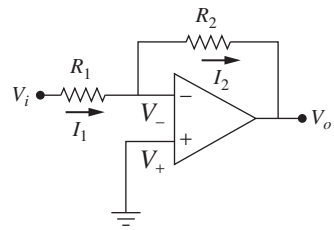


Figure 2.28
Inverting op-amp circuit

and the current I_2 is equal to

$$I_2 = \frac{V_- - V_o}{R_2} = -\frac{V_o}{R_2} \tag{2.41}$$

Equating I_1 to I_2, and solving for the op-amp output V_o gives

$$V_o = -\frac{R_2}{R_1} V_i \tag{2.42}$$

Thus in this circuit the op-amp inverts the input voltage and amplifies it by a factor equal to the ratio of the resistance of R_2 to R_1. An application of this circuit is to perform signal inversion where the output will have a 180° phase shift with the input. Example 2.5 illustrates the use of the inverting op-amp circuit.

Example 2.5 Summing Circuit

Draw a circuit that shows how op-amps can be used to perform summation of three analog voltage signals.

Solution:

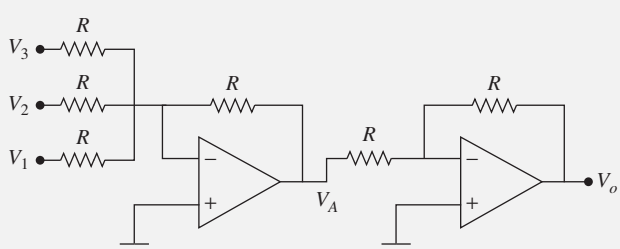


Figure 2.29

The circuit that performs this operation is shown in Figure 2.29. It basically consists of two cascaded op-amps each wired as an inverting amplifier. The three voltage signals are connected to the left op-amp. Note that the sum of the currents through the three resistors that are connected to the input voltages V_1, V_2, and V_3 is the same as the current that goes through the resistor in the feedback loop in the left op-amp circuit:

$$\frac{V_1 - V_-}{R} + \frac{V_2 - V_-}{R} + \frac{V_3 - V_-}{R} = \frac{V_- - V_A}{R} \tag{1}$$

Since $V_- = 0$, and cancelling R from each term in Equation (1), this gives

$$V_A = -(V_1 + V_2 + V_3) \tag{2}$$

From the second op-amp circuit, we get

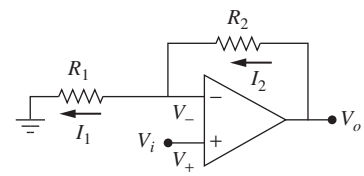
$$V_o = -V_A = V_1 + V_2 + V_3 \tag{3}$$

2.9.3 Non-Inverting Op-Amp

The non-inverting op-amp circuit is shown in Figure 2.30. Here the non-inverting input (+) is connected to an input voltage V_i, and the inverting input (−) is connected to ground through a resistor R_1. There is also a feedback loop between the op-amp output and the inverting input.

Figure 2.30

Non-Inverting op-amp circuit



The voltage V_+ is equal to V_- and is also equal to V_i in this case. But the voltage at the inverting input is also given by

$$V_- = \frac{R_1}{R_1 + R_2} V_o \tag{2.43}$$

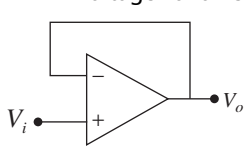
since R_1 and R_2 act as a voltage-dividing circuit between V_o and ground. Thus, the output V_o of the op-amp is given by

$$V_o = \frac{R_1 + R_2}{R_1} V_i = \left(1 + \frac{R_2}{R_1}\right) V_i \tag{2.44}$$

Notice how the gain of the op-amp in this case is always greater than 1. Now if we let R_2 to be zero and R_1 to be infinite, this gives the circuit shown in Figure 2.31. This circuit is known as a **voltage follower** or buffer, and $V_o = V_i$ in this case. Because the op-amp has a low output impedance (about 75 Ω), and a high input impedance (about 2 MΩ), the voltage follower circuit can be used in a variety of ways to reduce loading effects. The output of a voltage source can be connected to the buffer input to isolate the source from the rest of the circuit, or the buffer output can be connected to a high-impedance circuit.

Figure 2.31

Voltage follower



Note that in both the inverting and the non-inverting op-amp circuits shown above, the feedback between the output voltage and the inverting input is known as negative feedback. Negative feedback results in a linear relationship between the output and input voltages. If the feedback loop was between the output voltage and the non-inverting input, then the output-input relationship is nonlinear. The nonlinearity is a hysteresis where the input has to change by a certain amount before the output changes state. Non-linear op-amp circuits are utilized in the design of **Schmitt triggers**, which are IC circuits that are used for converting slowly changing or noisy analog signals into two-level digital signals (see Section 7.4.1 which discusses their use in the wiring circuit for a Hall-effect proximity sensor). The symbol for a standard (non-inverting) Schmitt trigger and the input and output voltages from a Schmitt trigger are shown in Figure 2.32. Note how the output of the Schmitt trigger goes to $V_{\max}$ when the input signal voltage exceeds the positive going threshold voltage (V_{T+}). The output signal stays at $V_{\max}$ until the input signal drops below the negative going threshold voltage (V_{T-}), at which point the output goes to $V_{\min}$. In Figure 2.32, $V_{\max}$ and $V_{\min}$ are the positive (typically 5 VDC) and the negative (typically 0 VDC) supply voltage, respectively, for the Schmitt trigger device. The 74HC7014 IC has six non-inverting Schmitt triggers with $V_{T+} = 3.1$ V and $V_{T-} = 2.9$ V when used with a 5-VDC supply voltage.

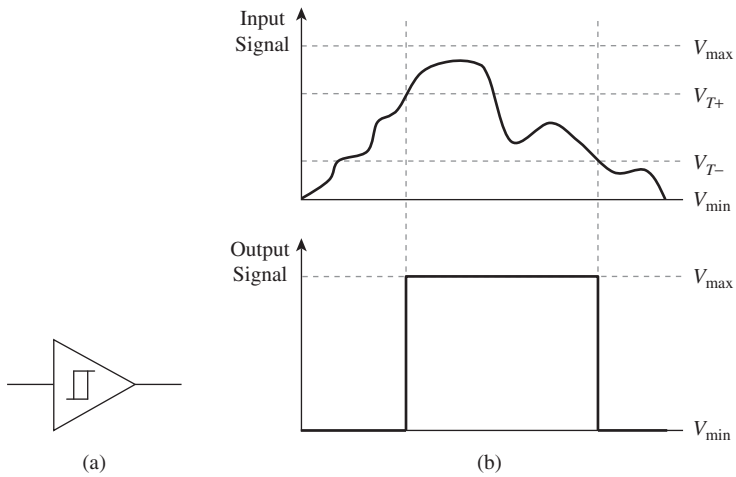


Figure 2.32

Schmitt trigger: (a) symbol and (b) input/output relationship

2.9.4 Differential Op-Amp

An op-amp circuit with two voltages (V_1 and V_2) applied to its inputs is shown in Figure 2.33. Two inputs (differential input) are used to reduce the circuit sensitivity to noise, since any noise applied to the circuit will be most probably the same on each of the inputs.

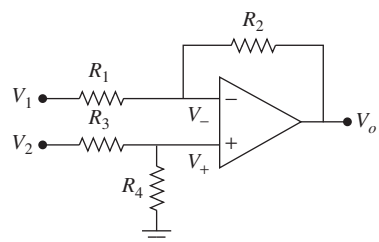


Figure 2.33

Differential input op-amp circuit

For this circuit, the current through the R_1 and the R_2 resistors is the same, since no current goes through the inverting input. This current is given by

$$I_{R_1} = \frac{V_1 - V_-}{R_1} = I_{R_2} = \frac{V_- - V_o}{R_2} \tag{2.45}$$

The voltage at the non-inverting input V_+ is given by

$$V_+ = \frac{R_4}{R_3 + R_4} V_2 \tag{2.46}$$

But $V_+ = V_-$. Substituting the expression for V_+ in Equation (2.45) and solving for V_o gives

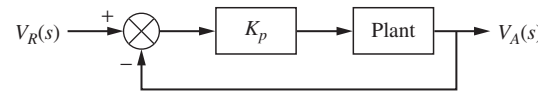
$$V_o = \frac{R_4}{R_3 + R_4}\left(1 + \frac{R_2}{R_1}\right)V_2 - \frac{R_2}{R_1} V_1 \tag{2.47}$$

If $R_3 = R_1$ and $R_4 = R_2$, this expression simplifies to

$$V_o = \frac{R_2}{R_1}(V_2 - V_1) \tag{2.48}$$

and shows that the output of this op-amp circuit is proportional to the voltage difference between the inputs V_2 and V_1. A differential amplifier circuit can be used,

Figure 2.34
Proportional control feedback loop



for example, to implement an analog proportional control feedback loop (see Figure 2.34). Feedback control is covered in Chapter 9.

If the reference signal V_R is the V_2 voltage, the actual or measured signal V_A is the V_1 voltage, and the ratio R_2/R_1 is the proportional gain K_p, then the output of the differential amplifier will be

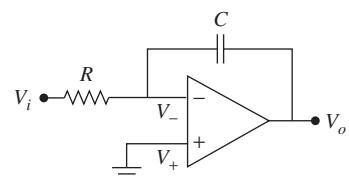
$$V_o = K_P(V_R - V_A) \tag{2.49}$$

Another application of the differential amplifier circuit is to amplify the difference between the voltage outputs from the arms of a Wheatstone bridge used to measure strain (see Section 7.10.3).

2.9.5 Integrating Op-Amp

The circuit for an integrating op-amp is shown in Figure 2.35, which has a capacitor C in the feedback loop.

Figure 2.35
Integrating op-amp circuit



From Equation (2.3), the current through a capacitor is given by

$$I_C = C\frac{dV}{dt} \tag{2.50}$$

For this capacitor, $V = V_- - V_o = -V_o$, since $V_- = 0$. But the current through this capacitor is the same as the current that passes through the resistor R, since no current flows through V_-. This current is given by

$$I_R = \frac{V_i - V_-}{R} = \frac{V_i}{R} \tag{2.51}$$

Thus,

$$\frac{dV_o}{dt} = -\frac{V_i}{RC} \tag{2.52}$$

Integrating Equation (2.52) from time $t = 0$ to time $t = t_1$ gives

$$V_o(t) = -\frac{1}{RC}\int_0^{t_1} V_i(t)\,dt + V_o(0) \tag{2.53}$$

where $V_o(0)$ is the initial condition for the capacitor voltage. Thus, in this circuit, the op-amp produces an inverted output of the integral of the applied input voltage. Note that if the capacitor and the resistor were interchanged in this circuit, the op-amp will act as a **differentiator** of the input signal. The op-amp output in this case will be

$$V_o = -RC\frac{dV_i(t)}{dt} \tag{2.54}$$

Note that any noise in the input signal will be amplified by differentiation.

2.9.6 Power Amplifier

A standard op-amp (such as the LM741) has a current output rating of about 25 mA. This is not sufficient to meet the current needs of driving loads (such as valve actuators, servo motors, and audio amplifiers). Commercial op-amps with a higher current output rating are available. These op-amps are called power op-amps, an example of which is the OPA547 chip from Texas Instruments. The OP547 can provide a continuous output current of 500 mA with the ability to control the output current limit. Power op-amps can be conveniently used to interface a digital-to-analog (D/A) converter (see Chapter 5) that needs to drive a DC motor. Table 2.4 gives a sampling of power op-amp devices.

Table 2.4
Power op-amp devices

Device	Power Supply Range	Continuous Output Current (A)	Peak Current (A)	Slew Rate (V/μs)	Adjustable Current Limit (Y/N)
OPA547	+8 V to +60 V ±4 V to ±30 V	0.5	0.75	6	Y
LM675	±8 V to ±30 V	3	4	8	N
OPA541	+20 V to +80 V ±10 V to ±40 V	5	10	10	Y
OPA549	±4 V to ±30 V	8	10	9	Y

In Table 2.4, the '*Power Supply Range*' column defines the allowable voltage levels that can be applied to the positive and negative supply inputs of the op-amp. The power supply range affects the op-amp **output voltage swing**, which is the maximum voltage that the op-amp can produce without saturation for a given load. Note that the output voltage swing is proportional to the power supply range. The '*Slew Rate*' column defines the rate at which the op-amp output voltage will change when the op-amp gain is set to unity. Several of the power op-amps listed in the Table 2.4 allow adjustment of the maximum output current of the op-amp. Due to their large output current, power op-amps are available in packages with a built-in copper tab to allow easy mounting to a heat sink for good thermal performance.

To show further the application of op-amps, Example 2.6 illustrates the use of op-amps in analog feedback control loops.

Example 2.6 PI Analog Feedback Loop

Illustrate how op-amps can be used to implement an analog proportional integral (PI) feedback control loop.

Solution:

A PI controller has the following relationship between the control output $V_o(t)$ and the error signal $e(t)$:

$$V_o(t) = K_P\, e(t) + K_I \int e(t)\, dt \tag{1}$$

where the error signal is defined as

$$e(t) = V_{\text{ref}}(t) - V_A(t) \tag{2}$$

with $V_{\text{ref}}(t)$ as the reference or desired value and $V_A(t)$ as the actual or measured value. The PI controller can be implemented as the cascade of three op-amps circuits (Figure 2.36). The first circuit is a differential op-amp circuit to compute the

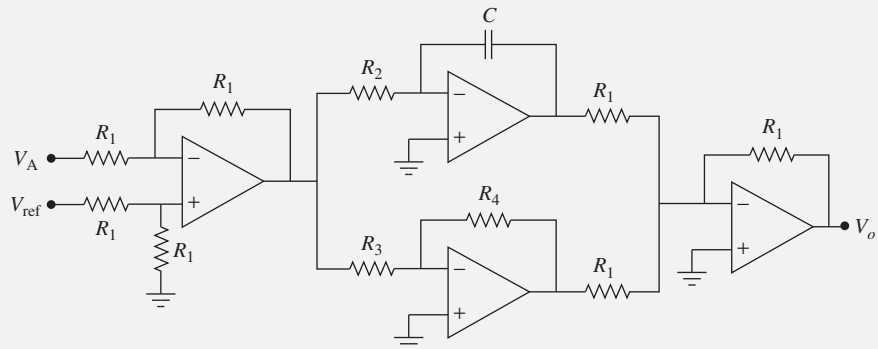


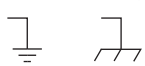
Figure 2.36

error signal. The second circuit, which actually consists of two op-amps, implements the *P* and *I* actions. The last stage sums and inverts the outputs from the *P* and *I* action circuits. Note that the K_P gain is the ratio of R_4 to R_3 and the K_I gain is equal to $1/(R_2C)$. To allow for variable gains, a potentiometer can be used to replace the R_4 and R_2 resistors.

2.10 GROUNDING

Figure 2.37

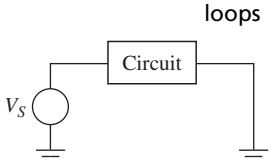
(a) Ground return symbol and (b) Chassis return symbol



When we talk about voltage or potential difference, we always refer to the value of one voltage level with respect to another level. **Ground voltage** or zero voltage is commonly used as a reference. It is indicated in circuit diagrams by the symbol shown in Figure 2.37(a). Using that voltage as reference, one can measure the other voltages in the circuit. Technically, a true ground voltage refers to the earth ground voltage which is obtained by a connecting a wire to a metal pole that is inserted into the earth's surface. However, in many circuits, a ground symbol does not mean connection to an earth ground but to a current return path to the negative terminal of the power supply. Another type of ground reference is called the **Chassis return**. This is indicated by the symbol shown in Figure 2.37(b). A chassis return refers to the connection between a device housing or casing and the earth line in the power cord.

Figure 2.38

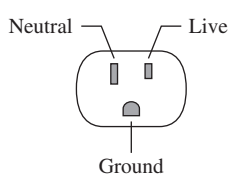
Illustration of a wiring that leads to ground loops



It is important in circuits to have a common ground to avoid the problem that arises from ground loops. **Ground loops** form when there is more than one path to connect a circuit or system to ground. An example of a ground loop wiring is shown in Figure 2.38. Ground loops lead to voltage differences between the two ground points, which results in noise in the circuit. A way to eliminate ground loops is to connect all of the return paths in a circuit to a common ground point. If this is not practically possible, then all of the return paths should be connected to a common ground bus which is itself connected to ground. In addition, if the circuit has analog and digital elements, then the analog ground and digital ground should be connected at one point.

Figure 2.39

Polarized wall socket



In discussing grounding, it is important to understand how this is done in AC circuits. In AC circuits (such as those in a typical home), the electrical wall socket outlet has three slots (see Figure 2.39). One slot is called live or hot, the other is neutral or return, and the third is earth or ground. The live slot carries the alternating current to the load (such as a small appliance) that is plugged into the wall socket. The neutral or return slot provides a return path of the current back to the source. The ground line, which is connected to earth ground, normally does not carry any current, but is provided as a safety feature in case of an electrical fault within the connected equipment. The metal case or covering of the electrical

device, where humans typically contact, is connected to the ground line to provide a path for current in fault situations. In the USA and many countries, polarized plugs are used in which there is only one way of connecting the plug to the wall socket to prevent the interchange of the live and neutral lines. This is done as a safety feature to prevent, for example, a switch to open a circuit using the neutral line. While such an arrangement interrupts the current flow through the device, the live line is still connected to the device, which would cause a safety hazard if one services the equipment while the plug is still connected.

2.11 Solenoids and Relays

2.11.1 Solenoids

A solenoid is an example of an inductive element that is widely used. It is commonly used for on-off applications such as locking or triggering. Applications include the switching of electromechanical relays, door locks, ratcheting devices, and gate diverters.

The solenoid is an electrically actuated mechanical device that has two states: retracted and extended. It is typically constructed from a movable armature core that moves inside a stationary iron core. A typical layout is shown in Figure 2.40. When the armature coil is energized with current, it moves out to increase the flux linkage by closing the air gap between the cores. The moveable core is spring loaded and will retract when the current is switched off.

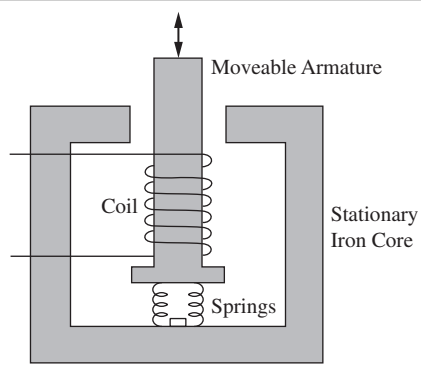


Figure 2.40
Schematic of a solenoid

Commercially, solenoids are available in two forms: linear and rotary. The linear form can be either pull-type or push-type. In a **pull-type solenoid**, the force is directed toward the solenoid body when the solenoid is energized, while in a **push-type solenoid**, the force is directed away from the solenoid body when the solenoid is energized. Linear solenoids have a stroke that is typically less than 1 inch, while rotary solenoids have a typical stroke of about 45°.

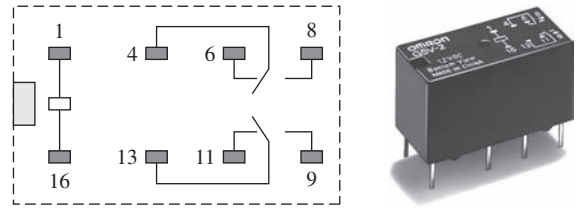
2.11.2 Electromechanical Relays

Many computer interfacing applications use electromechanical relays which are electrically actuated switches that use a solenoid to make or break the mechanical contact between electrical leads. The connection diagram of a typical small power relay is shown in Figure 2.41. When the coil circuit is closed (terminals 1 and 16), the solenoid will move the two poles that contact terminals 6 and 11 to contact terminals 8 and 9, respectively. The switch configuration in this relay is an example of

Figure 2.41

G5V-2, a typical small power relay

(Courtesy of Omron Corporation)



the double-pole, double-throw switch configuration that was previously discussed. Some of the important characteristics of this relay are listed in Table 2.5. This relay can be used to switch up to 60 W (or 62.5 VA) using a coil (solenoid) that requires 5 VDC at 100 mA to operate. This current input value is beyond the current output limits of digital output ports (discussed in Chapters 4 and 5), so a current amplifying component (such as a transistor) is normally used to interface the digital output port to the coil terminals of the relay. The advantage of a relay is that the input circuit is electrically insulated from the output circuit. So any noise-induced voltages in the output circuit have a minimal impact on the input circuit. The second advantage is that a small coil current can be used to switch a much larger load current.

Table 2.5

Characteristics of the G5V-2 OMRON relay

Coil Rating	Rated voltage	5 VDC
	Rated current	100 mA
	Coil resistance	50 Ω
Contact Rating	Rated load	0.5 A at 125 VAC, 2 A at 30 VDC
	Maximum switching voltage	125 VAC, 125 VDC
	Maximum switching current	2 A
Operating Characteristics	Operate time	7 ms maximum
	Release time	3 ms maximum
	Maximum operating frequency	Mechanical: 36,000 operations/hr Electrical: 18,000 operations/hr

One disadvantage of electromechanical relays is their relatively long switching time. For the previous relay, the maximum operate-release cycle time is 10 ms, and the maximum mechanical switching frequency is 10 Hz. This is in contrast to solid-state transistors (discussed in Chapter 3), which have nanoseconds switching time.

2.12 Chapter Summary

This chapter discussed analog circuits and components. The chapter started by discussing basic circuit elements (such as resistors, capacitors, and inductors) and the laws for their combination in series or parallel form. It then discussed the various configurations of mechanical switches, including push-button and toggle switches. Circuit analysis was then covered. Kirchhoff's voltage law (KVL) and Kirchhoff's current law (KCL) are the two basic laws that are used to perform circuit analysis. KVL states that the algebraic sum of the voltage drops and rises around any closed path in a circuit is zero, while KCL states that the sum of the current into a node is zero. To simplify certain circuits, equivalent circuits are used. The equivalent circuit can be in one of two forms: the Thevenin equivalent circuit or the Norton equivalent circuit. The concept of impedance, which is a generalization of the concept of

resistance, was then covered. Impedance concepts are useful in analyzing loading effects that occur when measurement devices are connected to circuits. Alternating current circuits were also discussed, including computing the power in these circuits. Operational amplifiers or op-amps are analog circuit components that require power to operate. Op-amps can perform various operations (such as comparison, amplification, inversion, summation, integration, differentiation, or filtering). The particular operation depends on how the op-amp is wired and what external components are connected to the op-amp. Several op-amps circuits were discussed. The concepts of ground loops and proper grounding techniques were also presented. The last section discussed solenoids and their use in electromechanical relays, which are used as interface elements.

Questions

2.1 Define what is meant by an analog circuit.

2.2 Name the two laws that are used to analyze electrical circuits.

2.3 List the different types of toggle switches.

2.4 Define impedance.

2.5 What impedance characteristic is desirable in measuring devices, and why?

2.6 What device has a very large impedance at low frequencies?

2.7 What characteristic of an op-amp make it useful to use it as an interface?

2.8 List the two rules that are used to analyze ideal op-amp circuits.

2.9 List the different types of op-amp circuits.

2.10 What type of op-amp circuit is used in the implementation of an analog proportional control feedback loop?

2.11 Can the output voltage of an op-amp circuit exceed the supply voltage?

2.12 List an advantage of AC signals.

2.13 Name one way to avoid a ground loop.

2.14 Define real power of an AC circuit.

2.15 List several applications of solenoids.

2.16 What is a relay?

Problems

P2.1 Illustrate how an SPDT switch can be used for wiring the low/high beam for car headlights using the car battery as the voltage source.

P2.2 Identify three household/consumer applications where push-button switches are used, and identify whether the switch is NO or NC type.

P2.3 Draw the wiring circuit for a light bulb that can be turned on/off from three switch locations.

P2.4 Draw the switching circuit for a three-position light bulb. A three-position light bulb has two filaments. At the low switch position, the low-intensity filament is turned on. At the medium setting, the medium-intensity filament is turned on. At the high position, both filaments are turned on.

P2.5 Determine the unknown currents through the resistor network shown in Figure P2.5.

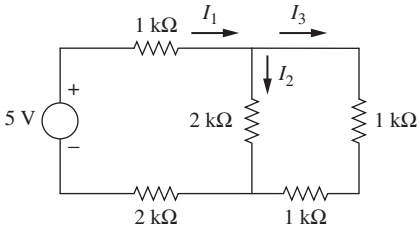


Figure P2.5

P2.6 Determine the unknown currents through the resistor network shown in Figure P2.6.

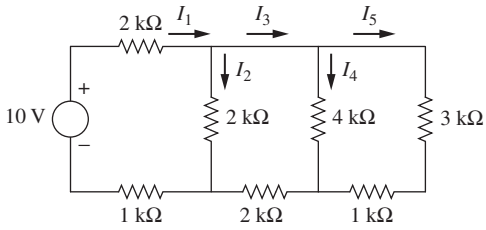


Figure P2.6

P2.7 The terminal voltage of a power supply is 24 VDC before a load is applied. When a 100 Ω resistor is connected to the power supply, the voltage drops to 23 V. What is the internal resistance of the supply source?

P2.8 Determine the Thevenin equivalent circuit at the nodes *a* and *b* for each of the two circuits shown in Figure P2.8.

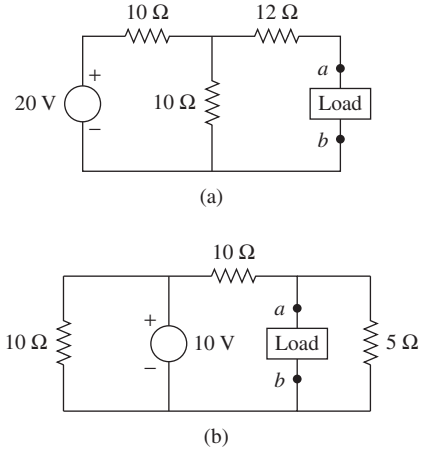


Figure P2.8

P2.9 Determine the impedance for the following components at 60 Hz. What is the total impedance of these components if they were connected in series?

a. 1000 Ω resistor

b. 500 mH inductor

c. 1 μF capacitor

P2.10 What is the power factor if the three components in Problem 2.9 were connected in series?

P2.11 A load made up of a resistor and a capacitor connected in series draws 0.2 A from a 60-Hz, 110-V source with a 0.8 power factor. Determine the resistance and reactance of the load.

P2.12 Plot the output of the circuits shown in Figure P2.12 for the following input signal. Note that $R_1 = R_2 = 100$ kΩ, and $C = 10$ μF.

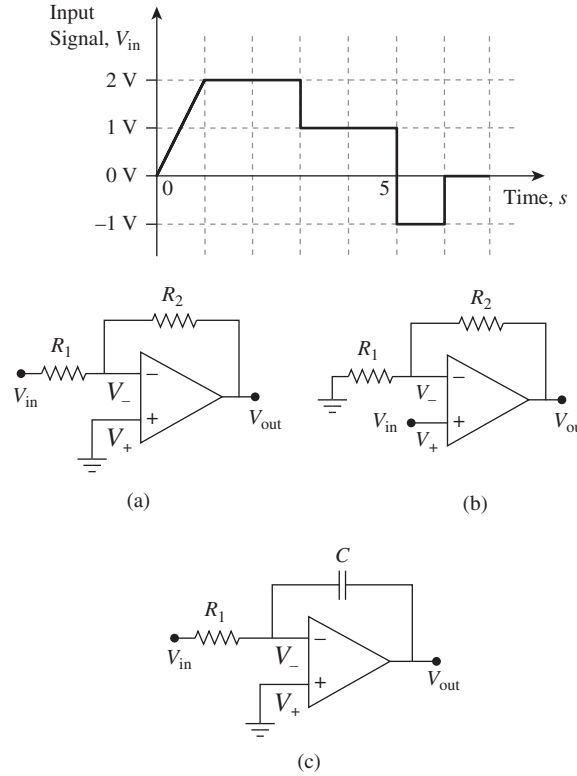


Figure P2.12

P2.13 The input signal shown in Figure P2.13 is applied to a non-inverting Schmitt trigger with

$V_{T+} = 3.1$ V and $V_{T-} = 2.9$ V. Plot the output signal if V_{max} is 5 V and V_{min} is 0 V.

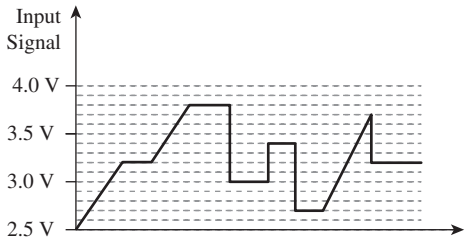


Figure P2.13

P2.14 An engineer has proposed the circuit shown in Figure P2.14 for performing closed-loop proportional control of the speed of a DC motor using a tachometer as the feedback signal. Show if this circuit will operate as proposed.

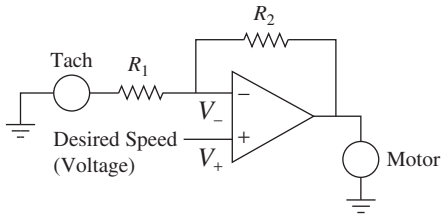


Figure P2.14

P2.15 Design an op-amp circuit to implement an analog PD controller action. Select components to give a K_p gain value of 5 and a K_d value of 0.1.

P2.16 Design an amplifier circuit that uses LM741 op-amps. The circuit should take an input voltage V_{in} and produce an output voltage V_{out} that is equal to $k\,V_{in}$ where $0 \le k \le 10$. Specify all of the components that are used in the circuit, the output voltage, and the current limits of the amplifier.

P2.17 Research and identify three household, commercial, or automotive applications that use solenoids. For each application, specify the type of solenoid used.

P2.18 Draw a circuit that uses two relays (similar to the one shown in Figure 2.41) to switch the direction of rotation of a DC motor. The circuit has three inputs, the supply voltage for the motor and two control inputs A and B, as shown in Figure P2.18. When A is 5 V and B is 0 V, the motor rotates in one direction, and when A is 0 V and B is 5 V, the motor rotates in the opposite direction.

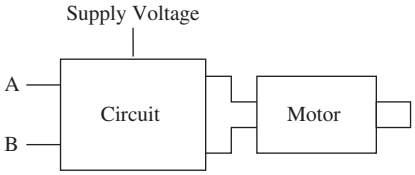
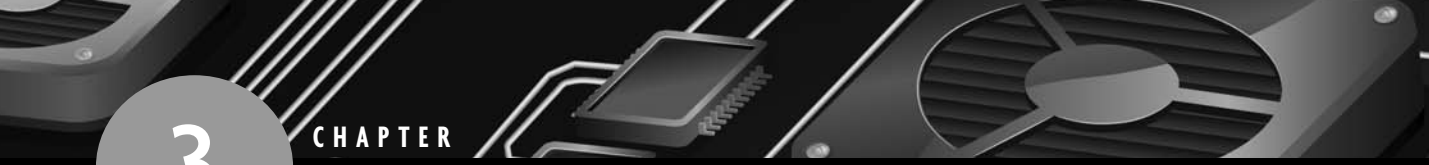


Figure P2.18

Laboratory/Programming Exercises

L/P2.1 Build the circuit shown in Figure P2.5, and measure the voltages and currents in the circuit. Compare the measured values to the computed ones.

L/P2.2 Build and test the circuit described in Problem P2.18.



CHAPTER 3

Semiconductor Electronic Devices and Digital Circuits

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- Explain the function of diodes and thyristors
- Predict the output of simple circuits involving regular and Zener diodes
- Explain the use of transistors in switching applications
- Analyze circuits containing bipolar junction and MOSFET transistors
- Analyze digital combinational logic circuits and generate a logic circuit from a truth table specification
- Generate the timing diagrams for various types of flip-flops
- Explain the properties of TTL and CMOS circuit families, their characteristics, and how to interface them
- Draw a wiring circuit for digital devices (such as timers and H-bridge drives)

3.1 INTRODUCTION

The previous chapter considered the design and analysis of analog circuits. This chapter discusses the operation of semiconductor electronic devices (such as diodes, thyristors, and transistors) that are used in many circuits and devices for switching or amplification purposes. A semiconductor is a material whose properties are in between a conductor and an insulator. Examples of naturally available semiconductor materials include silicon and germanium. For use in semiconductor electronic circuits, small quantities of other elements (such as boron and phosophorous) are added to silicon or germanium crystals to alter their properties. Semiconductor electronic devices have properties that depend on temperature, lighting conditions, or the amount and direction of voltage applied to them. A basic and important semiconductor device is the transistor, whose invention has led to the development of digital circuits in which transistors form the building blocks. An important feature of a transistor is that it can amplify an input signal. Semiconductor electronic devices (such as transistors) are commonly used as an interface in the operation of real devices (such as motors and heaters). Digital circuits are widely used in devices such as computers, wireless phones, and digital cameras. This chapter considers both combinational and sequential digital logic circuits as well as digital devices. Digital circuits form the foundation for microprocessors and microcontrollers, and the next chapter will discuss microcontrollers that give a flexible but complicated method of implementing control logic. For further reading on the topics covered in this chapter, see [7-9].

3.2 Diodes

Diodes and transistors are examples of solid-state switches. Solid-state switches are devices in which the switching action is caused by non-mechanical motion and is due to the change in the electrical characteristics of the device. A diode is a directional element that allows current to flow in one direction. The characteristics of a real diode (i.e., not ideal) are shown in Figure 3.1. Unlike a resistor or a capacitor, the diode current–voltage relationship is highly nonlinear and does not follow Ohm's law. The figure shows that, when the diode is forward biased or the anode voltage is positive with respect to the cathode, current flows in the diode in the direction of the arrow of the diode symbol. The current becomes very large when the forward-bias voltage approaches V_F, which is the diode **forward voltage**. The forward-voltage value is dependent on the material from which the diode is made. For silicon diodes (such as the 1N914 diode), the V_F voltage is about 0.6 V (at T = 300°K). When a reverse-biased voltage is applied to the diode, very little current (in the nano-ampere range—see scale in Figure 3.1) flows unless the applied voltage reaches the breakdown voltage (or V_R about 75 V for the 1N914), which causes the diode to break down. Regular diodes are not designed to operate with a voltage lower than V_R unless a Zener diode is used.

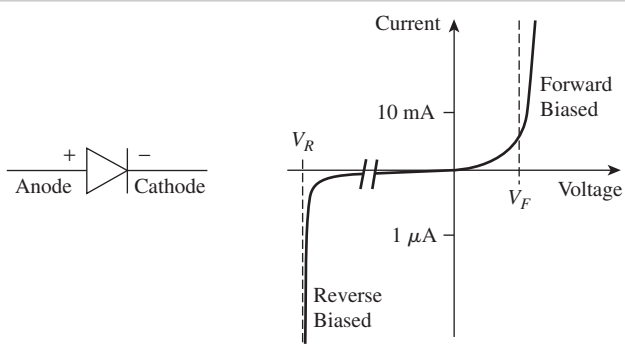


Figure 3.1

Characteristics of a diode

One common use of diodes is to change AC voltages to DC voltages, which is a process called **rectification**. Figure 3.2(b) shows the output of the circuit shown in Figure 3.2(a), which is called a half-wave rectifier. The rectification occurs only if the amplitude of the sinusoidal signal exceeds the forward voltage (V_F) value for the diode. Notice how the negative portion of the sinusoidal input voltage is eliminated and how the amplitude of the positive portion of the output signal is smaller than the input voltage.

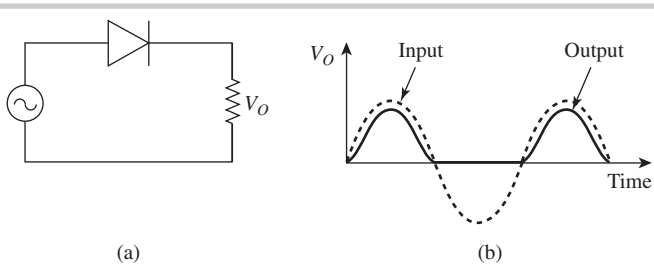
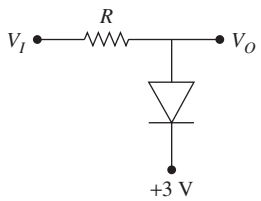


Figure 3.2

Half-wave rectification: (a) circuit and (b) output voltage

Figure 3.3

Diode clamp circuit



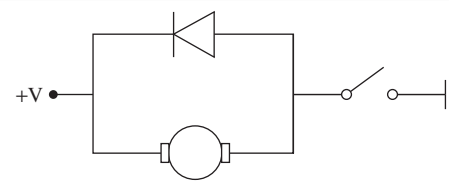
Another use of diodes is to limit the range of a signal in a circuit. This is called voltage clamping, and the diode is called a **diode clamp** when used in such a circuit. Figure 3.3 shows such a circuit.

Notice that, because a diode conducts large current only when the forward-bias voltage is greater than V_F (or about 0.6 V for 1N914 diode), the above circuit will limit the output voltage V_O to 3.6 V (0.6 V plus the voltage applied at the cathode). Any input voltage lower than that (including negative voltage above the breakdown voltage) is passed as an output. This is because, when the diode is not conducting, no current is flowing across the resistor R in the circuit of Figure 3.3, and thus, $V_O = V_I$.

A further application of diodes is to limit voltage spikes generated when switching off inductive loads (such as DC motors or relay coils). The use of the diode in this case is called a **flyback** diode. A typical wiring is shown in Figure 3.4.

Figure 3.4

Flyback diode circuit

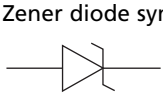


To understand the function of the flyback diode, assume first that the diode is not present. Note that when the switch is opened, the current in the motor coil starts to change. Since a coil has inductance, a large voltage of opposite polarity develops across the motor lead according to the relationship $V = L\ di/dt$. Thus, the two sides of the switch will have voltages of opposite polarity, leading to arcing and premature wear of the switch contact. If, on the other hand, a diode was added as shown in Figure 3.4, the diode provides a path for the current in the coil leading to a reduced voltage spike at the switch leads.

3.2.1 Zener Diode

Figure 3.5

Zener diode symbol

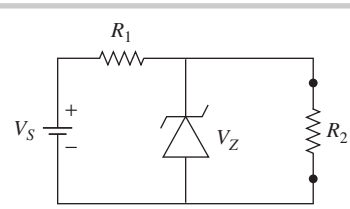


A special type of diode is called the **Zener** diode, and its symbol is shown in Figure 3.5. A Zener diode behaves like a normal diode when it is forward biased, but it can conduct current without destroying itself when the reverse-biased voltage exceeds the breakdown voltage, V_R. The breakdown voltage or Zener voltage, V_Z, can be smaller than that for a normal diode. Common low Zener voltages include 2.7, 3.0, 3.6, 3.9, 4.7, 5.1, 5.6, and 6.2 V, but Zener voltages of 20, 51, 100, and even 200 V are available (for example, the 1N5221 to 1N5281 silicon Zener diodes series from Sematech Electronics LTD).

A common use of Zener diodes is to regulate the output voltage in a circuit when the supply voltage is variable or unstable. The circuit for performing this operation is shown in Figure 3.6. Note that the output voltage of this circuit or the voltage drop across the load resistor R_2 always will be V_Z if the voltage drop across R_2 is about to exceed V_Z. Example 3.1 illustrates the use and sizing of such a diode.

Figure 3.6

Voltage regulation using a Zener diode



Example 3.1 Voltage Regulation using a Zener Diode

For the circuit shown in Figure 3.6, assume that the voltage source is an unregulated supply that varies between 10 to 12 V. Select a Zener diode and appropriate resistors to give close to a 5 V drop across the load resistor R_2 using this supply.

Solution:

Let us select a Zener diode with a breakdown voltage of 5.1 V (which is the closest to 5 V). Select R_2 to be 100 Ω. Then the current across the load resistor R_2 is 51 mA, because the potential drop across the R_2 resistor is the same as that across the Zener diode for cases when the voltage drop across R_2 is about to exceed V_Z. The current through the resistor R_1 has to be greater than the current through the load resistor, because the Zener diode will not operate unless some current flows through it, since $I_{R_1} = I_Z + I_{R_2}$. This implies that R_1 has to be less than 96.1 Ω for $V_S = 10$ V from the requirement that

$$I_{R_1} > I_{R_2}$$

or

$$\frac{V_S - V_Z}{R_1} > \frac{V_Z}{R_2}$$

If we select R_1 to be 90 Ω, then the current through R_1 will be 54.4 mA for $V_S = 10$ V and 76.7 mA for $V_S = 12$ V. Notice that the current that is not passing through the load is being dissipated through the Zener diode. For the diode not to heat up, the diode power rating must be greater than 5.1 V × (76.7 − 51) mA or 0.13 W. A 0.25 W diode will do this. A commercially available diode with such specifications is the 1N4689.

3.2.2 LED

One common form of a diode is the light-emitting diode (**LED**). These diodes emit light when forward biased, and the amount of light they emit is proportional to the current passing through the LED. They are typically encased in a colored plastic casing. An advantage of an LED over other light sources is that it takes only a few milliamps to light the diode. They also can be powered by a digital power supply (5 VDC), since the voltage drop across the LED when it is on is about 2 V. A typical LED and its symbol are shown in Figure 3.7. Note that the anode or the positive terminal is the one that has a longer lead.

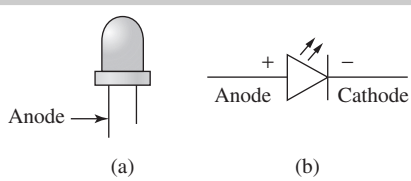
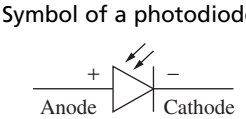


Figure 3.7
(a) LED and (b) symbol

3.2.3 Photodiode

Another form of a diode is the **photodiode**. A photodiode (see Figure 3.8 for a symbol) behaves like an LED but in an opposite fashion. The amount of current that the photodiode passes is proportional to the amount of light it receives, and the current flows from the cathode to the anode (reverse biased). Photodiodes are commonly used as light sensors.

Figure 3.8
Symbol of a photodiode

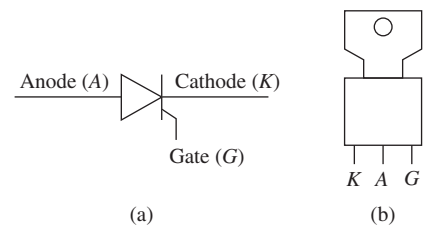


3.3 THYRISTORS

A **thyristor** (silicon-controlled rectifier or SCR) is a three-terminal semiconductor device that behaves like a diode but with an additional terminal. The additional terminal is called a *gate*, and when a small current flows into the gate, it allows a much larger current to flow from the anode to the cathode (provided that the voltage between the anode and the cathode is forward biased). The symbol and a typical component form of a thyristor are shown in Figure 3.9.

Figure 3.9

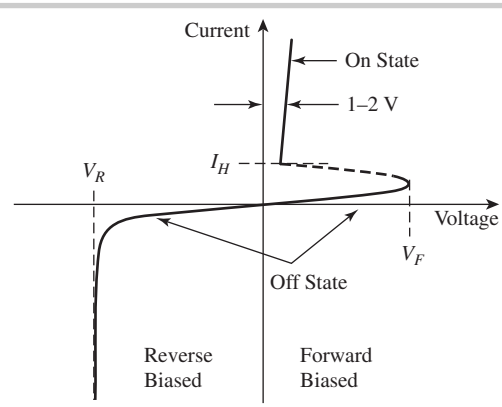
(a) Thyristor symbol and (b) typical component



The current–voltage characteristics of a thyristor are shown in Figure 3.10. When no current is applied to the gate (off state), the current flow between the anode and the cathode in the thyristor is negligible for voltages greater than V_R and less than V_F. Note that the forward voltage (V_F) of a thyristor is quite large (from 50 up to several hundred volts), unlike that of a regular diode. When a small current (mA range) is applied to the gate, the thyristor conducts if the voltage applied to it causes it to be forward biased. When the thyristor is conducting (on state), the forward voltage across the thyristor is small (1 to 2 V), and the thyristor current can be in the several ampere range. Note that if the current to the gate is cut off, the thyristor continues to conduct as long as the voltage applied to it causes it be forward biased. The thyristor is turned off only when the current between the anode and the cathode drops below a certain value called the **holding current** (I_H). The **gate current** (I_{GT}) that causes the thyristor to conduct is small and is typically a few milliamps or less. For example, for the 2N6401 SCR, V_F is 100 V when not conducting and 1.7 V when conducting, I_{GT} is about 30 mA, and I_H is about 20 mA. The ability of the thyristor to remain on even though the gate current is switched off is called latching.

Figure 3.10

Current–voltage relationship for a thyristor



Thyristors are commonly used in power control applications to control heaters, dimming switches, and motors. They are particularly useful in controlling the current from an AC source to control the speed of AC-driven motors, such as the Universal Motor. Figure 3.11 shows such a circuit used for this purpose.

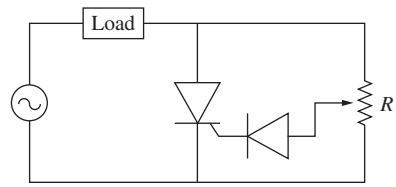


Figure 3.11

Half-wave variable-resistance phase-control circuit

This circuit is called a **half-wave variable-resistance phase-control circuit**. Because a thyristor is unidirectional, the circuit only affects the positive portion of the AC signal or half-wave. When the voltage crosses the zero mark, the thyristor stops conducting. As seen in Figure 3.11, the gate current to the thyristor is controlled by a potentiometer through a diode. The diode is used to prevent the negative half of the AC signal from affecting the gate. Changing the resistance of the potentiometer causes the gate current to change. Since the applied voltage is sinusoidal (not constant), this causes the gate to trigger at different times with respect to the AC signal (hence the name phase control). If the triggering occurs at the beginning of the positive half of the voltage signal, then all of the current is passed, as shown in Figure 3.12(b). If the triggering occurs at a later time, then only a portion of the current is passed, as shown in Figure 3.12(c). Since the duration of the current passed affects the speed or delivered power to the controlled device, the thyristor offers a simple way to control the power.

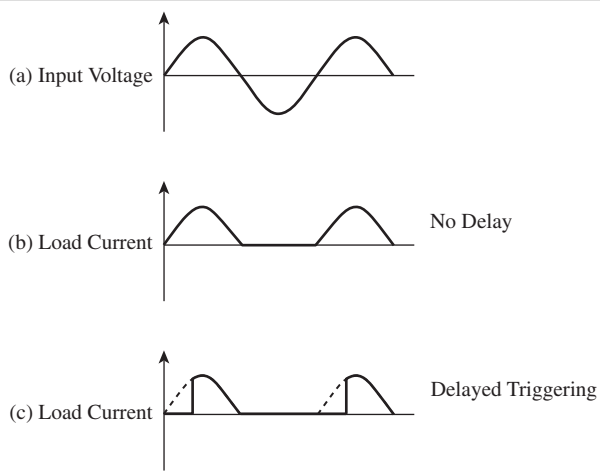


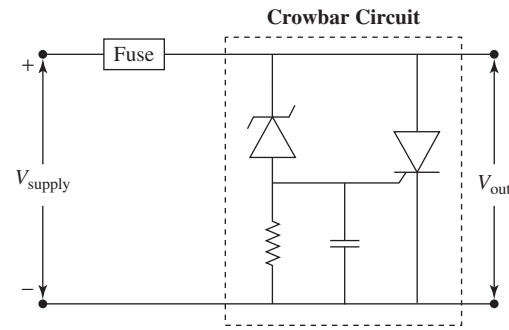
Figure 3.12

Current output of a half-wave variable-resistance phase-control circuit

Another application of thyristors is to use them in overvoltage protection circuits. The circuit shown in Figure 3.13 is commonly referred to as a **crowbar circuit**. It is used to protect from power surges or power-supply malfunction problems. It basically uses a Zener diode in combination with a thyristor. The Zener voltage should be selected to be higher than the nominal supply voltage. When the Zener diode switches on due to overvoltage condition, this causes a current to flow to the gate of the thyristor. The thyristor will conduct, allowing a large current to flow, thus acting as a short circuit and blowing the circuit fuse. The capacitor is added to prevent the thyristor from triggering when powered up. Because of component tolerances, this circuit will operate reliably only if the overvoltage is 30 to 40% higher than the nominal voltage.

Figure 3.13

Crowbar circuit for overvoltage protection

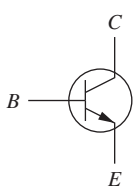


3.4 Bipolar Junction Transistor

A transistor is a solid-state switch that opens or closes a circuit. Unlike an electromechanical relay, the switching action in a transistor is caused by non-mechanical motion and is due to the change in the electrical characteristics of the device. A transistor is a three-terminal device. One terminal is used as the control input, another is connected to the load voltage, while the third is connected to ground or a constant potential. There are several types of transistors available: the bipolar junction transistor (BJT), the field effect transistor (FET), and the metal-oxide semiconductor field effect transistor (MOSFET). We will discuss the BJT and the MOSFET, which are widely used. We will start by discussing the BJT. There are two types of BJT: *npn* and *pnp*, which refer to the arrangement of *n*-type (negative) and *p*-type (positive) semiconductors in the construction of the transistor. We will limit the discussion to the *npn* configuration, which is more widely used. A schematic of an *npn* BJT is shown in Figure 3.14. The terminals of the BJT are labeled as the emitter (*E*), base (*B*), and collector (*C*).

Figure 3.14

Schematic of an *npn* bipolar junction transistor (BJT)



Let us label the voltages applied to the *B*, *C*, and *E* terminals as V_B, V_C, and V_E, respectively. Let us further define the following voltage differences:

$$V_{BE} = V_B - V_E$$

$$V_{CE} = V_C - V_E$$

Some general characteristics of a BJT are

- A BJT is an active device that requires power to operate.
- The BJT is a current-controlled device whose operation depends on the magnitude of the current supplied to the base.
- A small base current allows a much larger current to flow between the collector and the emitter.
- The BJT has three states of operation. These include the off or non-conducting state, the linear state, and the saturation state. These states of operation are determined by the magnitude of the V_{BE} and V_{CE} voltage. The former is set by the current supplied to the base.
- The voltage at the emitter (V_E) is always lower than the voltage at the base (V_B) by about 0.6 V.
- The collector voltage (V_C) has to be more positive than the emitter voltage (V_E).
- If AC voltages are applied to the base input, then a DC offset voltage (called a bias voltage) needs to be added in series to the AC voltage to enable the transistor to be controlled by both the positive and negative parts of the AC signal.

Two of the most common standard BJT circuits are called the transistor switch (or common emitter circuit) and the emitter follower circuit. These circuits are discussed next.

3.4.1 Transistor Switch Circuit

The **transistor switch** or **common emitter circuit** is shown in Figure 3.15. In this circuit, V_{in} is the control voltage, V_{out} is the output voltage, and V_{CC} is the supply voltage. This circuit is also called the *common emitter circuit*, because both the emitter and the supply voltage ground are connected to the same common point. In this circuit, a resistor (R_C) is always placed between the supply voltage lead and the collector. In practice, this resistor represents the resistance of a load (such as an LED or motor) that needs to be switched on and off, and hence, the name of this circuit.

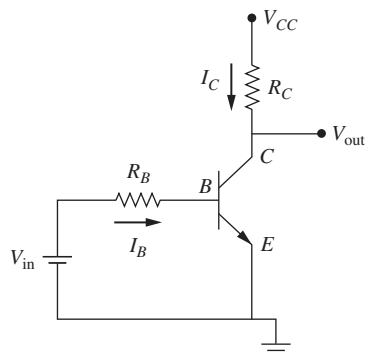


Figure 3.15

Common emitter circuit

The **transfer** and **output characteristics** of a BJT are shown in Figure 3.16. In Figure 3.16(a), the collector current (I_C) is plotted against V_{BE}. The figure shows that the collector current (I_C) is zero unless V_{BE} exceeds 0.6 V, at which point I_C starts increasing. Figure 3.16(b) shows the relationship between I_C and V_{CE} as a function of the base current (I_B). The figure shows that away from the vertical axis or in the linear region, the collector current is mainly a function of the base current and does not change appreciably with an increase in V_{CE}. This region is called the *active* or *linear* region. Close to the vertical axis or in the saturation region, I_C is a function of both V_{CE} and I_B.

As mentioned before, a BJT has three states of operation. When $V_{BE} < \sim 0.6$ V, the transistor is said to be in the **off state** (non-conducting state). In this state, no current flows between the collector and the emitter, so $I_C = 0$. The V_{out} voltage will be the same as the V_{CC} voltage, because no current flows between V_{CC} and V_C.

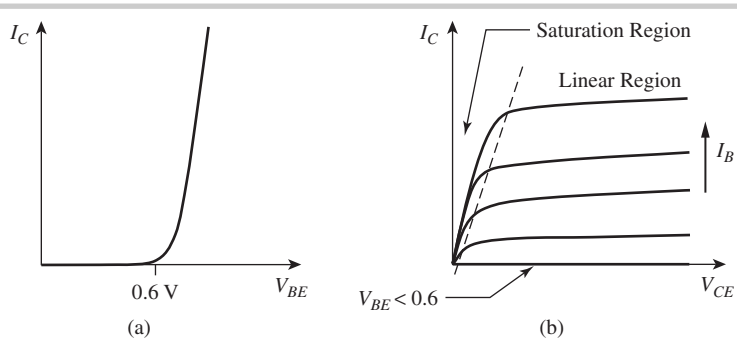


Figure 3.16

(a) Transfer and (b) output characteristics of a BJT

When V_{BE} is >= ~0.6 V and < ~0.7 V and $V_{CE} > 0.2$ V, the transistor is in the **linear operation state**. In the linear operation state, the collector current (I_C) is linearly related to the base current (I_B) by the following relationship:

$$I_C = \beta I_B = h_{FE} I_B \tag{3.1}$$

where β or h_{FE} is the current gain. The current gain is not constant, and its value is dependent on the current (I_C) as well as V_{CE} and temperature. A typical value for β is 100, but it could range from 50 to 1000 due to variations in the manufacturing process.

When V_{BE} is >= ~0.7 V, the transistor is in the **saturation state**. In the saturation state, current flows between the collector and the emitter, and V_{CE} has a value of ~0.2 V. V_{out} in this case will be same as V_{CE}, and the output voltage will switch from V_{CC} to ~0.2 V when the transistor switches from the off state to the saturation state.

In the transistor switch circuit, the transistor is normally designed to operate in either the off state or the on (saturation) state, but not in the linear state. The question is then what is the minimum V_{in} voltage needed to cause the transistor to saturate? By referencing Figure 3.15 and using KVL, we get

$$V_{\text{in}} = I_B R_B + V_{BE} \tag{3.2}$$

and just before saturation, I_B is related to I_C by

$$I_B = I_C/\beta \tag{3.3}$$

where I_C is determined from

$$I_C = (V_{CC} - V_{CE})/R_C \tag{3.4}$$

These equations can be solved to find V_{in} to cause saturation. Example 3.2 illustrates these calculations with data from a widely used small transistor, the 2N3904 (see website for complete data sheet).

Example 3.2 Voltage Saturation Calculations for the 2N3904 Transistor

Using the data sheet for the 2N3904 transistor and with reference to Figure 3.15, determine the input voltage needed to cause the transistor to saturate. What is the output voltage (V_{out}) of this circuit during the saturation and the off states if $V_{CC} = 10$ V? Let R_C be 1 kΩ, and R_B be 5 kΩ. Also, determine the power output of this transistor.

Solution:

From the data sheet, $V_{CE} = 0.2$ V at saturation for $I_C = 10$ mA. From Equation (3.4), I_C is given by

$$I_C = (10 - 0.2)/1000 = 9.8 \text{ mA}$$

which is close to the assumed value for I_C. Notice that I_C has to be smaller than the 200 mA limit for the collector current, which is satisfied in this case. Also from the data sheet, *beta* or h_{FE} is 100 for $I_C = 10$ mA, and V_{BE} is greater than 0.65 V at saturation. We set $V_{BE} = 0.7$ V. From Equation (3.3), I_B just before saturation is given by

$$I_B = 9.8/100 = 0.098 \text{ mA}$$

Now plugging this value into Equation (3.2) gives

$$V_{\text{in}} = 0.098 \times 5 + 0.70 = 1.19 \text{ V}$$

To insure saturation, V_{in} has to be greater than 1.19 V. This can be achieved easily if we let V_{in} be 2 V for example.

When this transistor is off, V_{in} has to be less than V_{BE} when the transistor just turns on (less than 0.6 V). In this case, V_{out} will be equal to V_{CC} (10 V). When the transistor is in saturation, $V_{out} = V_{CE} = 0.2$ V.

The output power of this transistor is the power that is dissipated by the load, which in this case is the R_C resistor. Power is then computed from $I_C^2 R$:

$$\text{Power} = (9.8 \times 10^{-3})^2 \times 1000 = 9.6 \times 10^{-2} \text{ W}$$

3.4.2 Emitter Follower Circuit

The **emitter follower circuit** is shown in Figure 3.17. Note how the output is connected to the emitter is this case, and there is no resistor between V_{CC} and the collector. This circuit is called the emitter follower, because the output voltage follows the input voltage with a difference of about 0.6 V. Assume first that there is no resistor R_B in this circuit. Then $V_B = V_{in}$,

$$V_{out} = V_E = V_{in} - V_{BE} = V_{in} - 0.6 \text{ for } V_{in} > 0.6 \text{ V} \tag{3.5}$$

and

$$V_{out} = 0 \text{ for } V_{in} < 0.6 \text{ V} \tag{3.6}$$

Now if the resistor R_B was present, we need to account for the voltage drop across this resistor and V_{out} is then equal to

$$V_{out} = V_{in} - I_B R_B - 0.6 \text{ for } V_{in} > 0.6 \text{ V} \tag{3.7}$$

But $I_E = I_C + I_B$, $I_E = V_{out}/R_E$, and $I_C = \beta I_B$ when the transistor is in the linear state. This gives

$$I_B = \frac{V_{out}}{(1 + \beta)R_E} \tag{3.8}$$

and

$$I_E = I_B(1 + \beta) \tag{3.9}$$

Equation (3.9) shows the current gain of this circuit is $(\beta + 1)$. Substituting Equation (3.8) into Equation (3.7) and solving for V_{out}, we get

$$V_{out} = (V_{in} - 0.6)\frac{(1 + \beta)R_E}{R_B + (1 + \beta)R_E} \tag{3.10}$$

Equation (3.10) shows that the output voltage is linearly related to the input voltage and is independent of the supply voltage V_{CC}. The output voltage is also in

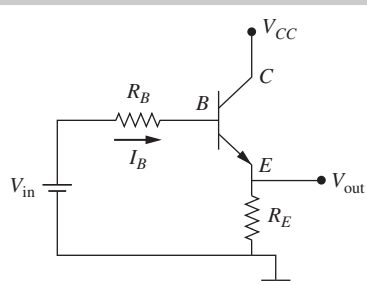


Figure 3.17
Emitter follower circuit

phase with the input voltage, and voltage, gain is slightly less than 1 as seen from Equation (3.10). Equations (3.8) through (3.10) apply as long as the transistor is not in saturation. When the transistor saturates, V_{out} is equal to $V_{CC} - 0.2$ because V_{CE} is about 0.2 volts at saturation. Example 3.3 further illustrates the voltage calculations for a BJT.

BJT transistors have certain parameters that should not be exceeded. These parameters include maximum collector current and the power dissipation capability. These parameters are listed in Table 3.1 for three common *npn* transistors. The TIP102 transistor is called a **Darlington transistor** and it consists of two cascaded BJT transistors to amplify the collector current (see Figure 3.18). Note that the power dissipation capability of a transistor is dependent on the environment temperature. In Table 3.1, the power is listed for air temperature of 25°C. The power dissipation decreases with increasing temperature.

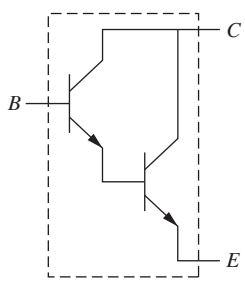
Table 3.1

Characteristics of common *npn* BJT transistors

Part #	Max V_{CE}	Max V_{BE}	Max I_C	β	Power Dissipation at $T_A = 25°C$
2N3904	40 V	6 V	200 mA	30–300	0.625 W
TIP29	40 V	5 V	1 A	15–75	30 W
TIP102	100 V	5 V	8 A	200–20000	80 W

Figure 3.18

Schematic of a Darlington transistor



In a BJT, the power that is seen by the load is computed from the product of the square of the collector current and the load resistance. BJTs are typically used for low power applications. Note that while a transistor has a very fast switching time in the order of 10's of ns or less, there is no electrical isolation between the base circuit and the load circuit unlike that of an electro-mechanical relay. In a relay, the coil circuit and the contact circuits are electrically isolated, so they are better suited to handle noise.

Example 3.3 Analysis of a BJT circuit

Determine the voltages at points 1 and 2 in the circuit shown in Figure 3.19 for a) $V_{in} = 0.1$ V and b) $V_{in} = 3$ V. Let $R_C = 1$ kΩ, $R_B = R_E = 100$ Ω, and $V_{CC} = 10$ V.

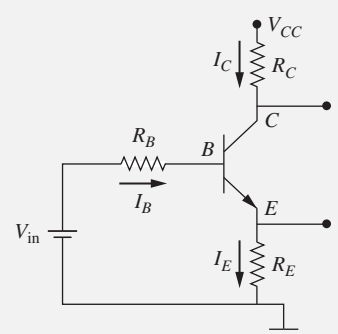


Figure 3.19

Solution:

a) For $V_{in} = 0.1$ V, the transistor is off because V_{in} has to be larger than 0.6 V to cause the transistor to start conducting. So $V_1 = V_{CC} = 10$ V since the current I_C is zero. V_2 is equal to zero because V_{BE} is less than 0.6 V.

b) For V_{in} = 3 V, the transistor is either operating in the linear range or saturated. We will assume that the transistor is just at the point of being saturated, and we will check this assumption by comparing the currents I_C and I_B.

Applying KVL to the V_{CC} loop gives

$$V_{CC} = I_C R_C + V_{CE} + I_E R_E \quad (1)$$

Similarly, applying KVL to the V_{in} loop gives

$$V_{in} = I_B R_B + V_{BE} + I_E R_E \quad (2)$$

Noting that $I_E = I_C + I_B$, and using the given values for R_B, R_C, R_E, V_{in} and V_{CC}, and assuming V_{CE} = 0.2 V and V_{BE} = 0.7 V at saturation, Equations (1) and (2) give

$$10 = 1100\, I_C + 0.2 + 100\, I_B \quad (3)$$

$$3 = 100\, I_C + 0.7 + 200\, I_B \quad (4)$$

Solving Equations (3) and (4) for I_B and I_C, we get I_C = 8.24 mA, and I_B = 7.36 mA. For I_C = 10 mA, the current gain β is about 100 if the transistor is operating in the linear range. Since $I_C \neq \beta\, I_B$, the transistor is in saturation and the assumption is correct. This gives V_1 = 1.76 V and V_2 = 1.56 V.

3.4.3 Open Collector Output

Many sensors used in mechatronic applications such as proximity sensors, see Section 7.4, have electronic circuits that use an internal BJT transistor as an interface. The output of the sensor electronic circuit drives the base of the transistor. These circuits are normally known as **open collector output** voltage circuits. To get an output from these sensors, an appropriate 'pull-up' resistor or load, and the supply voltage needs to be applied to the terminals of the sensor. Figure 3.20 shows a typical wiring for such a sensor. In this example, a positive voltage needs to be applied to terminal 1 and a 'pull-up' resistor needs to be connected between terminals 1 and 2. When the proximity sensor is OFF, the transistor is not in saturation, and there will be no voltage drop across the load resistor, since the collector terminal (2) is open with respect to the emitter terminal (3). The output in this case will be 'pulled up' by the load resistor to the value of the external supply voltage V. When an object is detected by the proximity sensor, the transistor conducts and a voltage drop develops across the load resistor, resulting in the output voltage changing from the value of the supply voltage to almost zero. Example 3.4 illustrates an open collector circuit.

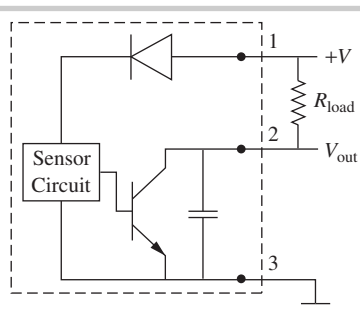


Figure 3.20

Typical circuit for an *npn* type non-contact, capacitive-type proximity sensor

Example 3.4 Open Collector Calculation

A proximity sensor with an output circuit similar to that shown in Figure 3.20 requires a 24-VDC supply voltage. Select a pull-up resistor and a wiring scheme for this sensor so the voltage output of the sensor can be read by a digital input port with 0 and 5 V logic levels. The maximum current through the pull up load resistor should be limited to 100 mA.

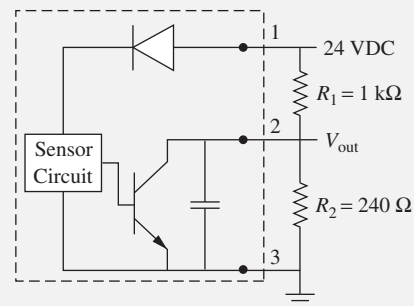


Figure 3.21

Solution:

The wiring scheme to perform this function is shown in Figure 3.21. Since the output voltage should be limited to 5 V when the sensor is off, we need to use a voltage dividing circuit in this setup. The resistors R_1 and R_2 are selected to have a resistance of 1 kΩ and 240 Ω respectively to achieve this. Note that these resistance values are standard. When the sensor is off, V_{out} will be

$$24 - 1000\frac{24}{1240} = 4.65\ \text{V}$$

which is well within the voltage threshold for the high logic (see Table 3.12). When the sensor is on, V_{out} is about 0.2 V. The maximum current through the load resistor occurs when the sensor is on. The current in this case is

$$\frac{24 - 0.2}{1000} = 23.8\ \text{mA}$$

This is well within the desired specification.

Figure 3.22

(a) Phototransistor and (b) photo interrupter

(Courtesy of ROHM Semiconductor, USA, San Diego, CA)

(a) (b)

3.4.4 Phototransistor, Photo Interrupter, and Opto-Isolator

Instead of using a voltage source to saturate the transistor, a **phototransistor** (see Figure 3.22(a)) uses light that is received by a photodiode to do the same thing. Typically a phototransistor and an LED are packaged together to make optical sensors that can be used to detect the presence of objects. In these sensors, which are commonly referred to as **photo interrupters** (see Figure 3.22(b)), the LED provides a light source that is received by the phototransistor. An interruption of the light received by the phototransistor causes the phototransistor to change state, thus indicating the presence of an object in the path between the LED and the phototransistor.

An **opto-isolator** or an **optocoupler** combines two elements (a light-emitting device such as a diode and a light-sensitive device) similar to a photo interrupter but in an enclosed package. An opto-isolator is also designed for a different purpose, which is to provide an optical coupling between the input and the output sides. The light emitter on the input side takes a voltage signal and converts it into a light signal. On the output side, the light-sensitive device detects the light from the emitter and converts it back to a voltage signal. The light-sensitive device could be a phototransistor, a photodiode, or a thyristor. This optical coupling provides

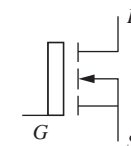
electrical noise isolation between the input and the output sides. To take advantage of this isolation, a separate power supply should be used for the input and output sides. Opto-isolators are used to prevent voltage spikes on one side of the device to damage or affect components on the other side. Opto-isolators are available with isolation of 5 kV or more between the input and output sides.

3.5 Metal-Oxide Semiconductor Field Effect Transistor

Metal-oxide semiconductor field effect transistors (MOSFETs) are the other family of transistors that are commonly used. The MOSFET is based on the original field effect transistor (FET) that was introduced in the 1960s. Similar to the BJT family, they are also three terminal devices, but they have different names for the terminals, and they operate differently. The three terminals are the gate (similar to base), drain (similar to collector), and source (similar to emitter). The naming of the terminals comes from the flow of electrons between the source and drain when the transistor is conducting. The most commonly used MOSFET is the enhanced type and is available as *n*- or *p*-type. We will limit the discussion to the *n*-type enhanced MOSFET here. Figure 3.23 shows the symbol of the *n*-type MOSFET.

Figure 3.23

Symbol of an *n*-type MOSFET



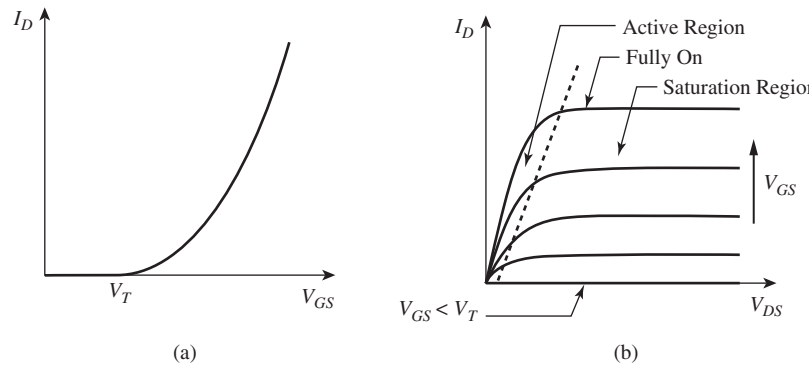
MOSFETs have the following characteristics.

- The voltage applied to the gate (or the electric field) is the signal that controls the operation of the transistor and hence the name field effect transistor. This is in contrast to a BJT where the current applied to the base controls its operation.
- The gate is insulated from the drain-source circuit. This is indicated in the symbol by the separation of the gate terminal from the drain-source connection. The gate has a very high internal resistance ($R_{Gate} = \sim 10^{14}\ \Omega$), such that almost no current flows into the gate. This insulation makes it easy to analyze MOSFET circuits, because the gate circuit can be analyzed separately from the drain-source circuit. In addition, the high input resistance means that the gate draws no current except for a small leakage current in the nanoampere range.
- The high-input impedance of a MOSFET gives it an advantage in interfacing with other logic circuits.
- MOSFETs have three states: cutoff, active, and saturation, which are similar to BJTs.
- They act as voltage-controlled resistors. When the transistor is OFF, the drain-source resistance is very high, and when the transistor is fully ON, the drain-source resistance is very low (can be less than 1 Ω). When the transistor is ON, current flows from the drain to the source (electrons travel in the opposite direction).
- *n*-type enhancement MOSFETs operate with a positive voltage applied to the gate.
- MOSFETs have a higher power rating and generate less heat than BJTs.

The **transfer** and **output characteristics** of a MOSFET are shown in Figure 3.24. Figure 3.24(a) shows the relationship between the drain current (I_D) and the voltage between the gate and the source (V_{GS}). The figure shows that the drain current is zero until V_{GS} exceeds the threshold voltage (V_T) for the MOSFET. When $V_{GS} < V_T$, the MOSFET is said to be in the **cutoff or non-conducting state**. The threshold voltage for normal MOSFETs (such as 2N4351) is between 2 to 5 V, while for logic-level MOSFETs (ones that are designed to be driven

Figure 3.24

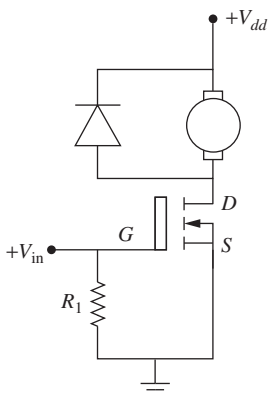
(a) Transfer and (b) output characteristics of a MOSFET



directly from outputs of logic gates, such as microcontrollers) the threshold voltage is about 0.3 to 1.0 V. As V_{GS} increases above V_T, the drain current increases. Figure 3.24(b) shows the relationship between the drain current and the voltage between the source and the drain (V_{DS}) for different values of V_{GS}. For a given $V_{GS} > V_T$, the drain current increases with V_{DS} for a small V_{DS}. This is called the **active** or **ohmic region**, where the MOSFET acts like a variable resistor whose resistance is controlled by V_{GS}. However, as V_{DS} increases, the drain current levels off and stays constant. This is called the **saturation region**, where the drain current value is independent of V_{DS}. When V_{GS} is significantly higher than the threshold voltage (approximately 10 V for a normal MOSFET), the transistor is said to be in the fully ON state where the drain current is maximum.

Figure 3.25

MOSFET circuit for driving a motor



MOSFETs are typically used for switching applications (ON/OFF) to drive motors or LEDs. A typical circuit is shown in Figure 3.25. When the transistor is OFF, no current flows from the drain to the source, and the motor is OFF. When the transistor is fully ON, current flows to the motor, and the motor is ON. Note the use of the flyback diode in the circuit to protect the transistor from the large voltage build-up that occurs when the transistor is switched off. The resistor at the input is used to drive the gate input to ground and completely turns OFF the transistor when the input voltage is zero.

Some of the parameters of a logic-level MOSFET (NTE2980), normal MOSFET (2N4351), and a Power MOSFET (IRFZ14) are shown in Table 3.2.

These parameters are defined here.

$R_{ds(ON)}$ The resistance between the drain and source terminals when the MOSET is fully turned on

$I_{D(max)}$ The maximum current between the drain and source that can be passed by the transistor. It is a function of $R_{ds(on)}$ and the package type of the transistor

Table 3.2

Parameters of selected MOSFETs

Part Number	On Resistance $R_{ds\,(ON)}$	Max. Drain Current $I_{D\,(max)}$	Power Dissipation P_d	Gate Threshold Voltage $V_{GS(th)}$	Drain-to-Source Breakdown Voltage V_{dss}
2N4351	<= 300 Ω	100 mA	375 mW	1–5 V	>= 25 V
NTE2980	0.2–0.28	6.7A (V_{GS} = 5 V)	25 W	1–2 V	>= 60 V
IRFZ14	0.2 Ω	10 A	43 W	2–4 V	>= 60 V

P_d The power dissipation rating for the transistor

$V_{GS(\text{th})}$ The minimum voltage between the gate and the source that causes the transistor to start conducting

V_{dss} The maximum voltage between the drain and source when the transistor is OFF

Similar to BJTs, the power-handling capacity of MOSFET transistors is a very important consideration in the selection of these components. When the power to be dissipated is above 1 W, the MOSFET is mounted on a heat sink. In these MOSFETs, the package has a metal tab which is mounted against the heat sink.

The dissipated power in a MOSFET is the product of the drain current and the voltage across the transistor. Since the voltage across the transistor is equal to the product of current and the resistance, the power is then given by

$$P_d = I_D^2 R_{ds(\text{on})} \tag{3.11}$$

when the transistor is fully conducting.

3.6 Combinational Logic Circuits

The invention of the transistor has led to the development of digital circuits in which transistors form the building blocks. Digital logic circuits can be classified into two categories. These are combinational logic circuits and sequential logic circuits. In a **combinational logic circuit**, the output is not dependent on the history of the input, and the circuit uses rules of mathematical logic to generate the output. On the other hand, in a **sequential logic circuit**, signal history is important and determines the output of the system. We start with combinational logic circuits. Table 3.3 lists the basic combinational logic devices that are used along with their symbols, logic function expressions, and truth tables. A **truth table** gives the output logic for all combinations of the input logic. Note that a bar above a logic variable means the inverse of that variable.

An example of a logic circuit that uses these devices is shown in Figure 3.26. The associated truth table and logic function are also shown.

Table 3.3 Basic combinational logic devices

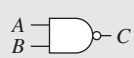
Device	Symbol	Logic Function Expression	Truth Table
AND gate		$C = A \cdot B$	A B C 0 0 0 1 0 0 0 1 0 1 1 1
NAND gate		$C = \overline{A \cdot B}$	A B C 0 0 1 1 0 1 0 1 1 1 1 0
OR gate		$C = A + B$	A B C 0 0 0 1 0 1 0 1 1 1 1 1

Table 3.3

(*Continued*)

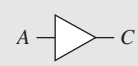
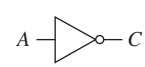
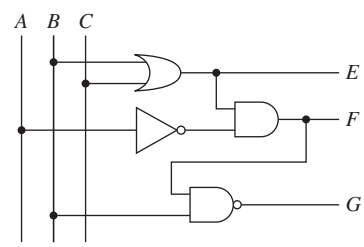
Device	Symbol	Logic Function Expression	Truth Table
NOR gate		$C = \overline{A + B}$	A B C 0 0 1 1 0 0 0 1 0 1 1 0
XOR gate		$C = A \oplus B$	A B C 0 0 0 1 0 1 0 1 1 1 1 0
Buffer		$C = A$	A C 0 0 1 1
Inverter		$C = \overline{A}$	A C 0 1 1 0

Figure 3.26

An example of a combinational logic circuit

Circuit



Truth Table

A	B	C	E	F	G
0	0	0	0	0	1
0	0	1	1	1	1
0	1	0	1	1	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	1	0	1

Logic Functions

$E = B + C$

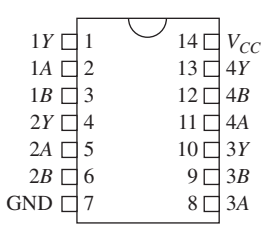
$F = \overline{A} \cdot E = \overline{A} \cdot (B + C)$

$G = \overline{B \cdot F}$

Figure 3.27

SN7402 package

(Courtesy of Texas Instruments, Dallas, TX)



The gates shown in Table 3.3 are available in packages. For example, Figure 3.27 shows the SN7402 package from Texas Instruments, which contain four independent two-input NOR gates. The *A*'s and the *B*'s are the gate's input, and the *Y*'s are the gate's output. V_{CC} is the supply-voltage connection pin. Note that the semicircular notch at the top of the IC is used as a mark for orientation of the device. Logic AND, NAND, OR, and NOR gates are also available with three and four inputs.

3.6.1 Boolean Algebra

Since digital circuits perform logic operations, we need to understand the logic rules that govern these operations. Here is a listing of rules that can be used to simplify Boolean expressions.

1. $A + A = A$
2. $A + 1 = 1$
3. $A + 0 = A$

4. $A \cdot A = A$
5. $A + B = B + A$
6. $A \cdot (B + C) = A \cdot B + A \cdot C$
7. $A + (B \cdot C) = (A + B) \cdot (A + C)$
8. $A + \overline{A} = 1$
9. $A \cdot \overline{A} = 0$
10. $A \cdot 0 = 0$
11. $A \cdot 1 = A$

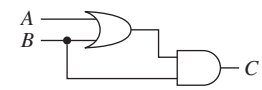
The following two rules are called De Morgan rules and are useful in converting between AND and OR gates:

12. $\overline{A + B + C} + \ldots = \overline{A} \cdot \overline{B} \cdot \overline{C} \ldots$
13. $\overline{A \cdot B \cdot C \ldots} = \overline{A} + \overline{B} + \overline{C}$

To illustrate these rules, consider the following example. Assume we are given a circuit, as shown in Figure 3.28.

Figure 3.28

Two-gate circuit



The output of this circuit in terms of the inputs is given by

$$C = (A + B) \cdot B$$

Using rule 6 and then rule 4, we can write the above expression as

$$C = A \cdot B + B \cdot B = A \cdot B + B$$

Also applying rule 11 and then rule 6, we get

$$C = A \cdot B + B = A \cdot B + B \cdot 1 = (A + 1) \cdot B$$

Since $A + 1 = 1$ by rule 2 and $(1 \cdot B) = B$ by rule 11, the output of this circuit is $C = B$. Thus, by using the above rules, we were able to convert a circuit that has two gates into one that requires no gate—just a wire between B and C.

As a further illustration of the above rules, consider Example 3.5.

Example 3.5 Boolean Logic Simplification

Simplify the following output function:

$$Q = (A \cdot C) + (B \cdot C) + (A \cdot \overline{B} \cdot C)$$

Solution:

Q can be written as

$$Q = (A + B) \cdot C + (A \cdot \overline{B}) \cdot C$$

by applying rule 6 to the first two expressions and factoring C from the third expression.

Application of rule 6 again gives

$$Q = [(A + B) + (A \cdot \overline{B})] \cdot C$$

Factoring out A and then applying rules 2 and 11 gives

$$Q = [A \cdot (1 + \overline{B}) + B] \cdot C = (A \cdot 1 + B) \cdot C = (A + B) \cdot C$$

Thus, the original expression which would need one inverter gate, four two-input AND gates, and two two-input OR gates now can be realized using just one OR gate and one AND gate.

3.6.2 Boolean Function Generation from Truth Tables

In this section, we look at the problem of finding a logic gate system with the minimum number of gates that can be used to realize a logic circuit operation that is specified in terms of a truth table. An application of this would be the design of a circuit to process the output from a number of ON/OFF sensors. The basic approach is to manipulate the logic functions into one of two equivalent forms.

Sum of products form: $A \cdot B + A \cdot C$

Or

Product of sums form: $(A + B) \cdot (A + C)$

In the **sum of products form**, different combinations of the inputs are AND-ed together to form products, and these products then are OR-ed together to generate the output. In the **product of sums** form, different combinations of the inputs are OR-ed together to form sums, and sums then are AND-ed together to generate the output. The sum of products form is more commonly used. As an example, consider the following truth table (Table 3.4) that defines the output Q in terms of the inputs A and B. Using the sum of products form, we included in each row of the table the product form that generates the output in that row. If the input is low (or zero), then that input is shown with a bar above it in the products expression. From the table, we can then say that this logic system is given by

$$Q = \overline{A} \cdot B$$

Note that we only considered rows that have an output of 1. Rows with zero output do not contribute to the final expression.

Table 3.4
Logic truth table

A	*B*	Output *Q*	Products
0	0	0	$\overline{A} \cdot \overline{B}$
0	1	1	$\overline{A} \cdot B$
1	0	0	$A \cdot \overline{B}$
1	1	0	$A \cdot B$

Table 3.5
Graphical representation of Table 3.4

	$\overline{B}$	*B*
$\overline{A}$	0	1
A	0	0

This example also can be solved graphically, as shown in Table 3.5, where we construct a table that has all of the input combinations. The output of the system is decided from the cells that have a non-zero output. In this case, we have only one non-zero cell, so we can read the output as determined before as $Q = \overline{A} \cdot B$.

This graphical technique works very well for logic functions with many inputs and is part of a method using **Karnaugh maps** (K-maps). A Karnaugh map is a graphical method that can be used to produce simplified Boolean expressions from sums of products obtained from truth tables. We will give a brief overview of Karnaugh maps in this text. For more detail, the reader should consult textbooks on digital logic design (see, for example, [8]). To illustrate the Karnaugh map approach, consider the truth table for a three-variable input problem that is shown in Table 3.6. We added a column to that table to show the product form for rows that have non-zero output. Unlike the truth table shown in Table 3.4, which has only one non-zero output, this truth table has five non-zero outputs. Trying to simplify these five products using the Boolean rules discussed earlier is not always straightforward. We will instead use the Karnaugh map approach for this example, since it offers a simpler method to obtain the output.

A B C	Output	Products
0 0 0	0	
0 0 1	1	$\overline{A} \cdot \overline{B} \cdot C$
0 1 0	0	
0 1 1	1	$\overline{A} \cdot B \cdot C$
1 0 0	0	
1 0 1	1	$A \cdot \overline{B} \cdot C$
1 1 0	1	$A \cdot B \cdot \overline{C}$
1 1 1	1	$A \cdot B \cdot C$

Table 3.6

Three-variable input truth table

The Karnaugh map for the data in Table 3.6 is shown in Table 3.7. Because we have three input variables in this problem, the Karnaugh map has eight elements. Note how the rows in the map are labeled such that only one variable changes in adjacent rows (i.e., $\overline{A} \cdot \overline{B} \rightarrow \overline{A} \cdot B \rightarrow A \cdot B \rightarrow A \cdot \overline{B}$). In the first step in this approach, map cells corresponding to non-zero output have a value of 1 placed in them. The next step is to group adjacent horizontal or vertical map cells that have a 1 in them. For this example, we have two groupings: a vertical group that has four cells and a horizontal group that has two cells. The last step is to derive a logic expression from the cell groupings. As seen in Table 3.7, the 1-output in the vertical group is independent of the values of A and B, so Q corresponding to this group is C. Also from the horizontal group, the 1-output is independent of the value of C, so Q is $A \cdot B$.

	$\overline{C}$	C
$\overline{A} \cdot \overline{B}$		1
$\overline{A} \cdot B$		1
$A \cdot B$	1	1
$A \cdot \overline{B}$		1

Table 3.7

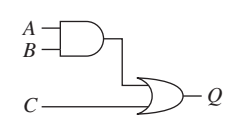
Karnaugh map for data in Table 3.6

Combining these expressions, the output of the truth table is then $Q = C + A \cdot B$. This output can be realized by the circuit shown in Figure 3.29.

Example 3.6 illustrates the use of combinational logic circuits.

Figure 3.29

Circuit corresponding to Table 3.6



Example 3.6 Application of combinational logic circuits

Design a combinational logic circuit to process the output from three sensors (A, B, and C). The circuit output should be on if the following is true:

- *A* is ON, but both *B* and *C* are OFF
- *A* is OFF, but either *B* or *C* is ON

Solution:

We first construct a truth table (Table 3.8) that has all of the input combinations and follows the above rules. Then we construct a Karnaugh map (Table 3.9) to

map all the non-zero outputs. Both of these entities are shown below:

Table 3.8 Truth table

A B C	Output	Products
0 0 0	0	
0 0 1	1	$\overline{A} \cdot \overline{B} \cdot C$
0 1 0	1	$\overline{A} \cdot B \cdot \overline{C}$
0 1 1	1	$\overline{A} \cdot B \cdot C$
1 0 0	1	$A \cdot \overline{B} \cdot \overline{C}$
1 0 1	0	
1 1 0	0	
1 1 1	0	

Table 3.9 Karnaugh map

	$\overline{C}$	C
$\overline{A} \cdot \overline{B}$		1
$\overline{A} \cdot B$	1	1
$A \cdot B$		
$A \cdot \overline{B}$	1	

We then group adjacent cells that have a 1 in them. From the Karnaugh map, we see that the output corresponding to the vertical group is $\overline{A} \cdot C$ because it is independent of the value of B. Similarly, the output corresponding to the horizontal group is $\overline{A} \cdot B$ because it is independent of the value of C. From the last row, the output is $A \cdot \overline{B} \cdot \overline{C}$. Combining these expressions, the output can be written as:

$$Q = \overline{A} \cdot C + \overline{A} \cdot B + A \cdot \overline{B} \cdot \overline{C} = \overline{A} \cdot (B + C) + A \cdot \overline{B} \cdot \overline{C}$$

This output can be realized using the appropriate gates (see Problem 3.11). Note the correspondence between the circuit output-logic expression and the problem statement.

Combinational logic circuits are used in a variety of useful applications, including multiplexers, decoders, and converters. A discussion of multiplexers and decoders follows.

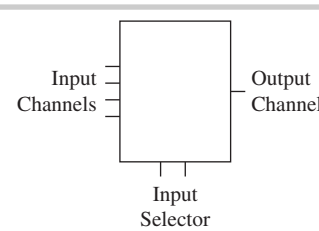
3.6.3 Multiplexers and Decoders

A **multiplexer** is a circuit that selects one input out of the several available to be connected to the output (see Figure 3.30). It is commonly used in the design of analog-to-digital convertors and in microcontroller circuits to select the timing source.

Multiplexer circuits are built from a combination of basic logic gates. Figure 3.31 shows the circuit for a two-input channel multiplexer. The desired channel number is selected by setting the value of the *Channel Selector* input. If the *Channel Selector* input value is 0, then channel 0 is selected. The input connected to channel 0 will then be transmitted to the output channel in this case. Similarly, if the *Channel Selector* input value is 1, then channel 1 is selected. If the multiplexer has four input channels instead of two, then we can see that we need to have two channel selector inputs. We can generalize this to a multiplexer with 2^n input channels, which will need n channel selector inputs.

Figure 3.30

A multiplexer



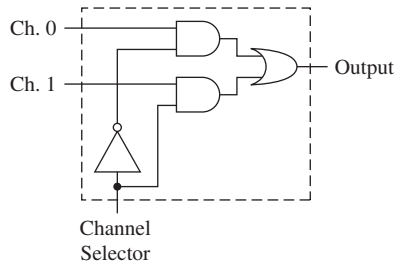


Figure 3.31

Two-input channel multiplexer circuit

A **decoder** or a **demultiplexer** operates in an opposite fashion to a multiplexer. A decoder with n inputs and m outputs will activate only one of the m outputs for a specified pattern of the n inputs. For example, a decoder with four inputs and one output will have a high output for only one combination of the four inputs. For all other combinations, the output will be low. Decoder circuits are used to select devices that are connected on a common line or a bus system. To select a particular device, the address of the device is placed on the bus that connects all of the devices. If the binary pattern of the address placed on the bus matches the individual address for that device, then that device is selected.

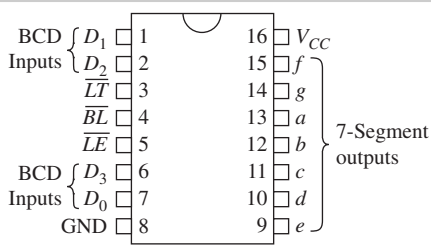


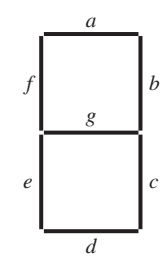
Figure 3.32

Pin layout for the BCD-to-7 decoder CD74HC4511 IC
(Courtesy of Texas Instruments, Dallas, TX)

A particular use of decoder circuits is in binary-coded decimal (BCD) format (see Section 4.2) to decimal-conversion applications. A popular commercial chip is the BCD-to-7 chip, such as the CD74HC4511 IC (see Figure 3.32) that is used to drive seven-segment digital displays (see Figure 3.33). Here the input to this IC is the BCD corresponding to the digit to be displayed on the display, and the output is a combination of the segments a through g.

Figure 3.33

Seven-segment digital display

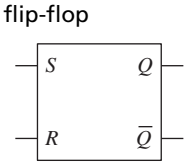


3.7 Sequential Logic Circuits

Unlike a combinational logic circuit, a sequential-logic circuit output is dependent on the history of the input. A sequential logic circuit can be thought of as consisting of a combinational logic circuit and memory. A basic sequential logic circuit is the **flip-flop**, which is a sequential logic device that can store and switch between two binary states. Examples of other sequential circuits include counters, shift registers, and microprocessors. We will discuss several types of flip-flops including the SR, the clocked SR, the JK, the D, and the T. We will start by talking about the SR flip-flop.

Figure 3.34

SR flip-flop

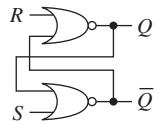


SR Flip-Flop The **set-reset (SR)** flip-flop has the symbol shown in Figure 3.34. It has two inputs, called S and R, and two outputs, called Q and complementary $\overline{Q}$. The operation of the SR flip-flop is set by the following rules.

1. When $S = 0$ and $R = 0$, the output of the flip-flop does not change.
2. When $S = 1$ and $R = 0$, the flip-flop is set to $Q = 1$ and $\overline{Q} = 0$.

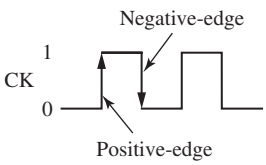
3. When $S = 0$ and $R = 1$, the flip-flop is reset to $Q = 0$ and $\overline{Q} = 1$.
4. S and R are not allowed to be set to 1 simultaneously, since the output will not be predictable.

Figure 3.35
Equivalent circuit for an SR flip-flop



To understand the operation of the SR flip-flip, consider the equivalent circuit made up of two NOR gates with feedback shown in Figure 3.35. Assume that we started with $Q = 0$ and $\overline{Q} = 1$. Now if S is set to 1 and R is 0, $\overline{Q}$ will reset to zero. Due to feedback, Q will also change from 0 to 1. By tracing the output of this circuit for different combinations of S and R, we can verify all of the above rules.

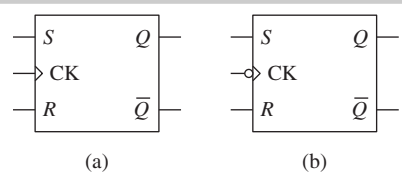
Figure 3.36
Clock transitions



Clocked SR Flip-Flop A clock signal is a two-state signal. It is commonly a periodic square-wave signal, but it also can be a non-periodic signal made up of a collection of pulses. A periodic square-wave clock signal is shown in Figure 3.36. When a device that uses a clock input responds to the low-to-high change in the clock signal, it is called a **positive edge-triggered** device. Similarly, a device that responds to the high-to-low clock transition is called a **negative edge-triggered** device.

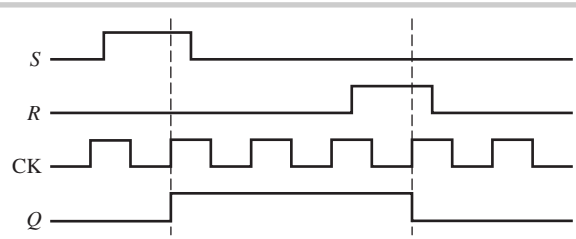
A clocked SR flip-flop is an SR flip-flop with added clock input. In a clocked SR flip-flop, the output changes state at clock transitions. This is done as to provide synchronization of the output change in complex circuits. Note that a flip-flop with no clock input is called a simple or transparent flip-flop. Two variations of a clocked SR flip-flop are shown in Figure 3.37.

Figure 3.37
Clocked SR flip-flops: (a) positive edge-triggered and (b) negative edge-triggered



The flip-flop shown in Figure 3.37(a) is said to be positive-edge triggered. In a circuit diagram, this is shown with a small triangle or wedge at the clock (CK) input. A negative edge-triggered flip-flop is shown in Figure 3.37(b), where the negative edge-transition is indicated with a small circle and a triangle at the clock input. Figure 3.38 shows a timing diagram for a positive edge-triggered SR gate. Notice how the output Q changes state at the instant of the positive edge transitions of the clock signal and not when the input S and R change states. In reality, the output does not change instantaneously, but there is a small propagation delay on the order of few nanoseconds or less.

Figure 3.38
Timing diagram for a positive edge-triggered clocked SR flip-flop



The truth table for a positive edge-triggered SR flip-flop is shown in Table 3.10. The up arrow (↑) in the clock column refers to the positive edge transition of the clock, while 0 and 1 in that column refer to the clock state. In Table 3.10, an X entry in any of the cells means that the output is not affected by that entry.

Clock	*S*	*R*	$Q_t \rightarrow Q_{t+1}$
↑	0	0	0 → 0
↑	0	0	1 → 1
↑	0	1	0 → 0
↑	0	1	1 → 0
↑	1	0	0 → 1
↑	1	0	1 → 1
↑	1	1	Not Allowed
0,1	X	X	0 → 0
0,1	X	X	1 → 1

Table 3.10

Truth table for a positive edge-triggered SR flip-flop

A common use for the SR flip-flop is to debounce switch input. Example 3.7 illustrates this case. Example 3.8 shows a further application of flip-flops.

Example 3.7 SPDT Switch Debouncing Circuit

Illustrate how an SR flip-flop can be used as a switch debouncer for a SPDT switch.

Solution:

The circuit that does the job is shown in Figure 3.39. The switch leads are connected to the flip-flop inputs through an inverter. When the switch pole is in the upper position, *D* is low and *E* is high. *S* is 1 in this case, and *R* is 0. The flip-flop output *Q* will be 1. As the pole leaves the upper switch position, the switch bouncing at that position will not affect the output of the flip-flop, since *S* becoming 0 does not reset the output. When the switch pole reaches the lower position, the first contact at the position will cause the flip-flop output to be turned off, since *R* will be 1, and *S* will be zero. Any subsequent bouncing at the lower position will not change the output, since *S* is 0. Note that for this circuit to work, the bouncing at the upper switch position should be completed before the pole reaches the lower position, which is the case in reality. Thus, using this circuit, the output of the flip-flop changes with no bouncing as the switch is rotated from the *D* to the *E* position.

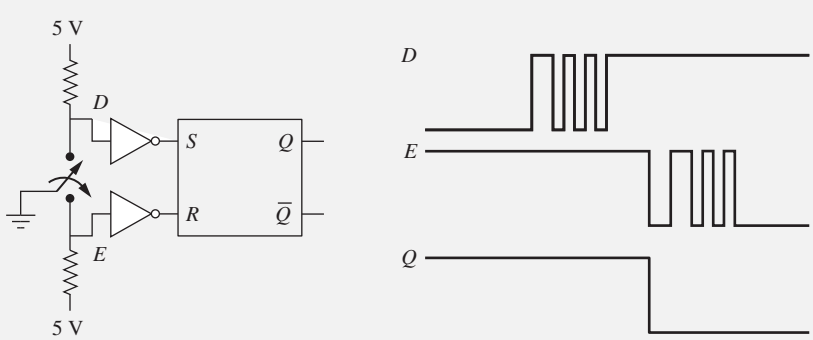


Figure 3.39

Example 3.8 Wire Game

An interesting skill game is to guide a closed ring around a wire path without the wire and the ring touching. Design a circuit that uses an SR flip-flop so that, when the ring and the wire touch, a buzzer is turned on and remains on even when the two no longer touch. The buzzer can be turned off by pressing a reset switch.

Solution:

The circuit that performs this function is shown in Figure 3.40. The ring and wire contact are represented by a NO push-button switch. When the ring and the wire are not in contact, the voltage at the *S* and *R* leads is zero, and the buzzer will be off, since output *Q* will be zero (the flip-flop can be initialized to this state through the reset switch). The instant the ring and the wire contact, the input *S* will be set, causing the buzzer to be tuned on. The buzzer will remain on even if the ring and the wire are no longer in contact. When the NC push-button reset switch is activated and assuming that the ring and wire are not in contact, *R* will be set and S will be reset, causing the buzzer to turn off.

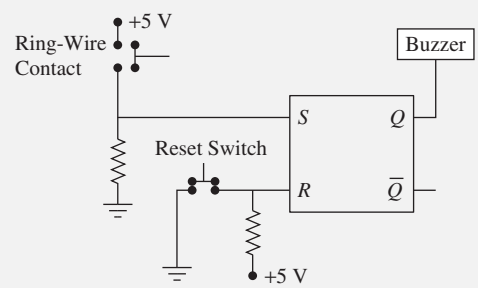
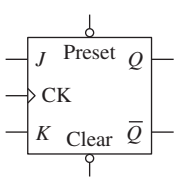


Figure 3.40

Figure 3.41

JK flip-flop



JK Flip-Flop A JK flip-flop is similar to an SR flip-flop, but allows a simultaneous input of $J = 1$ and $K = 1$ similar to $S = 1$ and $R = 1$ in an SR flip-flop. The symbol for a JK flip-flop is shown in Figure 3.41, and a truth table for its operation is listed in Table 3.11. Figure also shows two other input lines for this flip-flop, called *Preset* and *Clear*. Many flip-flops have these additional inputs which can be used to force the output of the flip-flop irrespective of the clock signal. These types of inputs are called *asynchronous inputs*, since they are not synchronized with the clock signal. The *Preset* input sets the output *Q* to 1 or high when it is activated, and the *Clear* input sets the output *Q* to 0 or low. In Figure 3.41, both of these inputs are active low-type (indicated by the small circle at the input lead), which means that the desired action occurs when the input is low. These inputs can be used to initialize the flip-flop output at power-up.

In Table 3.11, an X entry in any of the cells means that the output is not affected by that entry. Notice that when $J = 1$ and $K = 1$, the output will toggle between 0 and 1.

D Flip-Flop A D flip-flop or **data flip-flop** is used to store data and make it available at clock transitions. The symbol of the D flip-flop and its equivalent circuit are shown in Figure 3.42. Due to the use of the clock input, the output only changes at low-to-high clock transitions. The D flip-flop is typically used to implement data

Preset	Clear	Clock	J	K	$Q_t \rightarrow Q_{t+1}$
1	0	X	X	X	0 → 0
1	0	X	X	X	1 → 0
0	1	X	X	X	0 → 1
0	1	X	X	X	1 → 1
0	0	Not Allowed			
1	1	↑	0	0	0 → 0
1	1	↑	0	0	1 → 1
1	1	↑	0	1	0 → 0
1	1	↑	0	1	1 → 0
1	1	↑	1	0	0 → 1
1	1	↑	1	0	1 → 1
1	1	↑	1	1	0 → 1
1	1	↑	1	1	1 → 0
1	1	0,1	X	X	0 → 0
1	1	0,1	X	X	1 → 1

Table 3.11

Positive edge-triggered JK flip-flop truth table

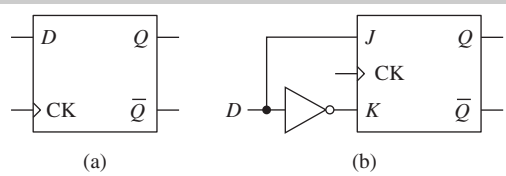


Figure 3.42

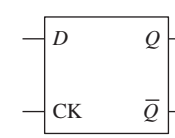
(a) D flip-flop and
(b) its equivalent circuit

registers, which are sets of memory elements that are used to hold information until it is needed. Example 3.9 illustrates this function.

A flip-flop that looks similar to the D flip-flop (but its clock input is not edge triggered) is called a **latch**. The symbol for the latch is shown in Figure 3.43. Notice that there is no triangle shown at the clock lead. With reference to the timing diagram shown in Figure 3.44, the latch flip-flop output changes when the clock signal is high and not at the clock transition. Latches are commonly used to maintain the output in a digital-to-analog converter (see Chapter 5).

Figure 3.43

Latch



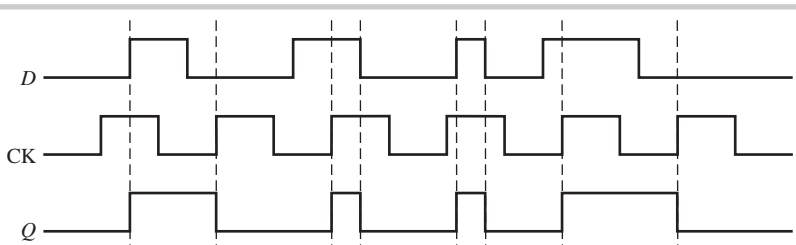


Figure 3.44

Latch Timing Diagram

Example 3.9

Show how a D flip-flop can be used as a 3-bit data register.

Solution:

The circuit to perform this operation is shown in Figure 3.45. Each bit input is connected to the D lead on the D flip-flop. The clock input is combined with a *load* input using an *AND* gate. In this way, the outputs are updated at the low-to-high clock transitions but only when the *load* line is high.

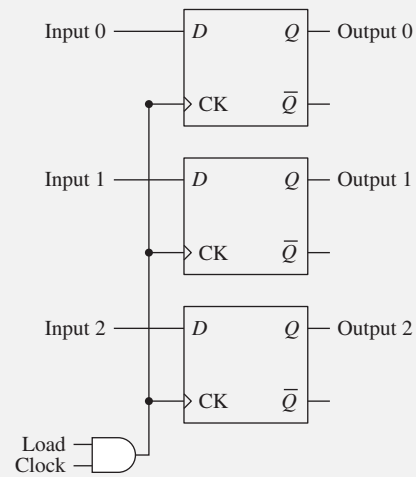


Figure 3.45

T Flip-Flop If the *J* and *K* inputs of the flip-flop are permanently set to 1, and the input is applied at the clock input, we get what is called a 'T' or **toggle flip-flop**. The symbol for a positive edge-triggered T flip-flop and its equivalent circuit are shown in Figure 3.46. The T flip-flop has a single input, and its output (*Q*) changes state (or toggles) at each low-to-high clock transition. The timing diagram for a T flip-flop is shown in Figure 3.47. A characteristic of the T flip-flop is that the output changes its state at a frequency that is half of the input clock frequency. This feature is utilized in the construction of binary counters and frequency dividers.

Figure 3.46

T flip-flop (a) symbol and (b) equivalent circuit

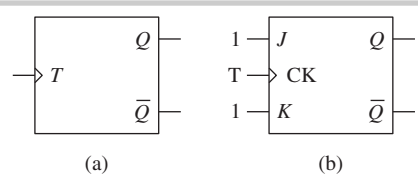
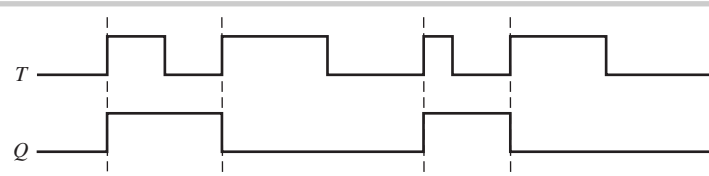


Figure 3.47

T flip-flop timing diagram



As an example of using T flip-flops for counting, consider a 3-bit counter that uses T flip-flops. The circuit for this **binary counter** is shown in Figure 3.48. A

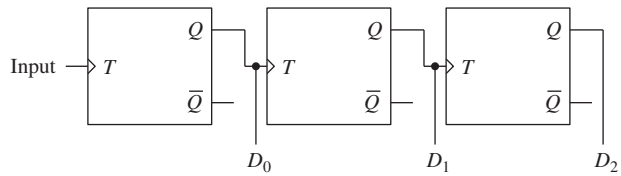


Figure 3.48

3-bit binary counter

clock signal is applied only to the first flip-flop T input, but the output of each flip-flop is fed to the T input of the next flip-flop in the circuit.

Figure 3.49 shows the timing diagram for the output of this circuit when square pulses are applied to the leftmost T flip-flop. The outputs D_0 through D_2 count the applied pulses in a count-down fashion with D_0 being the least significant bit. As seen in Figure 3.49, the counter reads 7 after the first pulse, 6 after the second pulse, and so forth. If we had used a negative edge-triggered T flip-flop (see Problem 3.17), this counter would count in count-up fashion with the counter reading 1 after the first pulse. Note how the frequency of each output line is half the frequency of the previous one. Thus, the D_0 output is a 'divide by 2' line, and the D_1 is a 'divide by 4' output, and so forth. Thus, this circuit can be used for either counting or frequency division.

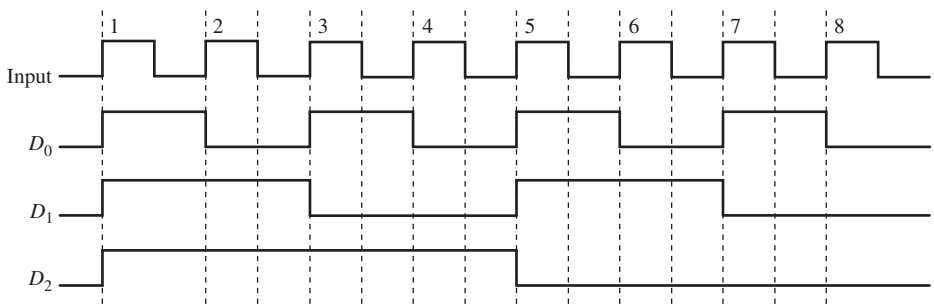


Figure 3.49

Timing Diagram for 3-bit counter

There are many commercially available counter ICs. These include 4-bit binary counters that count from 0 to 15 and decimal (or decade) counters that count from 0 to 9. The 7490 IC is an example of a 0 to 9 **decimal counter** (see Figure 3.50). The signal to be counted is applied to the clock input of the 7490, and the count is available from the output lines labeled Q_A through Q_D as a 4-bit BCD. The 7490 counts from 0000 (0) to 1001(9), and then resets back to 0. At the reset from 1001 to 0000, the Q_D line goes from high to low. Since the clock input on the 7490 is negative edge-triggered, the Q_D line from one 7490 can be fed to the clock input of another 7490 to create a counter that can also count the tens digit. In Figure 3.50, the counter counts from 0 to 999 using three cascaded 7490 ICs. Note that this IC has other inputs and other modes of operation, and the reader should consult the data handbook for detailed information on this IC.

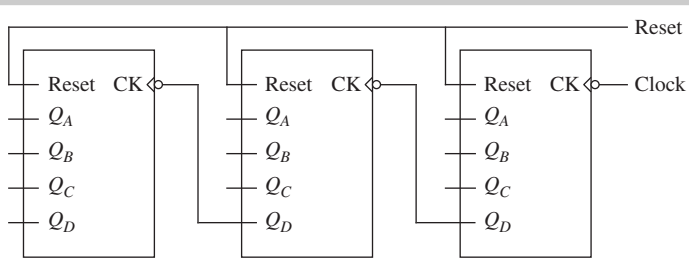


Figure 3.50

A 0 to 999 counter using three 7490 IC

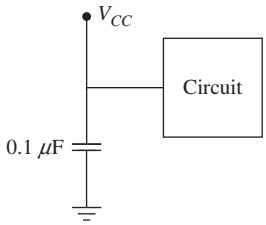
The counter circuit shown in Figure 3.50 is called a **ripple counter**, because the counter output lines for each decade or digit in one IC do not change at the same time as the lines in the next IC. This is because each IC triggers the next IC in series in the order of the connection of the ICs. This creates no problem if the output from each IC is connected to a digital display and is read by human eye, since the 'ripple' effect is too fast to be seen by the human eye. However, if the output lines were connected to other digital circuits, this creates glitches. To circumvent this problem, a synchronous counter should be used. In a **synchronous counter**, all of the digits of the counter change at the same time. This is done by feeding the same clock signal (or the signal to be counted) to the clock input line on each of the cascaded ICs. This causes all of the output lines on all ICs to change at the same clock transition. An example of a synchronous counter IC is the 74160 IC.

There are many other applications for flip-flops. For example, JK flip-flops can be used in serial-to-parallel conversion or in parallel-to-serial conversion.

It is very important in digital circuits to reduce the noise in the power supplied to the circuit. This is done with the help of a **bypass** capacitor. Typically a 0.1μF capacitor is placed between the voltage source and ground on the power line to the circuit, as seen in Figure 3.51. The purpose of the bypass capacitor is to dampen any AC component in the DC power signal.

Figure 3.51

Bypass capacitor



3.8 Circuit Families

Digital circuits are commonly available in two families: transistor-transistor logic (TTL) and complementary metal-oxide semiconductor (CMOS). There are other families, such as emitter-coupled logic (ECL), but they are not as widely used. TTL devices are based on the bipolar junction transistor technology, while CMOS devices are based on the FET transistor technology. Table 3.12 shows the voltage levels for the two families for a 5 V supply. The values define the allowable voltage ranges for the low and high logic states.

Note how the output level range for either logic state is smaller than the corresponding input level range for that state. This is to allow for noise and signal variation in the output voltage values. Table 3.13 compares the two families in several categories. In general, CMOS ICs consume less power than TTL ICs and can operate over a wider voltage supply range, but they can be easily damaged by static electricity, and proper grounding is needed when handling CMOS ICs. In Table 3.13, **gate propagation delay** refers to the time it takes for a gate to switch logic levels from either high to low or from low to high. The propagation delay time from low to high (t_{PLH}) and from high to low (t_{PHL}) are generally not the same, and the largest of the two is used. **Fan out** refers to the number of inputs that can be driven by one output. CMOS devices have a higher fan out than TTL devices. The terms current **sinking** and **sourcing** are commonly used when listing

Table 3.12

TTL and CMOS* voltage levels

	Low State Voltage Range		High State Voltage Range	
Operation	**TTL**	**CMOS**	**TTL**	**CMOS**
Input	0–0.8 V	0–1.5 V	2.0–5.0 V	3.5–5.0 V
Output	0–0.5 V	0–0.05 V	2.7–5.0 V	4.95–5.0 V

***CMOS data is for operation using a 5 VDC supply**

Table 3.13

Comparison between TTL and CMOS families

	TTL	CMOS
Supply Voltage	Tight supply voltage (about 4.50 to 5.50)	Can operate over a wide supply range from 3 to 18 V
Power Consumption	High but power consumption does not increase with signal frequency	Much lower power consumption than TTL, but power consumption increases with frequency
Static Sensitivity	Not sensitive	Very sensitive
Unused Inputs	Can be left floating	Should be tied to ground or to +V
Operating Frequency	Higher than CMOS	Lower than TTL due to MOSFET gate capacitances
Gate Propagation Time	~10 ns	Slower than TTL*
Input Current	High current draw	Very low current draw at gate input
Output Current	Source about 2 mA but can sink about 16 mA	Can sink or source about 4 mA
Fan-out	One output can drive about 10 inputs	One output can drive about 50 inputs

*** Advanced CMOS logic features gate delays of less than 0.1 ns**

specifications about IC. A device is said to be sinking current if the current flows *into* the output gate of the IC device when the output is low, while a device is said to be sourcing current if the current flows *from* the output gate of the IC device when the output is high. As listed in Table 3.13, TTL devices can sink much more current than CMOS devices.

Within each family, there are several sub-families or series of ICs. The different subfamilies are listed in Table 3.14 along with some information about each sub-family. This table is not comprehensive, as there exist over 30 sub-families. Most TTL and CMOS devices are designated using the notation:

mmNNssddp

where

mm is a 2- or 3-letter code for the manufacturer (such as *DM* for National Semiconductor and *SN* for Texas Instruments)

NN is either 74 or 54 and refers to operating temperature range, where 74 is for industrial applications (0 to 70°C), and 54 is for military applications (−55 to 125°C).

ss refers to the sub-family (such as *LS* for low-power Schottky)

dd is a 2- to 4-digit device number (such as 08 for an AND gate)

p is the designation for the type of package in which the device is available (such as *N* for plastic DIP)

For example, the *SN74LS08N* is a TTL AND gate of the low-power Schottky series designed for commercial applications and manufactured by Texas Instruments in the DIP plastic package type. Note that TTL and CMOS devices have been in existence for many years, and several of the sub-families listed in Table 3.14 are now obsolete but are mentioned for reference.

Table 3.14

Listing of TTL and CMOS circuit families

TTL				
Series	**Designation**	**Example Device**	**Feature**	**Notes**
Regular TTL	—	7408	Original TTL series. Has high power consumption	Obsolete
Low-Power TTL	L	74L08	Lower power than regular TTL but also lower speed	Obsolete
High-Speed TTL	H	74H08	Double the speed and power of the regular TTL series	Obsolete
Schottky TTL	S	74S08	Uses more power than regular TTL but is faster	Obsolete
Low-Power Schottky TTL	LS	74LS08	Lower power version of S series	Very commonly used
Advanced Schottky TTL	AS	74AS08	Faster than S series with lower input current requirements	
Advanced Low-Power Schottky TTL	ALS	74ALS08	Very low power dissipation	
Fast TTL	F	74F08	Lower power than S and LS series	
CMOS				
Metal Gate	C	74C08	Pin-compatible with TTL	Can use 3 to 15 V power supply
High-Speed Silicon Gate	HC	74HC08	Pin-compatible with TTL and has same speed as the 74 LS	Requires 2 to 6 V power supply. Can drive 74LS devices but not driven by them
High-Speed Silicon Gate TTL Compatible	HCT	74HCT08	Pin compatible with TTL	Requires 5 $\pm$ 0.5 V supply. Can be interfaced with 74LS devices for both input and output
Advanced CMOS	ACT	74ACT08	Inputs are TTL-voltage compatible	Requires 5 $\pm$ 0.5 V supply

In data sheets for TTL and CMOS devices, several parameters are defined:

V_{CC} Supply voltage

V_{OL} Output voltage when the output is LOW

V_{OH} Output voltage when the output is HIGH

V_{IL} Input voltage when the input is LOW

V_{IH} Input voltage when the input is HIGH

I_{OL} Output current when the output is LOW

I_{OH} Output current when the output is HIGH

I_{IL} Input current when the input is LOW

I_{IH} Input current when the input is HIGH

As an example, Table 3.15 lists the values of these parameters for the SN74LS08 and the SN74HCT08 AND gates.

Parameter	SN74LS08	SN74HCT08
V_{CC}	4.75–5.25 V	4.5–5.5 V
V_{OL} (max)	0.5 V	0.1 V
V_{OH} (min)	2.7 V	4.4 V
V_{IL} (max)	0.8 V	0.8 V
V_{IH} (min)	2 V	2 V
I_{OL} (max)	8 mA	4 mA
I_{OH} (max)	−0.4 mA	−4 mA
I_{IL} (max)	−0.4 mA	±1 μA
I_{IH} (max)	20 μA	±1 μA

Table 3.15

Voltage and current parameters for an AND gate

Note that for current, the convention is that the current entering a device (sinking) is positive, and a current leaving a device (sourcing) is negative. By examining the values of the input and output currents at low and high logic states, one can determine how many inputs can be connected to the output of one gate or fan out. For example, for the TLL AND gate in Table 3.15, one output can drive up to 20 TTL AND inputs (I_{OL}/I_{IL} = 8 mA/0.4 mA = 20). Table 3.15 shows that for the CMOS AND gate, the input current is significantly smaller than that for the TTL gate.

TTL devices are available with different types of outputs. These include totem-pole, open-collector, and tristate. **Totem-pole** is the most commonly used construction. The output gate has two transistors stacked on the top of each other, as seen in Figure 3.52(a) and, hence, the name totem pole. When the output is high,

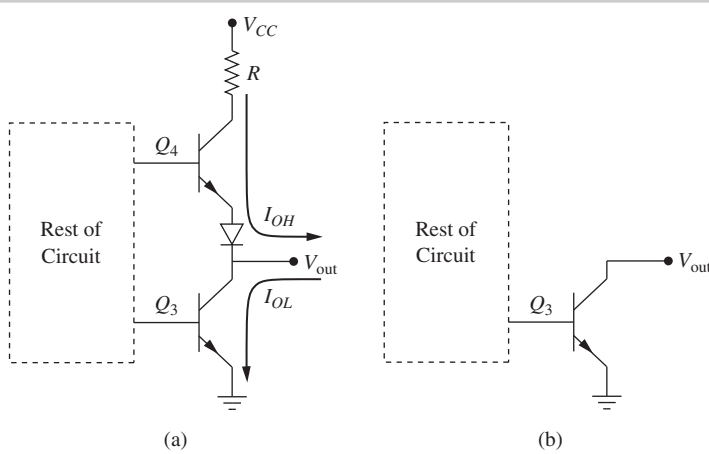


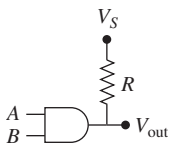
Figure 3.52

(a) Totem-pole output and (b) open-collector output

transistor Q_4 is ON and transistor Q_3 is OFF. In this case, current flows through Q_4 and out of the device. Thus, when the output is high, the gate is sourcing current, and I_{OH} is negative. When the output is low, transistor Q_4 is off, and transistor Q_3 is on. Current flows into the output gate through transistor Q_3. Thus, the output gate is sinking current, and in this case, I_{OL} is positive.

Figure 3.53

Wiring of open-collector AND gate



In **open-collector** configuration, an external 'pull-up' resistor needs to be connected to the output gate. This situation is similar to the totem-pole configuration of Figure 3.52(a) with transistor Q_4 not present (Figure 3.52(b)). Figure 3.53 shows a typical wiring of an open-collector AND gate. When the transistor Q_3 is ON, the output voltage of the gate will be at a low logic level or close to 0 V (about 0.2 V). When transistor Q_3 is OFF, the output voltage of the gate will be pulled to the supplied voltage (V_S). Note that with open-collector output, the high logic-state voltage output is not limited to the IC V_{CC} voltage but could be any voltage higher or lower than V_{CC}. An example of open collector AND gate is the DM74LS09 IC. ICs with open-collector output are typically used to interface ICs from different logic families (such as TTL and CMOS).

In **tristate** output, the gate has an additional input called *enable*. When the enable input is low, the output can be either low or high, depending on the input applied to the gate. When the enable input is high, the output is disconnected from the rest of the circuit. The gate will have a high output impedance in this state. Three-state output is used in cases where data from several digital devices is transferred on a common line or bus. The enable signal is then used to connect/disconnect these devices from the bus. Examples of such devices include buffers, flip-flops, and memory chips.

The question that arises is whether devices from different families can be interfaced together. The answer depends on the sub-family of the device. In general, most TLL and CMOS sub-families cannot be directly interfaced due to voltage-level incompatibility, but the CMOS 74HCT and the TTL 74LS sub-families can be mixed together without using any additional components. For interfacing a TTL output to a CMOS input where there is a voltage incompatibility, a pull-up resistor is added to the output of a TTL device before it is interfaced with the CMOS device. This insures that the high-output voltage of the TTL device (V_{OH}) is higher than the high-input voltage (V_{IH}) of the CMOS device. The pull-up resistor resistance value should be selected such that the I_{OL} for the TTL device is not exceeded. A CMOS device from the HC or HCT series can drive a single LS device. For driving multiple LS devices, a buffer is inserted between the CMOS output and the TTL inputs to meet the current requirements.

3.9 Digital Devices

In addition to logic gates such as AND or NOR integrated circuits, many specialized integrated circuits are commercially produced. We previously discussed a few of them (such as digital counters and multiplexers). This section discusses another commonly used digital device; the 555 timer chip. The **555 timer chip** (such as the NE555 8-pin chip from Texas Instruments) is an integrated circuit that uses a transistor, resistors, flip-flops, comparators, and capacitors to produce a variety of clock signals, including a fixed pulse, a periodic signal, and a frequency dividing signal. The NE555 can operate over a wide voltage supply range (5 to 15 VDC). With a 5 V supply, it has a TTL-compatible output that can sink or source up to 200 mA.

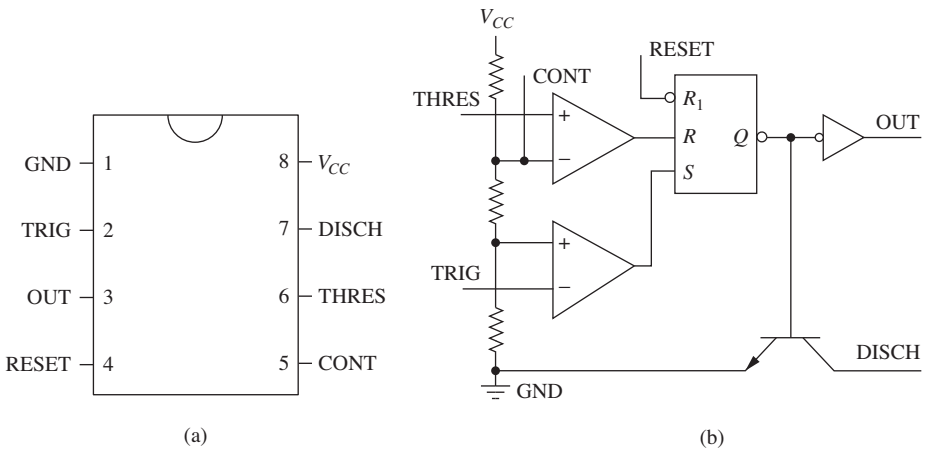


Figure 3.54

(a) Pin layout and (b) a functional diagram of the NE555 timer chip

The pin layout and a functional block diagram of the NE555 timer are shown in Figure 3.54.

The chip operation is controlled by the inputs applied to the trigger (TRIG) and threshold (THRES) pins. Each of these inputs is connected to a two-input comparator. The comparator outputs are attached to the set (S) and reset (R) inputs of the SR flip-flop. The trigger and threshold inputs are irrelevant if the RESET input is low. When the RESET input is high, the timer output changes according to the trigger and threshold levels. The functional operation is shown in Table 3.16. The trigger input sets the timer output to high if the trigger voltage level is below one-third of the supply voltage (V_{CC}) regardless of the voltage level applied to the threshold input. When the trigger voltage is larger than one-third of the supply voltage, the timer switches from high to low if the threshold voltage exceeds two-thirds of the supply voltage and maintains its output if the threshold voltage is below two-thirds of the supply voltage.

Table 3.16

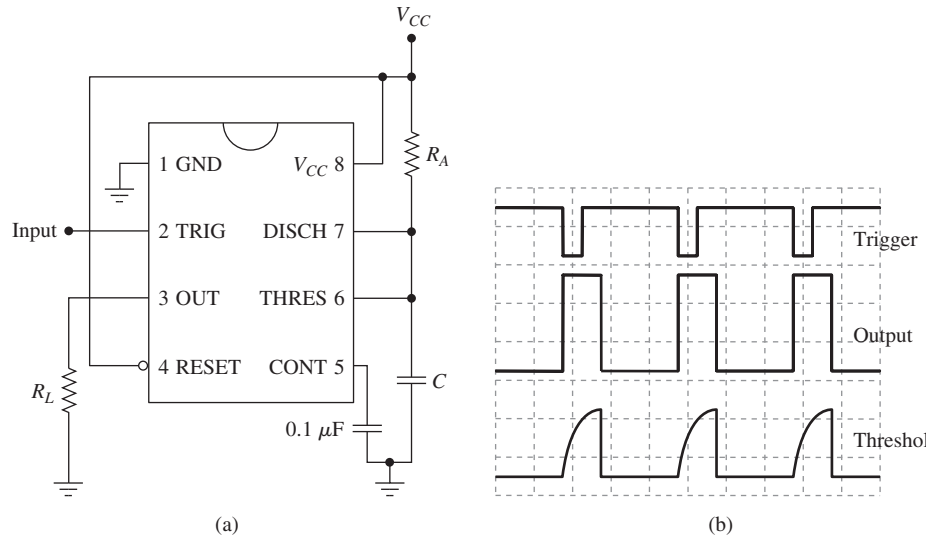
Functional operation of the 555 timer chip

RESET	Trigger Voltage	Threshold Voltage	Timer Output	Discharge Switch
Low	X	X	Low	On
High	$< 1/3\ V_{CC}$	X	High	Off
High	$> 1/3\ V_{CC}$	$> 2/3\ V_{DD}$	Low	On
High	$> 1/3\ V_{CC}$	$< 2/3\ V_{DD}$	No change	No change

While the NE555 timer chip has several modes of operation, we will focus on two of them here. These are the **monostable** (or fixed-pulse generation) mode and the **astable** (or self-generating periodic signal) mode. In the monostable mode, the pulse properties are controlled by one external resistor and one capacitor. In the astable mode, two external resistors and one capacitor control the duty cycle and the frequency of the timing signal. The wiring diagram for monostable operation is shown in Figure 3.55(a), and the timing diagram is shown in Figure 3.55(b). Here the output of the timer is controlled by the input signal applied to the trigger input. Initially, the internal SR flip-flop output is OFF, and the external capacitor C is

Figure 3.55

(a) Wiring diagram for monostable operation and (b) timing diagram



held in the uncharged state by the internal transistor inside the timer. When a falling-edge pulse signal is applied to the trigger input with a voltage level less than one-third of the supply voltage, it causes the internal SR flip-flip to turn ON, and the timer output will be high. This happens because the internal, lower-comparator output will be high, which will set the S input of the internal RS flip-flop to high. When the timer output turns high, the internal transistor is not conducting. This causes the capacitor C to charge through the resistor R_A, since the DISCH pin is not connected to ground voltage in this case. When the voltage level at the capacitor C has reached two-thirds of the supply voltage, the internal flip-flop will reset, because the internal, upper-comparator output will be high and the internal lower comparator output will be low (provided that the trigger input has returned to high at this point). This causes the output of the timer to go low, and the voltage across the capacitor C will discharge through the internal transistor. This cycle is repeated for every application of a falling-edge trigger pulse. The **output pulse duration** is approximately given by

$$T_H = 1.1\, R_A\, C \tag{3.12}$$

Note that trigger signal duration has to be smaller than the output pulse duration. Otherwise, the timer output will remain high.

The wiring diagram for astable operation is given in Figure 3.56(a), and the timing diagram is shown in Figure 3.56(b). A second resistor (R_B) is added to the monostable circuit of Figure 3.55(a), and the threshold and trigger inputs are connected together causing the timer to self trigger. In this configuration, the capacitor C charges through the resistors R_A and R_B, and discharges through R_B only. When V_{CC} is first turned on, the capacitor C is discharged, and the trigger input voltage level is zero. The timer output will be high. When the voltage across the capacitor reaches two-thirds of the supply voltage, the internal SR flip flop resets, the timer output switches to low, and the voltage across the capacitor C discharges through the internal transistor. In the astable mode, the capacitor C alternates between charging and discharging states with the charging time is a function of the

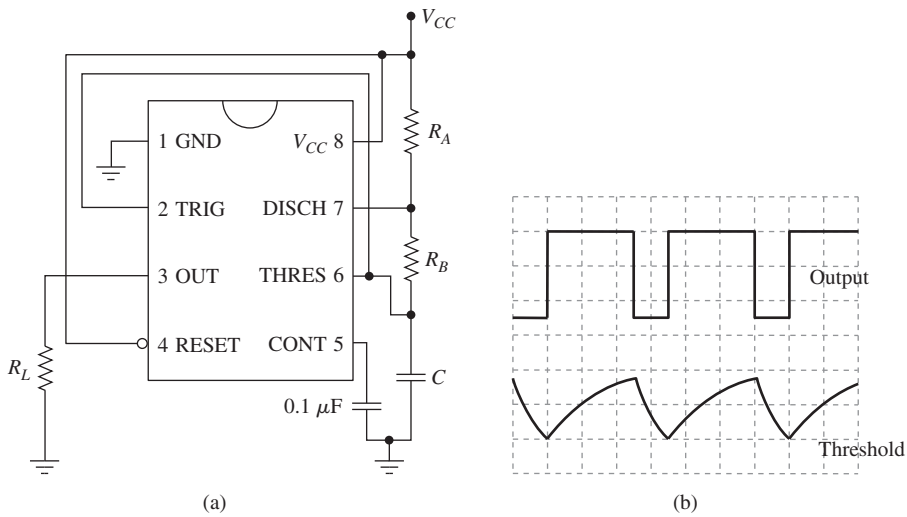


Figure 3.56

(a) Wiring diagram for astable operation and (b) timing diagram

values of the resistors R_A and R_B and the capacitor C, and the discharging time is a function of the resistor R_B and the capacitor C.

The on-time period (T_H) and the off-time period (T_L) are given by

$$T_H = 0.693(R_A + R_B)C \tag{3.13}$$

and

$$T_L = 0.693\,R_BC \tag{3.14}$$

Example 3.10 illustrates the selection of components for a 555 timer.

Example 3.10 555 Timer

Design a 555 timer circuit to produce a timing signal at a frequency of 1 kHz, and a duty cycle of 75%.

Solution:

For 75% duty cycle, we obtain from Equations (3.13) and (3.14):

$$0.75 \times 10^{-3} = 0.693(R_A + R_B)C$$
$$0.25 \times 10^{-3} = 0.693\,R_BC$$

Dividing these equations, we obtain

$$3R_B = R_A + R_B$$

Selecting C as 0.15 μF and solving, we obtain R_B as 2405 Ω and R_A as 4810 Ω. The 2.4 kΩ and the 4.8 kΩ are standard resistor values which can be used.

3.10 H-Bridge Drives

A very common application of transistors is to construct drivers to drive motors. One such circuit is the H-bridge driver circuit, which is commonly used to drive motors in both directions. The circuit looks like the letter 'H' in circuit schematics, so it is called an H-bridge. An H-bridge circuit is constructed using four switching elements that are situated at the corners of the 'H' as the main components. The switching elements used are transistors. The transistors can be of the BJT type or MOSFET type, but depending on the transistors used, the circuit will have different power ratings. To understand the operation of an H-bridge circuit, consider first the simple schematic shown in Figure 3.57, which shows four switches connected to a DC motor. Let us assume that for this motor to turn clockwise, a positive voltage needs to be applied to the left lead, and the right lead should be grounded. This can be achieved by closing switches 1 and 4 and keeping switches 2 and 3 open. This causes current to flow from the power source to ground through switch 1, the motor leads, and switch 4. If we want to rotate the motor in a counterclockwise fashion, then switches 2 and 3 should be closed, and switches 1 and 4 should be open. This reverses the voltage polarity across the motor and causes the motor to rotate in the opposite direction. Obviously, closing switches 1 and 3 or 2 and 4 at the same time should not be done, as this will lead to a short circuit.

Figure 3.57

H-bridge circuit using switches

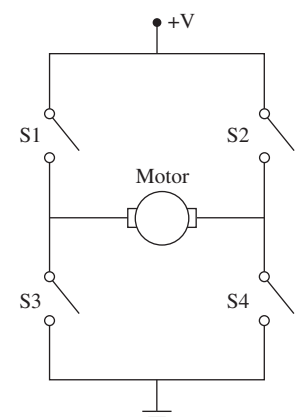
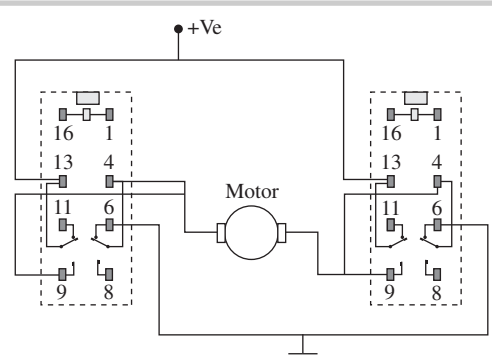


Figure 3.58 shows an implementation of the H-bridge circuit using the G5V-2 Omron® relay as the switching element. The wiring for the relay coil is not shown in the figure. Notice that using two relays, there are four states of operation for this

Figure 3.58

H-bridge implementation using DPDT relays



circuit. When both relays are OFF, the switch configuration is as shown in the figure, and the motor will not operate. Similarly, if both relays coils are ON, the motor still will not operate. In the third state when the left relay is ON and the right relay is OFF, the motor will rotate in one direction. In this case, a positive voltage is applied to the left lead of the motor through pin #9 on the left relay, while the right lead of the motor is grounded through pin #6 on the right relay. In the fourth state, the left relay is OFF, and the right relay is ON. The polarity of the voltage applied to the motor will be opposite to that of state three, and the motor will rotate in the opposite direction.

While Figure 3.58 shows how an H-bridge can be implemented using electromechanical relays, typical H-bridge circuits are implemented using transistors due to the fast switching time of transistors compared to electromechanical relays. Figure 3.59 shows such an implementation using MOSFET power transistors. This L6203 chip from STMicroelectronics supplies up to 1 A current, which is enough to drive small brush type DC motors. The supply voltage can be up to 48 V. The L6203 H-bridge operates as follows. There are two input signals *IN1* and *IN2* and one *ENABLE* input signal. The *IN1* and *IN2* signals are used to select the particular leg of the H-bridge. The *ENABLE* input has to be high for the H-bridge to operate. The motor leads are connected to *OUT1* and *OUT2* leads. If *IN1* is high and *IN2* is low, *OUT1* will be at the supply voltage (*Vs*), and *OUT2* will be grounded. In this case, the upper-left and lower-right transistors will be conducting, and the upper-right and lower-left transistors will not be conducting. Similarly, if *IN1* is low and *IN2* is high, *OUT1* will be grounded, and *OUT2* will be at the supply voltage (*Vs*), thus reversing the polarity of the voltage supplied to the motor.

Notice that the chip has a temperature-sensor circuit that shuts down the H-bridge if the temperature exceeds a preset value (typically 150°C). The flyback diode that is placed between the source and drain leads of the MOSFET transistor is called an *intrinsic diode* and is built-in as part of the transistor. Its purpose is to protect the transistor when the transistor switches its state with inductive loads attached to the chip. Section 8.2.4 has additional information on using H-bridge drives in DC motor control.

H-bridge drives are used to control various types of actuators, including brush DC, brushless DC, and stepper motors. H-bridge drivers are typically packaged with additional components to produce what is called a servo drive, which is used in controlling actuators. Actuators and servo drives are discussed in Chapter 8.

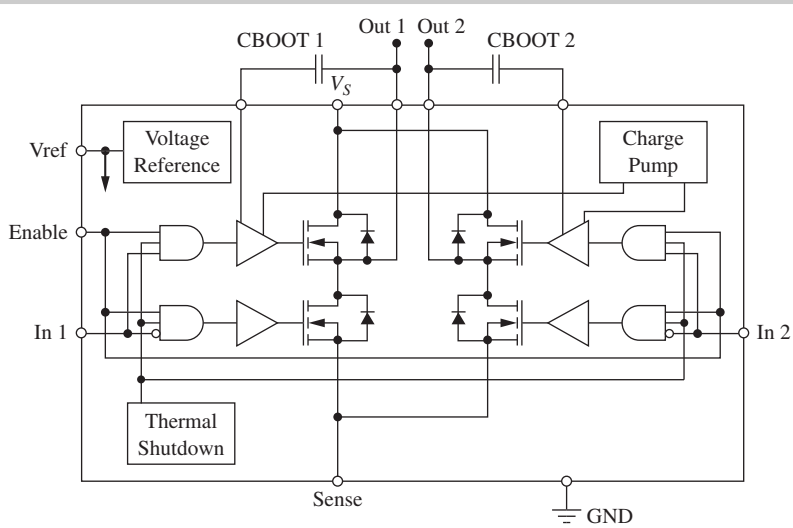


Figure 3.59

Block diagram of the L6203 H-bridge
(STMicroelectronics P/N L6203 Datasheet)

3.11 Chapter Summary

This chapter discussed the operation of semiconductor electronic devices (such as diodes, thyristors, and transistors) that are used in many circuits and devices for switching or amplification purposes. These devices are examples of solid-state switches, which are devices in which the switching action is caused by non-mechanical motion and is due to the change in the electrical characteristics of the device. Solid-state devices do not obey Ohm's law. A diode is a directional element that allows current to flow in one direction and is a two-terminal device. There are several varieties of diodes, including regular, Zener, LEDs, and photodiodes. A thyristor is a three-terminal semiconductor device that behaves like a diode but with an additional terminal called a gate that controls its operation. A transistor is a three-terminal device and has three states of operation. These are the off state, the linear state, and the saturation state.

This chapter discusses the bipolar junction transistors (BJTs) and the metal-oxide semiconductor field effect transistors (MOSFETs). The BJT is a current-controlled device, while the MOSFET is a voltage-controlled device. Transistors form the basis for the construction of digital circuits. Both combinational and sequential logic circuits were discussed. In a combinational logic circuit, the output is not dependent on the history of the input, and the circuit uses rules of mathematical logic to generate the output. On the other hand, in a sequential logic circuit, the past sequence of the input is important and determines the output of the system.

This chapter also discussed application of digital circuits in the design of devices such as digital counters, multiplexers, and timers. The two most commonly available families (transistor-transistor logic (TTL) and complementary metal-oxide semiconductor (CMOS)) for digital circuits were discussed along with information on how to interface them. This chapter discussed the 555-timer chip. It also discussed the H-bridge driver circuit, which is commonly used to drive motors in both directions and is based on the use of transistors as the switching elements.

Questions

3.1 What is the functional difference between a normal diode and a Zener diode?

3.2 What is a thyristor?

3.3 What is the difference between a relay and a transistor?

3.4 What are the three states of operation of a BJT?

3.5 How does a phototransistor operate?

3.6 Name one major difference between a BJT and a MOSFET.

3.7 What is meant by an open-collector output circuit? Why do manufacturers make devices with such an output type?

3.8 What information does a timing diagram give?

3.9 What is the difference between combinational and sequential logic circuits?

3.10 What is a Karnaugh map?

3.11 Why do many digital circuits have clocked input?

3.12 Explain the function of a multiplexer.

3.13 What is the difference between an SR and a JK flip-flop?

3.14 How is a latch different from a D flip-flop?

3.15 What is a T flip-flop?

3.16 List the names of digital circuit families.

3.17 Define what is meant by 'fan out'.

3.18 List the different output methods of digital circuits.

3.19 Define current sinking and current sourcing.

3.20 Name two modes of operation of the 555 timer chip.

3.21 For what purpose is an H-bridge driver circuit used?

PROBLEMS

P3.1 Consider the half-wave diode rectifier circuit shown in Figure 3.2. Assume a V_F value of 0.6 V. Plot the output voltage of the circuit for an AC sinusoidal signal with the following amplitudes.

a. 0.5 V

b. 2 V

c. 5 V

P3.2 Consider the signal conditioning op-amp circuit shown in Figure P3.2 with an ideal diode in the feedback loop. Show that, for the case $V_i > 0$ and $V_o < 0$, the input–output relationship is given by

$$\frac{V_o}{V_i} = -\frac{R_2R_3}{R_1(R_2 + R_3)}$$

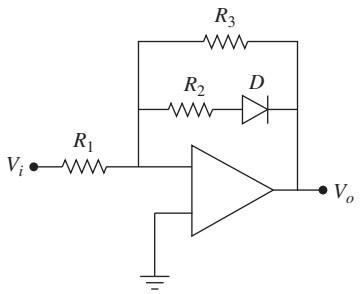


Figure P3.2

P3.3 Draw a circuit that uses a solar cell (can be represented as a voltage source) with a small output current to turn on a lamp (that uses a much larger current). We want the brightness of the lamp to be controllable by the amount of light received by the solar cell.

P3.4 Consider the BJT circuit shown in Figure P3.4 where $V_{CC} = 15$ V, $R_C = 5$ kΩ, $I_B = 40$ μA, and $\beta = 70$. Determine I_C and V_{CE}.

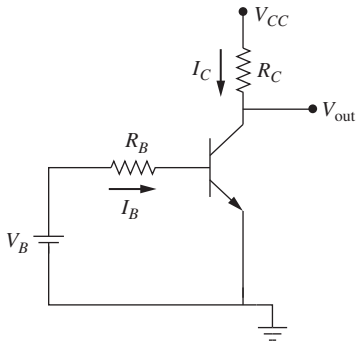


Figure P3.4

P3.5 Figure P3.5 shows a BJT transistor circuit with a bias current. Determine the voltages at points 1 and 2 in the circuit. Let $\beta = 50$, $V_{BE} = 0.6$ V, $R_C = 2$ kΩ, $R_E = 100$ Ω, $R_1 = 2$ kΩ, $R_2 = 200$ Ω, and $V_{CC} = 10$ V.

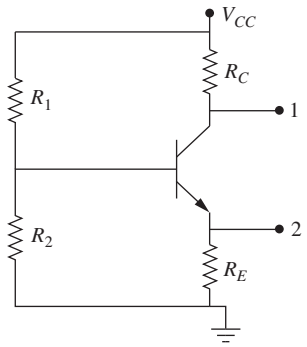
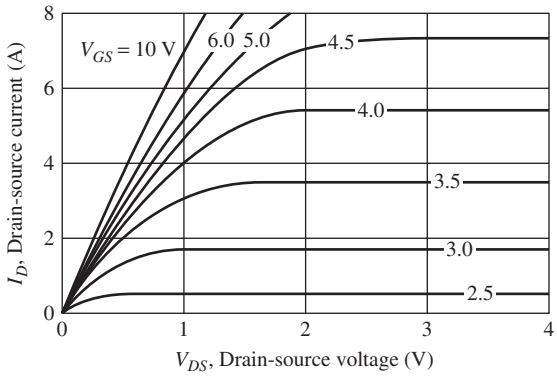


Figure P3.5

P3.6 Figure P3.6(a) shows the output characteristics of a power MOSFET transistor with a $V_T = 2$ V. Assume $V_{DD} = 5$ V. Determine the maximum I_D current in the circuit in Figure P3.6(b) for the following cases.

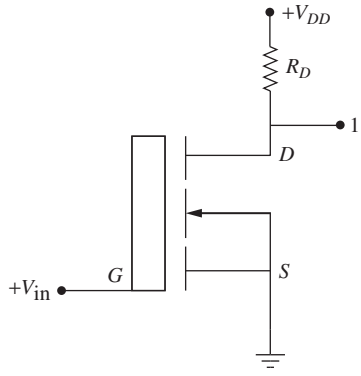
a. $V_{\text{in}} = 1$ V

b. $V_{\text{in}} = 4$ V



(a)

(Courtesy of Fairchild Semiconductor, South Portland, ME)



(b)

Figure P3.6

P3.7 Draw a circuit to show how to use the proximity sensor shown in Figure 3.20 to turn on the coil of the small relay shown in Figure 2.41. Assume that the proximity sensor uses a 24 VDC supply. Specify the value of all the resistors that are needed. Make the coil current compatible with the relay characteristics in Table 2.5.

P3.8 Design a circuit that uses the 2N3904 transistor to activate the coil of the small relay shown in Figure 2.41. Design your circuit so that it is compatible with the 2N3904 and the relay characteristics.

P3.9 Using only NAND gates, draw a circuit that operates as follows.

a. AND gate

b. OR gate

P3.10 Using the Boolean rules, simplify the following expressions.

a. $Q = A.B + \overline{A}.B + \overline{A}.C$

b. $Q = A.B + A.C + A.\overline{B}.C$

P3.11 Draw a circuit realization of the output Q of Example 3.6.

P3.12 An overheating monitoring system uses three digital temperature sensors operating in ON/OFF mode. The output of each sensor is turned ON when the temperature exceeds a specified value and is OFF otherwise. Design a combinational logic circuit to process the output from these sensors such that the circuit output should go high when the output of *any two* of the three temperature sensors goes high.

P3.13 Redo Problem 3.12, but assume the system uses four temperature sensors and the circuit output should go high when the output of *any three* of the four temperature sensors goes high.

P3.14 Draw a combinational logic circuit that implements a four-channel multiplexer which uses two input lines to select the input channel to be connected to the output.

P3.15 Complete the timing diagram in Figure P3.15 for a positive edge-triggered JK flip-flop.

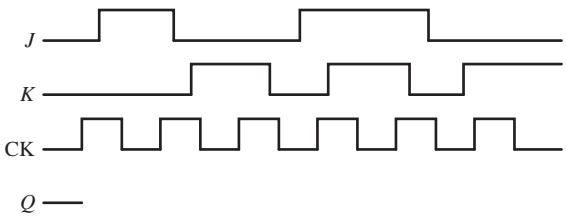


Figure P3.15

P3.16 Referring to the data sheet for the 7490 decade counter IC, draw a circuit that shows how a single 7490 IC can be used as a divide-by-5 counter.

P3.17 Draw the timing diagram for the counter circuit shown in Figure 3.48, but assume that a negative edge-triggered T-flip flop was used.

LABORATORY/PROGRAMMING EXERCISES

L/P3.1 Build the common-emitter transistor circuit shown in Figure 3.15. Use $R_C = 1\ \text{k}\Omega$ and $R_B = 1\ \text{k}\Omega$. Starting from zero, increase the voltage V_{in} in increments of 0.1 V over the range of 0 to 2 V and measure V_{CE}. Change R_B to 10 kΩ, and repeat the procedure. At what input voltage did the transistor turn on in each case? How do the measured results agree with theory? Use a V_{CC} of 5 V.

L/P3.2 Build the wire-game circuit discussed in Example 3.8 that uses an SR flip-flop.

L/P3.3 Build the circuit shown in Figure L/P3.3 that uses a NO push-button switch and a 555-timer chip which act as a bounceless switch. Pressing the switch should cause the circuit to produce a clean, single pulse. The pulse duration is a function of the resistor and capacitor values used in the circuit. Use 1 μF for C_1 and 0.1 μF for C_2. Try 100 kΩ, and 1 MΩ values for R_1, and measure the duration of the output pulse.

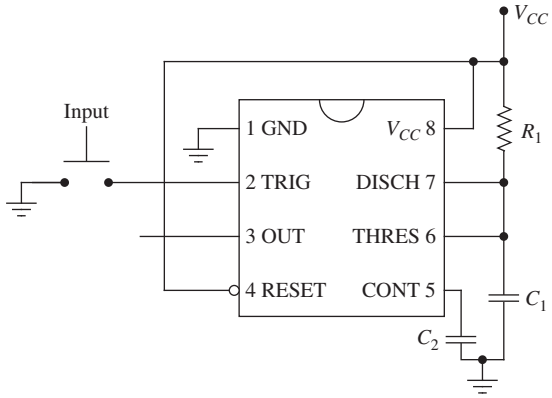
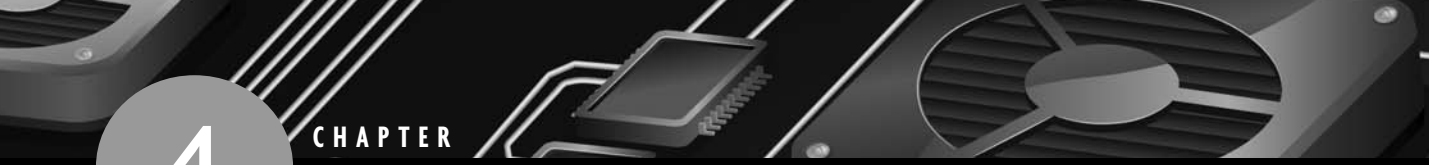


Figure L/P3.3



CHAPTER 4

Microcontrollers

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- Write and interpret data in different numbering systems
- Explain the difference between microprocessors and microcontrollers
- Explain the process of programming a microcontroller
- Explain the basic components of a microcontroller
- Develop code for performing basic operations (such as digital I/O, A/D, timing, and PWM)
- Explain advanced features (such as serial communication, and interrupts)
- Develop code in PIC-C language for programming microcontrollers
- Develop code to incorporate interrupts into programs
- Explain the purpose and usage of assembly language programming and develop code for incorporating it with C-language

4.1 Introduction

This chapter focuses on microprocessors and microcontrollers. Unlike combinational and sequential circuits, which were covered in the previous chapter, microcontrollers and microprocessors offer a flexible but complex method to implement control logic. The flexibility comes from the fact that the control logic is software-based and can be changed by changing the software or control program. The complexity comes from the fact that a typical microprocessor or microcontroller is made up of a very large number of digital circuits. Microcontrollers are widely used for control applications in vehicles, toys, appliances, and telecommunication devices. Microcontrollers are also known as embedded controllers, since they normally are not seen in the control application. Due to their small size, microcontrollers do not have many of the components that a typical PC has (such as a display device, a keyboard or a mass-storage device).

This chapter discusses the use and programming of microcontrollers in detail. The objective of this chapter is to give the reader a complete coverage of the features and capabilities of a typical microcontroller. To understand the operation details and the performance characteristics of microcontrollers, this chapter starts by discussing different numbering systems. This is followed by some background information about microprocessors and microcontrollers. This chapter then discusses different types of memory and buses before it continues with the details of using and programming a microcontroller.

This book focuses on PIC-type microcontrollers made by Microchip, Inc. In particular, it focuses on the PIC16F690 microcontroller, which is a mid-range microcontroller that supports digital input/output (I/O), analog-to-digital (A/D) conversion, pulse-width modulation (PWM), and serial interfacing. It also covers the PIC18F4550 microcontroller, which has more advanced features. This chapter covers the basic elements of a microcontroller (such as clock sources, different memory areas, and basic interface devices) and how to program a microcontroller using the PIC-C language. It also covers advanced features on a microcontroller (such as timers, the watchdog timer, and interrupts). Coverage of assembly language also is included. This chapter has code examples in PIC C-language. For further reading on PIC microcontrollers, see [10-11].

4.2 Numbering Systems

Data can be represented using different numbering systems. For everyday operations, we use the decimal system, which has as its basis ten digits (0, 1, 2, . . . , 8, 9). For computer operations, other numbering systems are used (such as the binary and hexadecimal systems). This section will discuss the representation of numbers using different numbering systems.

4.2.1 Decimal System

In the decimal or base-10 system, a number is represented as a combination of any of the base-10 digits that are used in that system. The value of the number is found by summing the product of each digit in the number multiplied by ten raised to the power of the location of that digit in the number where the rightmost digit has a location of zero and increases by 1 for every place when moving to the left. For example, the value of the number 763 is found from

$$763 \Rightarrow 7 \times 10^2 + 6 \times 10^1 + 3 \times 10^0$$

While the decimal system is convenient for financial, scientific, and everyday mathematical operations, it is not convenient to use when representing numbers that are used in computer systems.

4.2.2 Binary System

The binary or base-2 system uses two digits (0 and 1) to represent numbers. This is similar to how data is stored inside the computer memory. Similar to the decimal system, the value of the number represented by this system is equal to the sum of the product of each digit multiplied by two raised to the power of the location of that digit. For example, the binary number 10110 is equivalent to the decimal number 22. This can be seen by writing

$$10110 \Rightarrow 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \Rightarrow 22$$

To convert from decimal to binary representation, we start by dividing the decimal number by the largest power of 2 that it can be divided by. We then take the remainder and divide by the next-highest power of 2 possible. We repeat this process until the remainder is zero. For example, consider converting the decimal number 59 into binary representation (see Example 4.1).

Example 4.1 Decimal-to-Binary Conversion

Represent the decimal number 59 in binary form

Solution:

We start by dividing 59 by 32, which is the largest power of 2 that we can divide 59 into. We take the remainder, which is 27, and then divide by the next-largest power of 2, which in this case is 16. The remainder after the second division operation is 11. This process is continued until we get a remainder of 0 when we divide 1 by 1. The details are shown here, where r is the remainder after each division.

59/32 = 1 r 27
27/16 = 1 r 11
11/8 = 1 r 3
3/4 = 0 r 3
3/2 = 1 r 1
1/1 = 1 r 0

The binary equivalent is then formed by arranging the quotients to form the binary number, so 59 ⇒ 111011.

The smallest unit of storage in a computer system is the **bit** (which comes from combining letters from the words **b**inary and dig**it**). A single bit can store either 0 or 1. A group of eight bits is called a **byte**, and two bytes are called a **word**. When dealing with binary numbers, the rightmost bit of a binary number is called the **least significant bit (LSB)**, since it represents the smallest power of 2. The leftmost bit is called the **most significant bit (MSB)**, since it represents the largest power of 2. The LSB is referred to as bit zero, the bit adjacent to it as bit 1, and so forth. Binary digits can be added in the same fashion as decimal digits with a carry of 1 to the next-higher order bit if the sum of the two bits is 2. As an example, the binary sum of 0011 and 0010 is 0101. Here, when 1 was added to 1 in the first bit location, we get a zero with a carry of 1 to the second bit location. When referring to computer memory, the term 1K is normally used, and 1K of memory is actually 1024 bytes. Similarly, 64K of memory is 65536 or 64 × 1024. The different number of combinations that can be stored in an n-bit wide memory location is 2^n. Thus, a byte can store 256 different combinations of 0 and 1. If these values are limited to unsigned integers, then all numbers from 0 to 255 can be stored in a byte. Similarly, a 16-bit memory location can store 65536 different combinations.

4.2.3 Hexadecimal System

When evaluating the contents of a large memory location (such as a 32-bit field), it is more convenient if we can write the values of each 4-bit into one digit. This can be done using the hexadecimal or base-16 system, which uses sixteen digits to represent numbers. The first ten digits are the same as the decimal digits 0 through 9, while the last six digits (10, 11, 12, 13, 14, and 15) are represented using the letters A, B, C, D, E, and F, respectively. The value of the number represented by the hexadecimal system is found in a similar fashion to that of the decimal and binary systems with the exception that the base used in this case is 16. Table 4.1 shows the numbers 0 through 15 in decimal, hexadecimal, binary, and binary coded decimal (BCD) systems. A hexadecimal number is indicated by a suffix *h* or *H* or a prefix *0x* added to the number. For example, 12h (or 12H) is hexadecimal 12 or decimal 18. Similarly, 0x10 indicates hexadecimal 10 or decimal 16.

Table 4.1
Different numbering systems

Decimal	Hexadecimal	Binary	Binary Coded Decimal (BCD)
0	0	0000	0000 0000
1	1	0001	0000 0001
2	2	0010	0000 0010
3	3	0011	0000 0011
4	4	0100	0000 0100
5	5	0101	0000 0101
6	6	0110	0000 0110
7	7	0111	0000 0111
8	8	1000	0000 1000
9	9	1001	0000 1001
10	A	1010	0001 0000
11	B	1011	0001 0001
12	C	1100	0001 0010
13	D	1101	0001 0011
14	E	1110	0001 0100
15	F	1111	0001 0101

Note that, in the **BCD system**, each decimal digit is coded separately in binary. Thus the BCD representation of decimal 12 is the binary representation of decimal 1 digit and decimal 2 digit grouped together. The BCD system is normally used to drive decimal display systems. Example 4.2 addresses conversions between the binary, hexadecimal, and decimal systems.

Example 4.2 Conversions between Binary, Hexadecimal, and Decimal

A 16-bit port has the following binary pattern. Determine the value of this data in hexadecimal and decimal.

Binary Pattern: b0110100101010011

Solution:

Using Table 4.1, the 16-bit binary number will be first converted to a four-digit hexadecimal number. This will be done by finding the hexadecimal digit that corresponds to each 4-bit grouping of the binary number starting from the right end. This gives us 0x6953. The next step is to determine the value of the hexadecimal number. This is obtained by evaluating

$$6 \times 16^3 + 9 \times 16^2 + 5 \times 16^1 + 3 \times 16^0 = 24576 + 2304 + 80 + 3 = 26963$$

4.2.4 Negative Number Representation

Negative numbers are represented using a method called the **2's complement**. It is calculated for a given number by taking the complement of its bit pattern and adding 1 to it. For example, consider the representation of −1 in binary format. We will assume an 8-bit field. We start by writing the number 1 in binary format using

the 8-bit field. This gives us 00000001. Next, we find the **complement** of this number, which is the bit pattern that contains the exact opposite values of the given bit pattern (i.e., 1 changes to 0, and 0 changes to 1). So the complement of 1 is then 1111110. The last step is we to add 1 to the complement, which gives us 11111111 as the 2's complement representation of −1. Note that the representation of negative numbers in binary or hexadecimal format is very dependent on the number of bits that are used to represent the number. If we had used 4 instead of 8 bits to represent −1, the answer would be 1111. An alternative method to determine the 2's complement is to find the binary pattern representation for the result of the following operation:

$$2^n - \text{number} \tag{4.1}$$

where n is the bit field width. So applying Equation (4.1) to the representation of −1 and using a 4-bit field gives 15 ($2^4 - 1 = 15$), which has binary representation of 1111.

For an n-bit field, the range of signed numbers that can be represented by that field is from -2^{n-1} to $2^{n-1} - 1$. For example, for $n = 4$, the range is −8 to 7 or a total of 16 numbers including 0. For $n = 8$, the range is −128 to 127.

4.2.5 Representation of Real Numbers

The representation of real or floating-point numbers in binary format is more complicated than the representation of integer numbers. There are several methods available to represent real numbers; the most common is the IEEE-754 floating-point method, which is used by all modern CPUs. The representation is dependent on the number of bits that are used to represent the number. We will illustrate this method using a 32-bit field or single-precision representation. In this method, bits 0 to 22 are used to represent the mantissa or fraction, bits 23 to 30 are used to represent the exponent, and the MSB or bit 31 is used to represent the sign. For positive numbers, the sign bit is 0, and for negative numbers, the sign bit is 1. The value of the exponent is computed from bits 23 to 30 by subtracting 127. This allows both positive and negative exponents to be represented. A value of 1 in bit 22 represents a one-half fraction; a value of 1 in bit 21 represents a one-quarter fraction, etc. Note that in this representation, an invisible leading bit with a value of 1 is assumed to be placed in front of bit 22. Thus, the values of these fractions are added to the invisible one to give a mantissa value between 1 and 2. The value of a floating-point number in this representation is then computed from

$$\text{Sign} \times 2^{\text{exponent}} \times \text{mantissa} \tag{4.2}$$

As an example, the binary pattern 0|100 0000 0|111 1000 0000 0000 0000 0000 is a representation for the number 3.875. Here the exponent value is 1 (128 − 127), the mantissa value is 1.9375 (1 + 1/2 + 1/4 + 1/8 + 1/16), and the value is $2^1 \times 1.9375 = 3.875$.

4.3 Microprocessors and Microcontrollers

The **microprocessor**, which is the brain of modern computers, is an integrated circuit (or a chip) that has a processor which consists of many digital circuits. For example, microprocessors such as the Core i5 contain millions of transistor elements. For personal computers, the microprocessor is housed on the motherboard of the PC and uses an external bus to interface with memory and other components on the PC (such as mass memory and system I/O). A **bus** is a set of shared communication

lines (physically, it could be tracks on a printed circuit board or wires in a ribbon cable). The combination of the microprocessor and the other elements on the motherboard is called a **microcomputer**. The **microcontroller,** on the other hand, is a **single-chip device** that contains a processor along with memory and interface devices on the same integrated circuit chip. The microcontroller uses an internal bus to communicate with memory and other devices on the chip. Note that microprocessors require peripheral chips to interface with I/O devices.

The basic job of a processor is to execute program instructions which are the low-level code that is generated by the compiler in translating a high-level computer program (such as C code) into machine instructions that are used by that particular processor. The processor is also called the **central processing unit (CPU)** and contains three basic elements: the **control unit**, the **arithmetic and logic unit (ALU)**, and the **registers**. The function of each of these elements is described here.

Control Unit: Determines timing and sequence operations. This unit generates timing signals that are used to fetch a program instruction from memory and execute it.

Arithmetic and Logic Unit: This unit performs logical evaluations and actual data manipulation such as the addition of two numbers.

Registers: Memory locations inside the CPU that hold internal data while instructions are being executed.

For example, to add two numbers, the following operations occur. The first number is brought from memory and then held in one of the registers. The second number is brought from memory, and the ALU operates on the two numbers. The result of the operation is stored first in one of the registers before it is transferred back into memory.

During operation, a processor stores and retrieves data from memory devices. Table 4.2 explains the different types of memory devices that are used. In read-only

Table 4.2

Different types of memory

Memory Type		Description
ROM	Read Only Memory	Nonvolatile memory that is programmed with required content during manufacture of the IC chip. Data can be read but cannot be written during use, and it does not lose its data when power is turned off. It is used for fixed programs such as computer operating systems.
PROM	Programmable ROM	Same as ROM but can be programmed once by the user with no further changes allowed.
EPROM	Erasable PROM	Can be programmed more than once during use. Contents can be erased by shining ultraviolet (UV) light through a quartz window on top of the device.
EEPROM	Electrically Erasable PROM	Similar to EPROM, but contents can be erased by applying a high-voltage signal rather than a UV light.
RAM	Random Access Memory	Volatile memory that requires power to operate. Data is lost when power is removed. The access time for the data is constant and is not dependent on the physical location of the data.
SRAM	Static RAM	A RAM in which data does not need to be refreshed as long as the power is applied. The data can be accessed faster than DRAM, but it is more expensive.
DRAM	Dynamic RAM	A RAM that uses capacitors to store data. Data must be refreshed (rewritten) periodically because of charge leakage.

memory (ROM) or any variation of it (such as erasable programmable ROM (EPROM)), the data remains in memory even after the power is turned off, while in random access memory (RAM), the data is lost if the power is turned off. Recent microcontrollers use electrically erasable programmable ROM (EEPROM) to store program instructions, which are downloaded to the microcontroller through a serial or USB connection.

In addition to memory, a processor system needs the means to transfer data between the microprocessor and the other devices on the system. This data transfer occurs over a **bus**. There are different kinds of buses.

Data Bus: Used to transport data from/to the CPU and the memory or the input/output devices. Data length could be 4, 8, 16, 32, or 64 bit.

Address Bus: Used to select devices on the bus or specific data locations within memory. Each memory location has an address that must be specified before the contents of that location can be accessed. The size of the address bus determines the number of locations to be addressed. A 16-bit bus will access 2^{16} addresses or 64K locations, while a 32-bit bus can access 4G locations.

Control Bus: Used to synchronize the operation of the different elements. It transmits read and write signals, system clock signals, and other control signals.

Microprocessors and microcontrollers are designed using two design approaches. These are the **complex instruction set computer (CISC)** approach and the **reduced instruction set computer (RISC)** approach. In general, a RISC processor uses a small number of simple instructions optimized for fast execution, while a CISC processor uses more complicated instructions that can perform more functions. The compiled program for a RISC processor tends to be larger than that for a CISC processor, but it can run faster. The PIC MCUs are designed using the RISC design approach.

While microprocessors and microcontrollers share many features, they had different evolution paths. Microprocessors were developed for use in personal computers and workstations, while microcontrollers were developed for use in control applications in the appliance, automotive, entertainment, and telecommunication industries. In microprocessors, the emphasis is on high speed and large word size (such as 32 or 64-bit), while in microcontrollers, the emphasis is on compactness and low cost. In microprocessors, the RAM size is typically in Mega to Gigabyte ranges, while in microcontrollers size is given in 1 to 100K of bytes. In microprocessors, clock speed is in the range of several GHz, while in microcontrollers, it is in the tens of MHz.

4.4 PIC Microcontroller

This textbook will discuss the PIC microcontroller unit (PIC MCU) manufactured by Microchip Technology, Inc. There are many other microcontrollers on the market today (such as those made by Atmel (AVR), Freescale (HCS12), Intel (MCS-51 family), and Motorola (68HC)). We selected the PIC microcontrollers due to their widespread use (several billions of PIC MCUs have been manufactured so far), low cost, and ease of use. As mentioned before, a microcontroller is a *single*-chip device

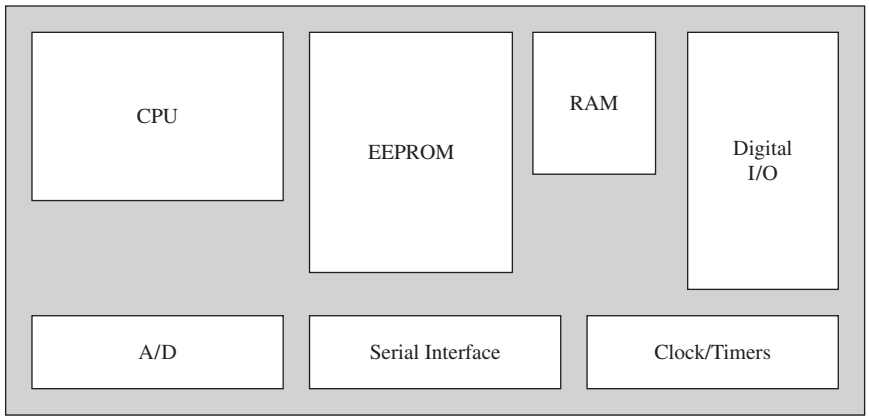


Figure 4.1

Typical components of a microcontroller

that includes a microprocessor, memory, and interface devices. The components of a typical microcontroller are shown in Figure 4.1. These components include the CPU, the nonvolatile memory (such as EEPROM) to store the code, the volatile RAM memory to store data while a program is executing, interface devices (such as digital input/output (I/O) ports, analog-to-digital (A/D) converter, serial port, or USB), the clock, and timers. Note that in a volatile memory, stored information is lost when the power supply is cut off.

4.4.1 PIC Microcontrollers Families

Microchip manufactures several families of 8-, 16-, and 32-bit microcontrollers. The number of bits refers to the size of the data bus which is used to transport data from/to the CPU and the memory or the input/output devices. Within each family, several microcontrollers are available that differ in their physical size, number of pins, memory size (program memory, RAM, and EEPROM), and type of interface devices provided. We will focus on the PIC16 and PIC18 families, which are 8-bit microcontrollers available in packages ranging from 8 to 100 pins. Table 4.3 shows a few of the many microcontrollers that are currently available in these families along with some pertinent data about them. Note that due to market forces, these microcontrollers are continually replaced by newer ones with improved features.

Program memory refers to the area on the chip that is used to store program instructions. Most of the chips in the PIC16 family have program instructions that are 14-bits wide, while those in the PIC18 family have program instructions that are 16-bits wide. Thus, for example, the PIC16F84A chip, which has 1.75 Kbytes (or 1792 bytes) of program memory, can store 1024 instructions (or 1K words). Microchip calls the PIC16 MCUs that have 14-bit program instructions as the mid-range architecture (the baseline architecture uses a 12-bit program instruction), and the PIC18 MCUs are called the PIC18 architecture. **RAM** refers to the area that stores variables and register values during program execution, while **data EEPROM** can be used to store data values during program execution (at a longer access time than RAM storage) but has the advantage that the data will not be lost if the power was lost. Many of the MCUs made by Microchip have program memory that is referred to as 'flash' memory. **Flash memory** can be erased and programmed electrically, similar to an EEPROM, but without the need for a dedicated programmer. However, a flash memory does not allow an individual memory

PIC MCU	Program Memory (Kbytes)	Data EEPROM (bytes)	RAM (bytes)	I/O Lines	A/D Channels	A/D Bits	PWM Channels	Pin Count	CPU Speed (MIPS)	Interface
PIC16F84A	1.75	64	68	13	0	N/A	0	18	5	–
PIC16F872	2.5	64	128	22	5	10-bit	1	28	5	MI^{2}C /SPI, MSSP
PIC16F690	7	256	256	18	12	10-bit	4	20	5	A/E/USART, I^{2}C/SPI
PIC16F76	14	0	368	22	5	8-bit	2	28	5	USART, I^{2}C/SPI
PIC18F2220	4	256	512	25	10	10-bit	2	28	10	A/E/USART, MSSP(SPI/I^{2}C)
PIC18F4550	32	256	2048	35	13	10-bit	4	40	12	A/E/USART, MSSP(SPI/I^{2}C), USB
PIC18F86J60	64	0	3808	55	15	10-bit	12	80	9.5	A/E/USART, MSSP(SPI/I^{2}C), Ethernet
PIC18F8722	128	1024	3936	70	16	10-bit	12	80	10	A/E/USART, MSSP(SPI/I^{2}C), LIN

Table 4.3

A sampling of different PIC MCUs in the PIC16 and PIC18 families

location to be erased; only a single block of memory locations can be erased. Program memory and data memory have separate buses to allow concurrent access and faster throughput.

The number of **I/O lines** refers to the number of TTL digital input/digital output lines available on the chip. These I/O lines are bi-directional and are configured by program code to be either an input or output type. Some chips come with analog-to-digital converter capability with 8- or 10-bit resolution. Some A/D channels use the same pins as those used for digital I/O lines, but they can be configured by program code to operate as A/D channels. Some chips have one or more PWM lines which can be conveniently used to drive motors through an H-bridge driver or a transistor. The **pin count** refers to the number of physical pins on the chip. Some chips are available in different pin counts, depending on the chip packaging configuration (see Section 4.4.2). For example, the PIC18F4550 MCU is available as 40 pins in the PDIP configuration and as 44 pins in the TQFP configuration.

The **CPU speed** refers to the maximum speed of the chip in units of millions of instructions per second (MIPS). A chip with a higher MIPS rating can execute a program faster than one with a lower rating. Some high-end chips have built-in **interfaces** for ease of communication with other devices. An explanation of some of these interfaces is given in Table 4.4. A Universal Synchronous Asynchronous Receiver Transmitter (USART or any variation of it such as AUSART) is a module commonly used for asynchronous serial communication using the RS232 protocol, while a Master Synchronous Serial Port (MSSP) is used for synchronous serial communication. The MSSP is commonly used to communicate with external EPROM or RAM. USB stands for universal serial bus, while the Local Interconnect Network (LIN) is a serial communication system. Other MCUs in the PIC18 family have an integrated liquid crystal display (LCD) module for

Interface Feature	Explanation
USART	Universal Synchronous Asynchronous Receiver Transmitter (USART) module is used for synchronous (data line and clock signal) and asynchronous (data line but no clock signal) serial communication
AUSART	Addressable Universal Synchronous Asynchronous Receiver Transmitter (AUSART) module can be configured as asynchronous (full duplex), synchronous–master (half duplex), or synchronous–slave (half duplex) serial communication line
EUSART	Enhanced Universal Synchronous Asynchronous Receiver Transmitter (EUSART) module supports RS-485, RS-232, and LIN compatibility with auto-baud detect and auto-wake-up on start bit
MSSP	Master Synchronous Serial Port (MSSP) module includes an SPI™ and I^2C™
SPI™	Synchronous serial port (SSP) configured as three-wire serial peripheral interface (SPI)
I^2C™	Synchronous serial port configured as a two-wire inter-integrated circuit (I^2C™) bus
MI^2C	Master I^2C port
USB	Universal serial bus
LIN	Local Interconnect Network, which is a serial communication system

Table 4.4

A list of interfaces available on some PIC MCUs

directly driving LCD devices. These MCUs are ideal for applications such as thermostats, handheld meters, and portable medical devices.

4.4.2 Pin Layout

As an example of the pin layout for a typical PIC MCU, consider Figure 4.2 which shows the pin configuration for the PIC16F84A, which is a popular PIC MCU. This MCU has 18 pins. Note that some of the pins have dual functions (such as pins 3 and 6). As is the case with many of the PIC MCUs, they are available in different pin layouts. A common layout is the PDIP, which stands for **plastic dual inline package (PDIP)**. In this layout, the pins are arranged in two parallel opposite rows, and this layout is normally used with breadboards. The small-outline **integrated circuit (SOIC)** package uses gull-wing pins extending outward, while in the **shrink small outline package (SSOP)**, the pins are also gull-wing shaped

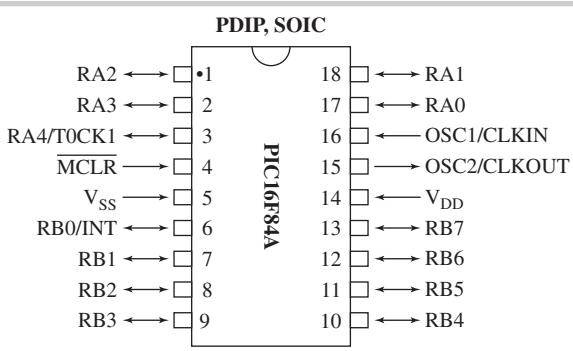


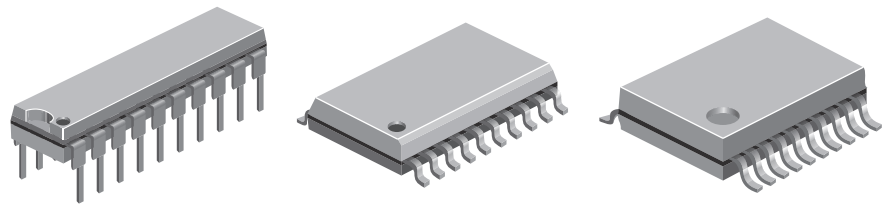
Figure 4.2

Pin layout for the PIC16F84A chip

(Reprinted with the permission of Microchip Technology Incorporated)

Figure 4.3

(a) PDIP, (b) SOIC, and (c) SSOP packaging
(Reprinted with the permission of Microchip Technology Incorporated)



but are more closely spaced than in SOIC package. The last two layouts are normally used in surface-mount type circuits. Figure 4.3 shows examples of these package types.

The PIC16F84A MCU is a good choice for applications that require no A/D conversion or PWM output. In this text, we will focus on both the PIC16F690 MCU and the PIC18F4550 MCU, which have more features than the PIC16F84A chip. The PIC16F690 chip has the following features.

- Program memory size of 7 Kbytes
- Twelve A/D channels
- Eighteen I/O pins arranged as ports A, B, or C
- Three timers
- PWM output
- Several options for serial communications

The PIC18F4550 adds USB communication, larger program memory, RAM size, and multiple interrupt-handling capability. The discussion about the PIC18F4550 will be limited to some of the features on that chip that are not available on the PIC16F690. Figure 4.4 shows the pin diagram for the PIC16F690 MCU. Due to the fact that this MCU has only twenty pins but supports many interface functions, many of the pins are designed for more than one function. For example, pin 3 can be configured as channel 4 of digital input/output port A (RA4), analog channel 3 (AN3), timer 1 gate input ($\overline{\text{T1G}}$), oscillator crystal connection pin #2 (OSC2), or clock output (CLKOUT). The limited number of external pins also means that you cannot have 12 A/D channels and 18 I/O lines operating at the same time. Table 4.5 provides an explanation of the functions of some of these pins.

Figure 4.4

Pin diagram for the PIC16F90
(Reprinted with the permission of Microchip Technology Incorporated)

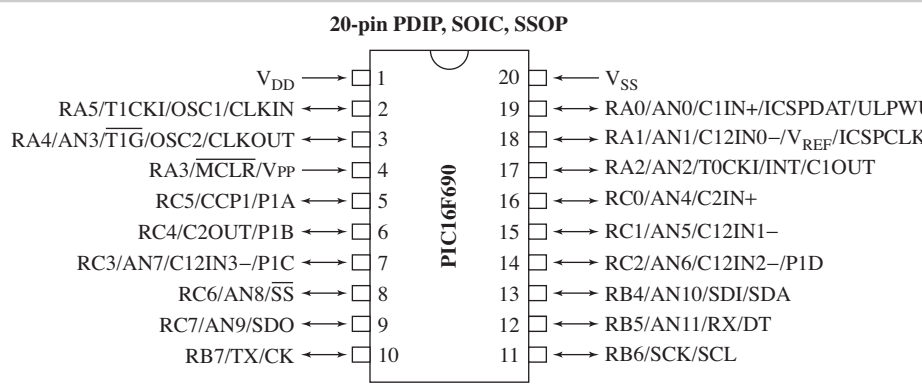


Table 4.5

Pin functions on the PIC16F690 MCU

AN	Analog-to-digital input channel
C1IN, C1OUT	Comparator input and comparator output
CCP	Capture/compare/PWM module
CK	EUSART synchronous clock
CLKIN	External clock input/RC oscillator connection
CLKOUT	$F_{osc}/4$ output line
DT	EUSART synchronous data
ICSPDAT, ICSPCKL	Integrated circuit serial programming (ICSP) data I/O and ICSP clock signal
INT	External interrupt
MCLR	Master clear
OSC	Crystal/resonator
P1A, P1B, P1C, PID	PWM output line
RA, RB, RC	General purpose I/O. The A, B, and C refer to the A, B, or C port
RX	EUSART asynchronous input
SCK, SCL	SPI clock and I^2C clock
SDI, SDA, SDO	SPI data in, I^2C data input/output, and SPI data out
SS	Slave select line for SPI
T1CK1	Timer1 clock input
TX	EUSART asynchronous output
VDD	Positive lead of supply voltage. This chip operates at an input voltage range of 2 to 5.5 V
VPP	Programming voltage pin
VREF	External voltage reference for A/D
VSS	Ground reference line

4.4.3 PIC MCU Components

Figure 4.5 shows a block diagram of the main components of the PIC16F690 MCU, including the data and program buses. Some of these components include the three digital I/O ports (A, B, and C), the three timers (0, 1, and 2), the EUSART module for RS232 communication, the A/D converter module, the synchronous serial port (SPI and I^2C), the enhanced capture compare PWM (ECCP) module, the program memory, and the CPU. Most of these components are explained in detail in later sections of this chapter. Note how the 8-bit data bus connects the CPU (the ALU and CPU registers such as the W and STATUS registers) to the other components on the MCU.

Program instructions are brought to the CPU through the 14-bit program bus under the control of the program counter. The **program counter (PrC)** is a special register that holds the address of the next instruction to be executed. The PrC size is made so that it should be able to access all of the instructions in program memory. On the PIC16F690, the PrC is 13-bits wide and thus can access up to 8192 instructions. When the MCU is powered on or is reset, the PrC is cleared and points to the address 0x0000. Associated with the PrC is the **stack**, which is a special area of memory that is not associated with data or program memory. The stack

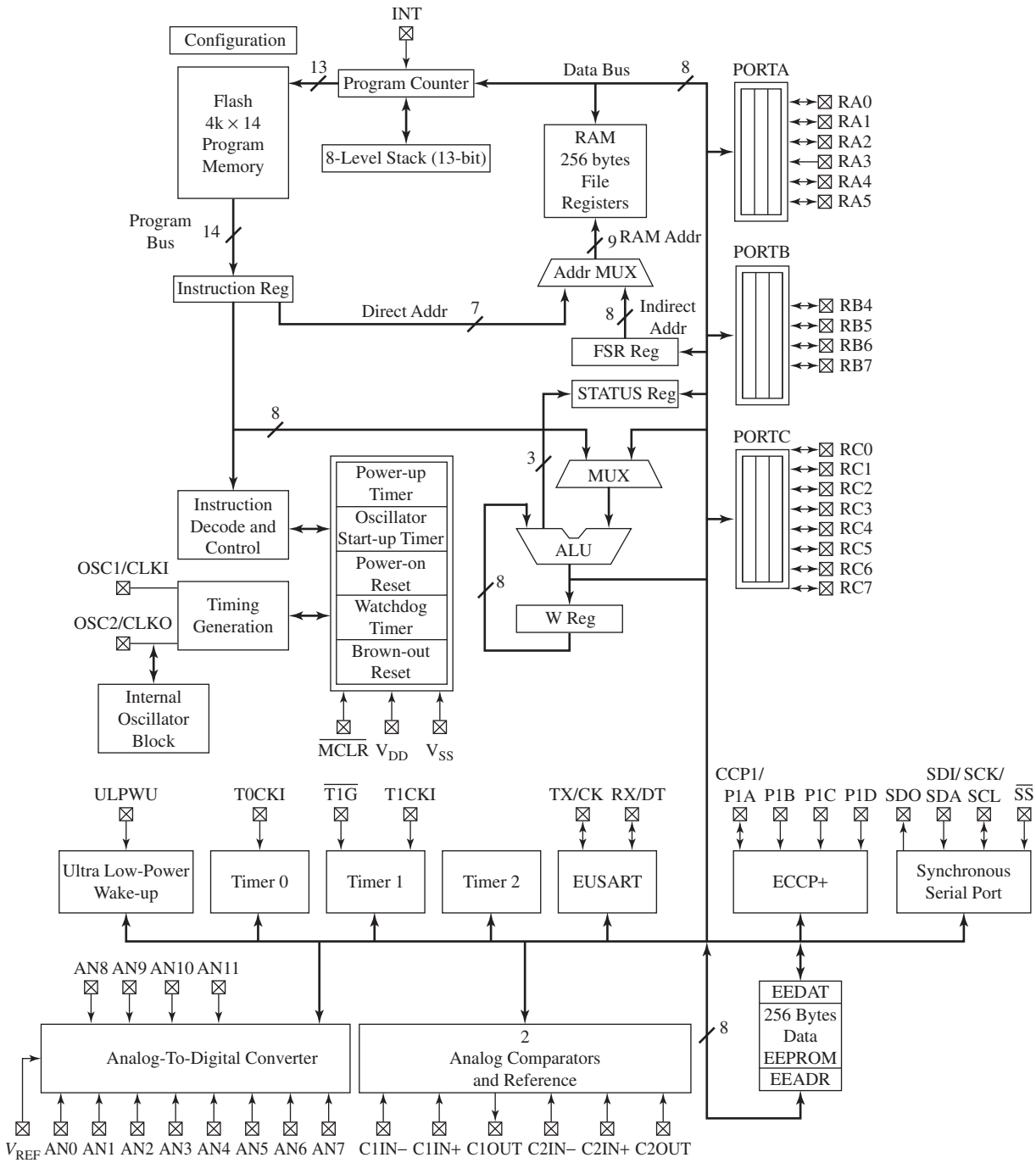


Figure 4.5

A block diagram of the PIC16F690 MCU

(Reprinted with the permission of Microchip Technology Incorporated)

operates in a first-in, last-out (FILO) fashion and is used as a storage area of the contents of the PrC when executing a subroutine call or an interrupt (see Section 4.8). When a program needs to perform a branching operation (such as when executing a subroutine call), the contents of the PrC are placed on the stack (pushed), and the address of the subroutine to be executed is loaded into the PrC. When the routine

completes execution, the address of the next instruction is loaded from the stack (popped) back into the PrC. The size of the stack determines the number of subroutine calls or interrupts that can occur. In the PIC16F690, the stack is eight levels deep.

To get a PIC MCU to run, you need to wire at least three pins. These include the VDD pin, which should be connected to the positive lead of the supply voltage (2 to 5.5 V for most chips but some chips are designed to operate with 1.8 to 3.6 V range); the VSS pin, which should be connected to ground voltage; and the MCLR pin, which should be connected to the VDD (through a resistor) to prevent the MCU from resetting itself. If the MCLR pin is a shared pin (such as in the PIC16F690 MCU), then the MCLR does not need to be wired to the VDD if it is programmed to treat the pin as the other function. For microcontrollers without internal clock sources, you also need to connect the external clock source to the microcontroller. The remainder of this section gives details on the basic operations of this chip.

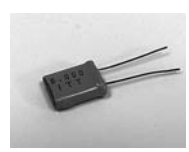
4.4.4 Clock/Oscillator Source

A microcontroller needs a clock source in order to function, since all CPU operations are synchronized with the clock. A **clock** is any device that can produce a train of pulses at a fixed frequency. Some chips have a built-in clock source (such as the PIC16F690, which has two internal clocks: an 8 MHz and a 31 kHz), while others require or allow an external device to produce the clock pulses. These external devices include a quartz crystal resonator, a ceramic resonator, a resistor-capacitor (RC) circuit, or an external clock source (such as the 555 timer chip). The microcontroller allows the selection of the clock source through software.

A **quartz crystal resonator** uses the mechanical resonance of a piezoelectric crystal to generate a very accurate timing signal, while a **ceramic resonator** uses the mechanical resonance of piezoelectric ceramics, commonly lead zirconate titanate (PZT). Ceramic resonators are small, rugged, and inexpensive, while quartz crystal resonators are more expensive, but more precise. Ceramic and quartz crystal resonators are available with different clock frequencies. A photo of a quartz crystal resonator is shown in Figure 4.6.

Figure 4.6

A quartz crystal resonator
(Jouaneh, University of Rhode Island)



Using either type of resonator, the clock signal is connected to pins OSC1 and OSC2 on the chip. For a crystal oscillator, the clock circuit wiring is shown in Figure 4.7. Two capacitors (C_1 and C_2) of 15 pF for an 8-MHz crystal oscillator are needed. In addition, a series resistor (R_S) may be required for quartz crystals with a low drive level. The value of the parallel feedback resistor (R_F) is dependent on the oscillator mode and varies between 2 to 10 MΩ.

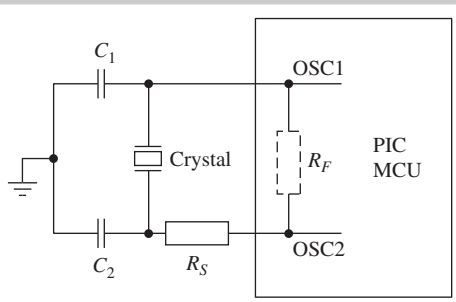


Figure 4.7

Connection diagram for a quartz crystal resonator for a PIC16F690 MCU

An **RC circuit** uses the charge and discharge time of an RC circuit to produce a clock signal. It is normally used in applications where clock accuracy (1 to 10% error) is not very important. The RC circuit connects to pin OSC1 on the microcontroller. Pin OSC2 becomes a general purpose I/O line (if the clock mode is set to RCIO) or can be used to output the RC oscillator frequency divided by 4 (if clock mode is set to RC). An **external clock source** (such as the 555 timer) also can be used as the clock. The clock output connects to the OSC1 pin on the chip, and the OSC2 pin is available for general purpose I/O. A 555 timer chip (such as the NE555 8-pin chip from Texas Instruments, discussed in Section 3.9) is an integrated circuit that uses transistors, resistors, and diodes to produce a variety of clock signals, including periodic signal output and time delays.

The PIC16F690 has two **internal oscillators**: an 8-MHz high-frequency oscillator and a 31-kHz low-frequency oscillator. The output frequency of the 8 MHz oscillator can be further divided by software. If an internal oscillator is used as the system clock source, then the OSC1/CLKIN pin is available for general I/O. The OSC2 pin can be used either for general purpose I/O (clock mode set to INTOSCIO) or to output the clock frequency divided by 4 (clock mode set to INTOSC). Note that the PIC instructions execute at a frequency that is one-quarter of the **clock oscillator frequency (F_{OSC})**. In the microcontroller documentation, this is referred to as $F_{OSC}/4$. Thus, a PIC chip with an 8-MHz clock source will execute one instruction every 0.5 μs.

In chips that support USB communication (such as the PIC18F4550), an additional clock branch is given on the chip to provide a 48 MHz clock with full USB operation. For the USB operation, a phase locked loop (PLL) circuit is used to provide the 48-MHz clock signal using input clock sources that can range in frequency from 4 to 48 MHz. While a single, primary oscillator (such as a 20 MHz crystal) can be used to provide timing signals for both the USB module and for instruction execution, the chip also allows the use of two oscillator types at once. In this case, the primary oscillator is used for USB timing, but the internal oscillator is used for instruction timing. Regardless of the use of single or dual oscillators, the instruction timing signals can be run at a different frequency than those used for USB timing.

4.4.5 I/O and A/D Operation

Many PIC MCUs support both digital input/digital output functions and analog-to-digital conversion. Before a digital I/O pin is used, the pin should be configured to either input or output. This is done by writing to the **tristate** (see Section 3.8) **register** (a register is a memory location set for a particular purpose) associated with the port to which the pin belongs to. For example, the PIC16F690 has three I/O ports (referred to as ports A, B, and C) and three tristate registers associated with these ports (referred to as TRISA, TRISB, and TRISC), respectively. A value of 1 written to the tristate register bit corresponding to a particular I/O pin causes that pin to function as digital input, while a value of 0 causes that pin to function as digital output. Note that each component on the MCU (such as an I/O line, an A/D converter, or a timer) has a number of registers that control its operation. The detailed listing of all of these registers is not the intention of this text, and the reader is encouraged to read the data sheet file for the particular MCU for further information.

If a pin is designated both as digital I/O and A/D (such as pin 3 in Figure 4.4), then one needs to select the desired function by writing to the appropriate register. A shared pin will function as an A/D if the bit corresponding to that pin in the

ANSEL (analog select) or ANSELH (analog select high) register is set to 1. Similarly, setting the same bit to 0 will cause that pin to function as digital I/O. Note that to use a pin for A/D, the corresponding bit in the tristate register has to be set to 1 (i.e., to function as an input). Some compilers (such as the PIC-C compiler that will be discussed later) can perform these setup operations automatically.

The PIC MCU A/D converter coverts analog voltages to digital numbers. The analog voltage range is 0 to either VDD (2.0 to 5.5 V) or to an external reference voltage (if +Vref is set). The resolution is 8, 10, or 12 bits. On the PIC16F690, the resolution is 10 bits, but the output can be mapped into 8 bits. The converted voltage can be either right or left justified. Default is left justified unless told in the software (by writing to the ADCON0 register for PIC16F690).

4.4.6 PWM Output and Reset Operations

None of the MCUs in the PIC16 or PIC18 families have a digital-to-analog conversion capability, but many PIC MCUs have a built-in module to generate pulse-width modulated (PWM) output (such as pins 5, 6, 7, or 14 in Figure 4.4). A **PWM signal** is a square-wave signal of fixed amplitude and frequency, but the width of the on and off parts of the signal (or duty cycle) can be varied (see Figure 4.8). The PWM output can be used to conveniently drive H-bridge drives and digital amplifiers. The PWM output mode is one of the three modes of operation of the **Capture/Compare/PWM (CCP)** module or **enhanced CPP (ECCP)** module on the MCU. The other modes are capture and compare. In the **capture mode**, the value of the particular timer associated with the CCP module is copied to a particular register when an input event occurs on a designated CCP pin. Thus, the capture mode can be used for timing input events. In the **compare mode**, an action is triggered when the value in the CCP registers matches the value stored in the particular timer associated with the CCP module.

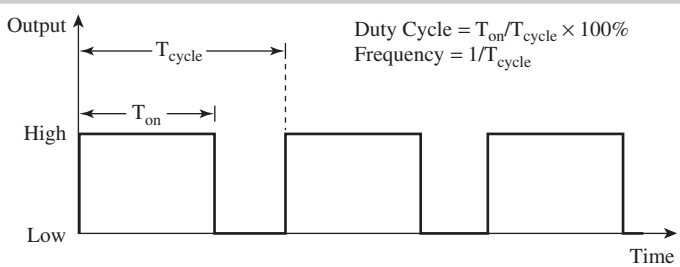


Figure 4.8

Illustration of a PWM signal

There are different **reset operations** that are available on a PIC MCU. These include power-on reset (POR), brown-out reset (BOR), watchdog timer reset, and external MCLR reset. These are discussed next. A POR occurs whenever the power (VDD line) is turned off and then on to the chip. After a POR, the code on the chip starts executing at the first program memory instruction, and some of the registers on the chip will reset to their "Reset" state. A BOR (if enabled) produces the same result as a POR and occurs whenever the VDD voltage level falls below the rated voltage (between 2 to 5.5 V for most chips). The BOR does not occur unless certain registers on the chip were set to detect this condition. This feature is useful in battery-powered applications to detect low voltage conditions. The watchdog timer reset occurs whenever the counter associated with the watchdog timer overflows, while a MCLR Reset occurs whenever the MCLR pin line goes to low. Note that the PIC MCU has a special register (called PCON or power control) that can indicate which type of reset has occurred.

4.5 Programming the PIC Microcontroller

The development of a control program running on a microcontroller shares some similarities with developing a program to run on a PC. On a PC, a high-level programming language (such as C or Visual Basic (VB)) is used to prepare the code. Using the built-in compiler in the integrated development environment (IDE) that comes with that particular language, the high-level code is translated into binary machine code. The binary code is then linked with other needed files to form the executable program that can be run by simply calling its name. The developed code can be debugged for errors by simply utilizing the *Debugging* function that comes with the IDE.

In a microcontroller, a program can be similarly developed using a high-level programming language (such as C or VB), but also can be developed using assembly language. Assembly language is a low-level programming language that is specific to the microcontroller used. For most PIC MCUs in the PIC16 family, the assembly language is made of just 35 instructions. While it is more difficult to program the code in assembly language, the user has better control of the execution timing of the code, because the execution timing of each instruction (in terms of the number of clock cycles it takes to execute that instruction) is known. Another advantage of programming in assembly language is that you do not need to buy any additional software tools, since the assembly compiler is provided free of charge. Similar to a high-level program, the assembly code needs to be compiled and translated into binary code. The binary code (or hex code file) is then downloaded to the microcontroller to be stored in the non-volatile program memory.

Microchip provides the MPLAB Integrated Development Environment free of charge for use in developing code for their line of microcontrollers. MPLAB is an integrated editor, compiler, linker, and debugger. It has a built-in assembly compiler.

4.5.1 Programmers

The process of transferring a compiled binary code to the MCU is called "programming" a chip. Originally, Microchip provided the *PICStart Plus*® programmer (see Figure 4.9), in which the user plugs the chip to be programmed into the device. Then, through a serial line from the PC to the PICStart Plus, the MPLAB IDE is used to transmit the binary code to the chip. Once the chip is programmed, the chip is removed from the programmer and transferred into the target system in which it will be used. More recently, Microchip introduced the PICkit 2 and then the PICkit 3 Microcontroller Programmer. These are low-cost development programmers that can be conveniently used to program many MCU chips.

Both the PICkit 2 and the PICkit 3 programmers (see Figure 4.10) connect to a PC through a USB cable. Using the software that comes with them, one can

Figure 4.9

PICSTART Plus

(Reprinted with the permission of Microchip Technology Incorporated)

Figure 4.10

(a) PICkit 2 and (b) PICkit 3 programmers

(Reprinted with the permission of Microchip Technology Incorporated)

(a) (b)

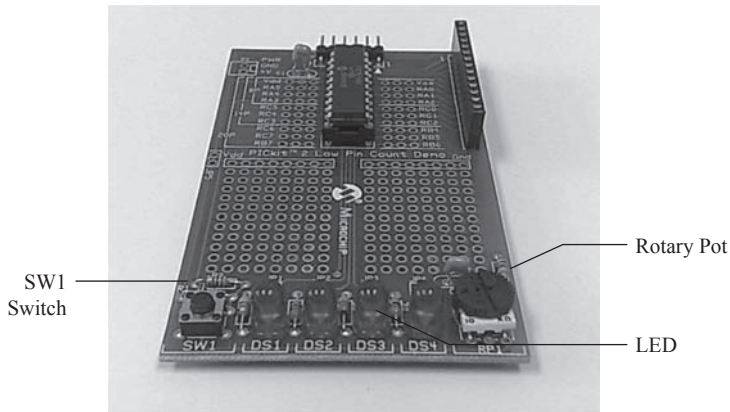


Figure 4.11

Microchip low pin-count development board
(Jouaneh, University of Rhode Island)

download programs to the PIC MCU. Both programmers conveniently plug into a number of development boards (an example of such one is shown in Figure 4.11). This 'low pin-count' development board has four LEDs, a single-turn potentiometer, and a switch. The development board can be used to run basic code without the need to build a custom board to house the PIC MCU. Microchip manufactures many different types of these development boards.

The PICkit 2 programmer can be used from the MPLAB IDE or through the use of a separate stand-alone program called the PICkit 2 program that Microchip provides. There is no stand-alone program for PICkit 3, and PICkit 3 has to be called from the MPLAB IDE. The PICkit 3 also can be used as a debugger to step and to examine a code as it executes.

Note that the PICkit 2 programmer can be used to provide power to the connection target provided that the power drawn from the development board is minimal. This is done by checking the *On* check box in the *VDD PICkit 2* group area (see Figure 4.12) and adjusting the VDD voltage value (normally set to 5.0 V).

Programming the PIC MCU through the use of the PICkit 2 or PICkit 3 makes use of a useful feature that is available on many PIC MCUs. This feature is called **integrated circuit serial programming (ICSP)**. The ICSP uses two pins (data and clock) to transfer code in a serial fashion into/from the PIC MCU. To do ICSP, three other pins on the chip need to be connected. These include the supply voltage pin (VDD), the ground pin (VSS), and the programming voltage pin (VPP). An advantage of ICSP is that the chip does not have to be removed from the development or target hardware board. This allows an unprogrammed chip to be mounted into the board and then to be programmed later.

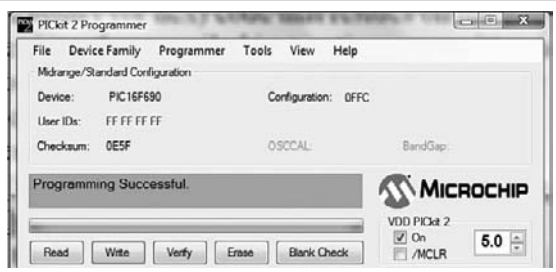


Figure 4.12

PICkit 2 interface
(Jouaneh, University of Rhode Island)

4.5.2 Bootloaders

Another way to program PIC microcontrollers is to use a bootloader program. A **bootloader** is code that resides on the MCU program memory. The bootloader code resides on an area of program memory that is not normally used for main program execution. The bootloading code on the MCU uses an RS-232 serial line to communicate with a corresponding PC bootloading application. The PC bootloading application allows the user to download a hex file to the PIC MCU without the use of any external programmer. The code is simply transferred from the PC to the MCU through an RS-232 serial line. The PC bootloading application also allows the user to read or verify application code on the MCU. Note that the user has to initially load the bootloader code into the PIC device using a programmer such as PICkit 2. Microchip Technology provides the AN1310 software package, which is a high-speed bootloader for PIC16 and PIC18 devices.

4.6 C-Language Programming

There are several programming languages available to program PIC MCUs. These include PicBasic Pro, C, and assembly. While the PICBasic Pro language (developed by microEngineering Labs, Inc., Colorado Springs, CO) is easy to learn, the language has some limitations. These include the lack of floating-point variables, lack of passing arguments to subroutines, and mathematical operations that treat all variables as unsigned (with the exception of a new version of the complier that supports a 32-bit signed variable).

There are several C-language compilers for PIC MCUs that have several of the features that are lacking in the PICBasic Pro compiler. Among these compilers is the PCWH compiler (will be referred to as the PIC-C compiler) developed by CCS, Inc. of Brookfield, WI. The PIC-C compiler has actually several versions: PCB for 12-bit program instruction MCUs (such as the PIC10F200), PCM for 14-bit program instruction MCUs (such as the PIC16F690), and PCH for 16-bit program instruction MCUs (such as the PIC18F4550), but all have a similar interface.

This section will only give a brief highlight of some features of this compiler. Further details about the compiler can obtained from the company website (www.ccsinfo.com). Details of the C-programming language can be found in [12]. We will start by talking about the supported variables types. Table 4.6 shows the five variable types supported by this compiler. They include four integer type variables: a one-bit variable (**int1**), an 8-bit (**int8**) integer, a 16-bit (**int16**) integer, and a 32-bit (**int32**) integer; as well as a 32-bit signed floating-point variable (**float32**). The four integer types are by default unsigned but can be changed to signed (not applicable to int1) variables by adding the **signed** keyword to the variable type. The last column in Table 4.6 shows the standard C variable types that correspond to these variables.

Table 4.6 Variable types supported in the CCS C-compiler

Type	Size	Range		C-Standard type
		Unsigned	Signed	
int1	1 bit number	0 to 1	N/A	short
int8	8 bit number	0 to 255	−128 to 127	int
int16	16 bit number	0 to 65535	−32768 to 32767	long
int32	32 bit number	0 to 4294967295	−2147483648 to 2147483647	long long
float32	32 bit float	-1.5×10^{45} to 3.4×10^{38}		float

Directive	Purpose	Example
#device	To specify chip options for the devices on a chip	#device (ADC = 10)
#fuse	To specify specific configuration settings	#fuse NOWDT
#include	To include the device specific functionality	#include <16F690.h>
#include	To include specific c-functions that are not included in the standard library	#include <string.h>
#use	To specify the configuration parameters for built-in libraries for devices such as the clock, RS232, and I^2C	#use delay(internal=8M)

Table 4.7

Listing of some pre-processor directives in PIC-C language

The compiler uses pre-processor directives to define the particular chip used as well as the chip settings. Pre-processor directives begin with a # and are followed by a specific command. A list of some of these directives is given in Table 4.7.

The *#device* directive is used to specify the options for the various devices on the chip (such as the number of bits that the A/D convertor should return) or to generate code compatible with the integrated circuit debugging hardware. The *#fuse* directive is a very nice feature of the PIC-C compiler. The user does not need to write to specific registers to set the configurations for the various elements on the chip. The user simply specifies a particular fuse setting. The list of allowable fuse settings is available in the *VIEW* tab in the compiler IDE. Figure 4.13 shows the fuse setting list for the PIC16F690 chip.

Figure 4.13

Fuse settings list for the PIC16F690 chip

The *#fuse* directive is used primarily to specify the timer operating modes, including the watchdog timer and the reset modes on the chip. Note that each chip has its own header file (with an *.h* extension) which has a listing of all of the constants that are used in the functions that access the devices on that chip. That file is included using the *#include* directive. The *#use* directive is used to specify the configuration parameters for built-in libraries for devices such as the clock, RS232, and I^2C. The PIC-C compiler has many other pre-processor directives that were not included in Table 4.7. These include the standard C directives (such as *#define* and *#if*), directives for pre-defined identifiers (such as — DEVICE —), and for memory control (such as *#asm*).

The PIC-C compiler has many built-in functions to setup and use the various devices on the MCU (such as digital I/O lines, A/D convertors, timers, the serial port, PWM output, and I^2C interface). Some of these functions are discussed next.

4.6.1 PIC-C I/O Functions

For I/O, the PIC-C compiler provides functions that can affect a single bit or the entire port. The **pin functions** are

```
output_high(pin)          //Set the selected pin to high
output_low(pin)           //Set the selected pin to low
output_pin(pin, value)    //Send a specified value (0/1) to selected pin
input(pin)                //Returns the state of the selected pin
```

and the **port functions** are

```
output_x(value)           //Send an entire byte to port x (x =  a, b, c, d . . .)
input_x()                 //Read an 8-bit integer representing the port input value
```

The compiler has directives to specify the type of input/output. In the STANDARD_IO (default method) mode (*#use standard_io(port_name)*), the compiler automatically generates code to make an I/O pin either input or output every time it is used The tristate register is automatically updated in this mode. In the FAST_IO mode (*#use fast_io(port_name)*), the compiler will perform I/O without programming of the direction register. The user has to set the direction by calling the set tristate register function (*set_tris_x()*).

The C-language code listing for turning a bit ON and OFF is shown in Figure 4.14. The code uses the *#include <16F690.h>* directive to include the constant definitions for the chip used (such as *PIN_C0*, which refers to pin 0 on port C). The code also specifies (through the *#use delay* directive) the clock frequency as that of

Figure 4.14

Code listing for performing digital I/O using the PIC-C compiler

```
////////////////////////////////////////////////////////////////////////////////
///                    BitOnOff.c
///  This program  turn on/off bit 0 of Port C
///
///   Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////
#include <16F690.h>                    //Include file for the particular chip used
#use delay(internal=8M)                //Use Internal 8 MHz-clock

#fuses INTRC_IO, NOMCLR                //Set clock mode to internal oscillator with
                                       // no clkout. Master clear pin is used for I/O

void main(void)                         //Main function
{
  while ( 2 > 1)                       // Start infinite loop
  {
  output_high(PIN_C0);                 //Set the selected pin to high
  delay_ms(1000);                      //Set a time delay of 1000 ms
  output_low(PIN_C0);                  //Set the selected pin to low
  delay_ms(1000);                      //Set a time delay of 1000 ms
  }
}
```

the internal 8 MHz oscillator. The *#fuse* directive specifies the clock mode as the internal oscillator and disables the use of the MCLR line. Due to the simplicity of the operation performed here, only one function (*main*) is used in the code to implement the infinite *Do-Loop* operation.

4.6.2 PIC-C A/D Functions

To use the A/D converter, the compiler provides several functions for this purpose. First, the user needs to call the function ***setup_adc()***, which sets up the clock source for the A/D converter. The user can select either a sub-frequency of the oscillator frequency (such as $F_{OSC}/2$ or $F_{OSC}/16$) or the A/D dedicated internal oscillator (F_{RC}) as the clock source. The clock frequency determines the time it takes to do a one-bit conversion (T_{AD}). To do a full 10-bit conversion, it takes 11 T_{AD} cycles to complete the conversion. Thus, with an oscillator frequency of 8 MHz and A/D clock frequency of $F_{OSC}/16$, it takes 22.0 μsec, to do one full 10-bit conversion. For accurate A/D conversion, the T_{AD} interval should be as short as possible but greater than the minimum T_{AD} (about 0.7 to 1.4 μsec, but it is device dependent). This means that if the oscillator frequency is 20 MHz, then the A/D clock source should not be set to $F_{OSC}/2$, since it results in T_{AD} value which is less than the minimum T_{AD} required. Note that, using the A/D dedicated internal oscillator, it takes about 2 to 6 μsec to just do one-bit conversion. Thus, if the oscillator frequency is above 1 MHz, it is not recommended to use the F_{RC} mode, since it will result in a longer time to do the conversion.

Next one needs to select which pins on the MCU will be used for the A/D conversion and the voltage reference to use when computing the A/D value. This is done by calling the function ***setup_adc_ports()*** function. Using this function, all, some, or none of the channels can be set to perform A/D. In addition, this function can be used to specify the channel number to be used as the analog reference voltage if a voltage reference other than VDD is used. The above two functions need only to be called once.

To read a particular A/D channel, the channel needs to be selected first by calling the ***set_adc_channel()*** before the ***read_adc()*** function is called. If the same A/D channel is read each time, then only the *read_adc()* function has to be called after the *set_adc_channel()* function was called once. The bit resolution of the A/D conversion is set by including the directive *#DEVICE ADC = Num_of_bits* at the top of the C-language file.

As an example, the PIC-C language code for turning ON/OFF an LED based on reading an analog input is shown in Figure 4.15. In the *main()* routine, the *setup_adc()* function is called first to select the main oscillator frequency divided by sixteen ($F_{OSC}/16$) as the clock source for A/D conversion. Then the function *setup_adc_ports()* is called to configure channel RA0 to operate as an A/D channel. This is followed by calling the function *set_adc_channel()* to select channel RA0 for the conversion. In the infinite loop, the A/D channel is read using the function *read_adc()*, and the 10-bit read value is assigned the variable *addata*, which is a 16-bit integer.

4.6.3 PIC-C Timing Functions

The PIC-C compiler has functions for setting and reading the timers available on the chip. To setup timer0 for example, the ***setup_timer_0()*** function is called. The function gives the user the choice of a clock source (internal or external) and the prescale factor (see discussion in next section about delays and timers) to use. As an

Figure 4.15

PIC-C code listing for turning on/off an LED based on analog input

```
////////////////////////////////////////////////////////////////////////////////
///                    Analog_Input_LED.c
///
///  This program set on/off LED on pin C0 based the value read
///  from A/D channel 0 (RA0)
///
///   Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////
#include <16F690.h>                    //Constant definitions for chip used
#DEVICE ADC = 10                       //10-bit A/D return value
#fuses  INTRC_IO, NOMCLR               //Set clock mode to internal oscillator with
                                       // no clkout. Master clear pin is used for I/O
#use delay (INTERNAL=8M)               //Use Internal 8 MHz clock
void main()
{
int16 addata;
setup_adc(ADC_CLOCK_DIV_16 );          //Set A/D clock frequency as FOSC/16
setup_adc_ports(sAN0);                 //Set channel 0 for A/D
set_adc_channel(0);                    //Select channel 0
while ( 2 > 1)                         // Start infinite loop
  {
  addata = read_adc();                 //Read selected A/D channel
  if (addata < 512)
    {
     output_high(PIN_C0);              //Set port C0 to high
    }
  else
    {
    output_low(PIN_C0);                // Set port C0 to low
    }
    delay_ms(1000);                    // Wait 1000 ms
  }  }
```

example, to use the internal clock on the MCU, and a prescale factor of 8, the function is called as

Setup_timer_0 (RTCC_INTERNAL, RTCC_DIV_8)

where the particular parameters used are dependent on the selected MCU (done by adding the header file for the MCU used to the C-language file). Once the timer is set up, the timer is accessed by using the provided ***get_timer0()*** function, which returns the count value of the counter associated with this timer. The compiler also provides the ***set_timer0()*** function, which sets the count value of the counter to a particular value.

4.6.4 PIC-C PWM Functions

As mentioned previously, the PIC16F690 MCU can supply PWM output signals from pins 5, 6, 7, or 14 (see Figure 4.4). The PIC-C complier provides functions for this operation. The PWM output functions are part of the CCP module on the MCU, and thus, the CPP module has to be configured for PWM operation by the calling the function ***setup_ccp1(CCP_PWM)***, which configures the P1A channel to operate in PWM mode. The timing for the PWM signals are controlled by timer2 on the chip, and the frequency of the PWM signals is set by calling the ***setup_timer_2()*** function. The duty cycle of the PWM signal is set by calling the ***set_pwm1_duty()*** function, which sets the duty cycle from 0 to 100%. The function uses a 10-bit value to set the duty cycle. Further details about PWM operation are provided in the next section.

4.7 PIC MCU Devices and Features

In this section, we will discuss devices and features of the PIC MCUs. These include data memory, EEPROM data, program memory, delays and timers, PWM duty cycle and timing, watchdog timer, power saving, A/E/USART, analog comparator, synchronous serial interface (SSP) module, and I^2C interface.

4.7.1 Data Memory

In a PIC MCU, the data memory (in which data is stored during program operation) is portioned into banks. In the PIC16F690 MCU, the data memory is divided into four banks, each 128 bytes in length. Each bank consists of general purpose registers and special function registers. The first 32 locations of each bank contain the special function registers, and the remaining 96 locations are for the general purpose registers. The **special function registers** are used by the CPU and the peripheral devices to control the operation of the MCU. Examples of special function registers include the tristate register (*TRISA*), the analog select register (*ANSEL*), and the timer1 control register (*TIMCON1*). The **general purpose registers** are used to store variables declared inside the code, and there are 256 of these registers. Note that while there are a total of 512 memory locations (4 banks × 128 location per bank), not all of the memory locations are implemented. Unimplemented data memory locations read as 0. The selection of a particular bank is done using the *RP0* and *RP1* bits of the *STATUS* register.

In the PIC18F4550 MCU, the data memory is portioned into 16 banks (256 bytes each) but with a different organization. Fifteen banks (bank 0 through 14) are used as general purpose registers, while one bank (bank 15) is used for the special function registers. When using a high-level compiler (such as C or Basic), the user does not need to know in which bank the data will be stored. On the other hand, when using assembly language, the user has to explicitly select the memory bank when writing or reading data.

4.7.2 EEPROM Data

The EEPROM Data can be used to store data at program time or during program execution. Storage time access during program execution takes longer time than writing to a RAM location. However, the data is permanently stored and will be maintained even if power is lost. The PIC-C compiler has the following functions for accessing EEPROM data:

```
read_eeprom(address)          //Read the data EEPROM at the specified memory location
write_eeprom(address, value)  //Erases and writes value to data EEPROM at the specified
                              //memory location
```

These two functions are designed for use during run time. The PIC-C compiler also has a pre-processor directive (*#ROM*) that can be used to access EEPROM data.

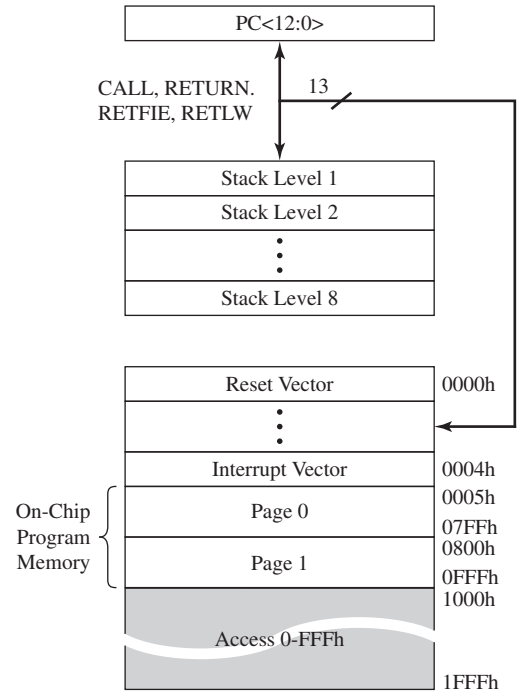
4.7.3 Program Memory

Program memory refers to the area on the chip that is used to store program instructions. The PIC16F690 has 7 Kbytes of program memory, while the PIC18F4550 has 32 Kbytes of program memory. Most of the chips in the PIC16 family have program instructions that are 14-bits wide, while those in the PIC18 family have program instructions that are 16-bits wide. The execution of program

Figure 4.16

Program memory and stack map on PIC16F690

(Reprinted with the permission of Microchip Technology Incorporated)



instructions is controlled by the program counter (PrC). On the PIC16F690, the PrC is 13 bits and can access 8K instructions. However, only the first 4K (0000h – 0FFFh) instructions are physically implemented on the PIC16F690, as seen in Figure 4.16. Accessing a location beyond this limit will cause a wraparound. The *Reset Vector* address (0000h) is the address of program start after the MCU is powered on or reset. The *Interrupt Vector* address (0004h) is the starting address of the interrupt service routine.

4.7.4 Delays and Timers

Compilers such as the PIC-C compiler provide built-in functions for timing delays. These include ***delay_ms(time)***, which delays execution up to 65535 ms; ***delay_us(time)***, which delays execution up to 65535 microsecond; and ***delay_cycles(count)***, which delays execution from 1 to 255 instruction clocks, where each instruction clock is equal to four oscillator clocks. All of the above functions work by executing a precise number of instructions to cause the requested delay. They do not use any timers. Note that no code can execute while one is waiting for the delay to expire with the exception of interrupt service routines. Time spent in an interrupt does not count toward the delay time. Furthermore, these functions cannot give us the time of occurrence of an event relative to some other point in the code (need to access timers directly).

The PIC16F690 has three **timers** referred to as Timer0, Timer1, and Timer2. More information about these timers is included in Chapter 6. **Timer1** uses a 16-bit counter, while **Timer0** and **Timer2** use 8-bit counters. Timers 0 and 1 can operate as either timers or counters. In the timing mode, the count value is incremented every *instruction* cycle (or at a multiple of it if a prescaler is used). In the counter mode, the Timer0 or Timer1 module will increment on every rising or falling edge of the external signal connected to the microcontroller. The user has a choice of setting the clock source for these timers, as well as setting the maximum overflow

interval. All three timers have a **prescalar**, which is a user-set factor that can divide the input clock frequency for that timer. For example, Timer1 has a choice of four prescale factors (1:1, 1:2, 1:4, and 1:8). Having a prescale factor higher than 1:1 increases the maximum counting interval before the counter overflow. Example 4.3 illustrates the use of the prescale factor.

Example 4.3 Timer Counting Interval

Using an 8 MHz internal clock, and 1:8 prescale factor, determine the maximum counting interval for Timer1 on the PIC16F690.

Solution:

The input clock frequency to Timer1 would be 0.25 MHz ($8 \times 1/4 \times 1/8$) or a counter resolution of 4 μs. The one-quarter factor is due to the fact that the instruction cycle frequency on all of the PIC chips is one-quarter of the clock frequency (F_{OSC}). Since Timer1 is associated with a 16-bit counter, the maximum counting interval is $2^{16} \times 4\ \mu s = 262144\ \mu s$ or 262.1 ms. Thus, with a 1:8 prescale factor, the Timer1 counter will overflow once every 0.2621 s.

Timer1 counter values are available by reading the low and high bytes of Timer1 registers (*TMR1L* and *TMR1H* registers). The operating mode of this timer is first set by writing to the Timer1 control register (T1CON).

Unlike timers 0 and 1, Timer2 has also a **postscaler** factor defined for it. A postscaler factor increases the time interval (by the postscalar factor) at which the interrupt flag bit in the peripheral interrupt request register (PIR1) is set due to Timer2 overflow. For example, if Timer2 overflows every 2.048 ms, then with a postscalar factor of 1:16, the interrupt flag bit will be set once every 32.768 ms.

Chapter 6 includes code listings for implementing a timing function that can return the time, since the timer was started as long as the timer was read often to prevent overflow errors.

4.7.5 PWM Timing and Duty Cycle

Timer2 is used to control the frequency of the PWM signal. The **frequency** of the PWM signal is given by

$$\text{PWM}_{\text{freq}} = (F_{OSC}/4)/((1 + \text{PR2}) \times \text{t2pres}) \quad (4.3)$$

where F_{OSC} is the oscillator frequency, *t2pres* is the prescaler factor for Timer2, and *PR2* is Timer2 period register value (0 to 255). For example, at a clock frequency of 8 MHz, a *t2pres* value of 4, and a Timer2 period register value of 255, the PWM frequency will be 1.953 kHz.

Using the PIC-C compiler, the command to setup Timer2 to obtain the above frequency is

```
setup_timer_2(T2_DIV_BY_4, 255,1)
```

where the first argument is the prescale factor, the second argument is the Timer2 period register value, and the third argument is the Timer2 postscaler value. Note that the Timer2 postscaler is not used in the determination of the PWM frequency. The **duty cycle** is given by

$$\textit{Duty\_cycle} = \text{value}/(4 \times (1 + \text{PR2})) \quad (4.4)$$

where *value* is the parameter written to registers CCPR1L and CCP1COM (or equivalently the parameter of the *set_pwm*_duty()* function in the PIC-C compiler). For a 50% duty cycle and a *PR2* value of 255, *value* is 512 in Equation (4.4). Note that the number of available discrete duty cycle values is dependent on the value of the *PR2* register. When *PR2* is 255, the PWM bit resolution is at a maximum at 10 bits, and 1024 discrete duty cycles values can be used. The bit resolution as a function of the PR2 register value is given by Equation (4.5).

$$\text{Resolution (in bits)} = \log(4(PR2 + 1))/\log(2) \tag{4.5}$$

Note that while the PIC16F690 has four PWM output channels, they all run from a single PWM generator. This means that the user cannot independently control the frequency and duty cycle of each of these channels. The user can however control which channel is used for PWM output (default channel is P1A on PIC16F690). This is done through the Pulse Steering Control (PSTRCON) register (see data sheet for details). PIC MCUs with the ECCP module (such as PIC16F690) can also be configured for full or half H-Bridge PWM control (see data sheet for details).

Example 4.4 illustrates the determination of the minimum and maximum PWM frequencies as well as the bit resolution.

Example 4.4 PWM Frequency

Determine the minimum and maximum PWM frequencies that can be achieved using an oscillator clock frequency of 8 MHz. What is the PWM bit resolution at the maximum frequency?

Solution:

From Equation (4.3), the minimum frequency is achieved with the largest value for *PR2* and *t2pres*. Using a value of 255 for *PR2* and 16 for *t2pres*, we obtain a PWM frequency of 488.28 Hz. If we need to have a lower frequency than 488.28 Hz, then we need to use an oscillator with a lower frequency. Similarly, the maximum frequency is obtained by setting *PR2* to 0 and using a *t2pres* value of 1. This gives a PWM frequency of 2MHz.

From Equation (4.5), the bit resolution at the maximum frequency is given as log(4)/log(2) = 2. This means that there are four possible duty cycle value settings (0 through 3). However, because the PWM period (0.5 μs) is the same as the instruction period (0.5 μs) in this case, the achievable duty cycles are only 0 and 100%. Note that if the duty cycle pulse-width setting results in a signal larger than the PWM signal period, then the PWM output will not change.

4.7.6 Watchdog Timer

The watchdog timer (WDT) is a special counter that resets the processor if it overflows. The purpose of the WDT is to cause the processor to reset if it times out in a lengthy operation. To prevent overflow, the program needs to periodically reset the counter associated with the watchdog timer. The operation of the WDT is controlled by the WDTCON register. The watchdog timer uses the internal low frequency clock (LFINTOSC typically running at 31 kHz) for its timing functions, which is independent of the internal high frequency oscillator clock. The overflow period is processor dependent. On the PIC16F690, the overflow period can range

```
////////////////////////////////////////////////////////////////////////////////
///                WDT_Test.c
///
///  This program test the WDT reset function
///
///   Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////

#include <16F690.h>                    //Include file for the particular chip used
#use delay(internal=8M)                //Use Internal 8 MHz- clock
#fuses INTRC_IO, NOMCLR, WDT           //Set clock mode to internal oscillator with
                                       // no clkout. Master clear pin is used for I/O. Enable WDT

void main(void)                         //Main function
{
  setup_wdt(WDT_2304MS);               //Set WDT to reset every 2304 ms
  while (2>1)
  {
  output_high(PIN_C0);                 //Set the selected pin to high
  delay_ms(1000);                      //Set a time delay of 1000 ms
  output_low(PIN_C0);                  //Set the selected pin to low
  delay_ms(1000);                      //Set a time delay of 1000 ms
  restart_wdt();                       //Restart WDT to prevent resetting
  }
}
```

Figure 4.17

Code listing for a program to illustrate WDT reset

from 1 ms up to 268 s, while on the PIC18F4550, the overflow period can range from 4 ms to 131 s.

The PIC-C compiler has built-in functions to control the WDT. These include the function *setup_wdt()* that sets up the WDT timer overflow period and the function *restart_wdt()*, which causes the WDT to restart. Figure 4.17 shows a code listing to illustrate the operation of the WDT. In the code, the WDT is set to overflow every 2304 ms after which an infinite loop is called, in which an LED is turned ON and OFF every 2 s. The WDT is restarted in every scan through the loop. Since the loop duration (2000 ms) is less than the WDT overflow period, the given program will cause the LED to flash as set in the code. However, if we change the delay interval when the LED is on from 1000 ms to 3000 ms, then the LED will remain on all the time. This is because the 3000 ms delay is longer that the WDT overflow period, so there is no chance to restart the WDT before it overflows. Thus, the WDT will keep resetting (i.e. causing the program to start from the beginning) every 2304 ms and the code will not be able to turn off the LED or execute the *restart_wdt()* statement.

4.7.7 Power Saving

PIC MCUs have a mode of operation called **sleep** mode which offers a very low current power-down mode. During sleep, the oscillator driver is turned OFF, and I/O ports maintain the status they had before the MCU went into sleep mode. If the watchdog timer was enabled, then the WDT will keep running, but its counter will be reset before the MCU goes to sleep. The program can wake up from a sleep mode through an external reset input on the MCLR pin, through a watchdog timer waking up, or through external or peripheral interrupts.

An MCU enters the sleep mode by issuing the *sleep* instruction in assembly or by calling the *sleep()* function in the PIC-C compiler. When the MCU wakes up

Figure 4.18

Code listing to illustrate sleep operation and wake-up

```
////////////////////////////////////////////////////////////////////////////////
///                 WDT_Sleep.c
///
///   A program to illustrate wake-up from sleep using the WDT
///
///   Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////
#include <16F690.h>                          //Include file for the particular chip used
#use delay(internal=8M)                      //Use Internal 8 MHz- clock
#fuses INTRC_IO, NOMCLR, WDT                 //Set clock mode to internal oscillator with
                                             // no clkout. Master clear pin is used for I/O.  Enable WDT
void main(void)                              //main function
{
 int8 i;
 setup_wdt(WDT_1152MS|WDT_TIMES_8 );         //Set WDT to reset every 1152 ms x 8
 for (i= 1; i<= 10; i++)
       {
       output_high(PIN_C0);                  //Set the selected pin to high
       delay_ms(1000);                       //Set a time delay of 1000 ms
       output_low(PIN_C0);                   //Set the selected pin to low
       delay_ms(1000);                       //Set a time delay of 1000 ms

       if (i == 3)
           {
           sleep();
           }
       }
}
```

due to other than external reset input, it continues execution at the instruction that follows the sleep instruction. Figure 4.18 shows a PIC-C code listing that illustrates waking up from sleep using the WDT. In the code, the WDT is configured to reset every 9.216 seconds before the code goes into a loop to turn ON and OFF an LED ten times. After the third iteration in the loop, the *sleep()* function is called. Before the MCU goes to sleep, it clears the WDT. When the WDT resets 9.216 seconds after clearing the WDT, it executes the instruction after the *sleep()* command. In this program, the LED will flash three times, then sleeps for 9.216 seconds, wakes up and continues with $i = 4$ in the loop. It will then flashes eight times (five times before the next WDT reset with $i = 4$ to 8 and three times after the WDT reset with $i = 1$ to 3) before it goes to the next sleep cycle. The program operation repeats every 24.432 (6 + 9.216 + 9.216) seconds.

4.7.8 A/E/USART

Many PIC chips have a built-in universal synchronous asynchronous receiver transmitter (USART), an enhanced USART (EUSART), or an addressable USART (AUSART) module. The USART module can be used for synchronous (data line and clock signal) and asynchronous (data line but no clock signal) serial communication. Serial synchronous communication is normally used to communicate with devices that do not have an internal clock for baud generation such as A/D or D/A devices or serial EEPROMs. These devices need a master synchronous device to provide the needed external clock. Asynchronous serial communication is used with devices that have their own internal clocks for baud generation. The EUSART has more features than a USART. These features include automatic detection and

calibration of the baud rate. Some PIC MCUs have an AUSART which allows the USART to ignore all data on the bus until a new address byte is present.

Note that while a USART provides the mechanism for converting parallel data into serial data and vice versa, additional hardware is needed to wire a PIC MCU to a PC for serial RS232 communication. This is needed since the PIC does not supply the voltages needed (up to +/− 15 volts) for serial communication with a PC. A commonly used device is the DS275 or the **MAX232/233** interface chip. The connection diagram using the MAX233 chip is shown in Figure 4.19.

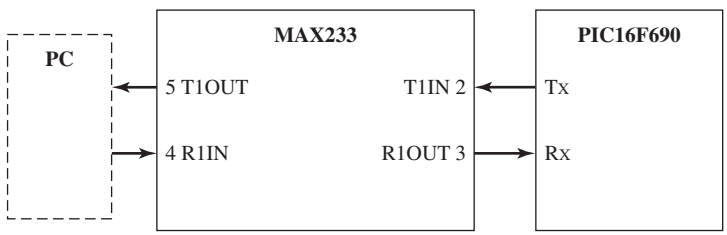


Figure 4.19

Wiring between a PIC16F690 and a MAX233

The CSS compiler has several functions for serial communication. These include the *getc()*, *gets()*, *putc()*, *printf()*, and *kbhit()* functions. The *getc()* gets a single character from the buffer. The *getc()* function is blocking, and thus, one should use first the *kbhit()* to determine if a character is available in the buffer before calling the *getc()* function. The *gets()* gets a string from the buffer until a carriage return is encountered. The *putc()* places a character in the output buffer, while *printf()* sends a formatted string to the buffer. The next chapter has more details about serial communication including example code.

4.7.9 Analog Comparator

Many PIC MCUs have an analog comparator module. The analog comparator compares two analog voltages and provides a digital output value that is an indication of the relative magnitude of the two analog voltages. This module is useful in interfacing analog-to-digital circuits. On the PIC16F690, there are two analog comparators called C1 and C2. Each comparator has two inputs labeled IN+ and IN−. When the analog input to the IN+ pin is higher than the analog input to the IN− pin, the output of the comparator is 1, otherwise it is zero. The analog comparator has several modes of operation, including simple comparison as discussed previously, synchronizing the comparator output with Timer1, and simultaneous read of comparator outputs to eliminate the timing skew of reading separate registers. The CSS compiler provides the function *setup_comparator()* to set up the analog comparator.

4.7.10 Synchronous Serial Port (SSP) Interface

To provide higher communication speeds, many PIC chips have a built-in synchronous serial port interface module. The PIC16F690 supports both the **Serial Peripheral Interface (SPI)** and the **Inter-Integrated Circuit (I^2C™)** interface. The SPI operates in full duplex and at speeds of 1 Mbps or higher. It is simple to implement (needs only four wires), and uses the concept of master/slave. The I^2C interface (pronounced I-squared-C) is a synchronous serial communication protocol that was developed by Philips Semiconductor. The I^2C or the Inter-Integrated Circuit (I^2C™) interface uses just two wires, one for data transmission and the other for the clock signal, for interface between two devices. Chapter 5 has more details about the programming and operation of these interfaces.

4.8 Interrupts

An interrupt is a mechanism in which a predefined event (such as the overflow of a hardware timer, the change of state in a digital input line, or the arrival of a character in a serial line) causes a program to stop execution after the current instruction, saves the state of the program, and then executes a predefined routine called an Interrupt Service Routine (ISR). After the ISR completes execution, the program resumes its operation at the next instruction. Interrupts are typically used in time critical applications to make the CPU take immediate action in response to situations such as alarm conditions. Interrupts are also used so as not to waste the computing resources in checking if an event occurs. The process of continuous checking is called **polling**. An example of a polling operation is the process of continuously reading a digital input pin to check if the pin has changed state.

4.8.1 Interrupts Applications

Typical applications of interrupts include the following.

- Use of timer overflow interrupts to schedule the execution of a code segment that needs to run periodically. Such code can be used to implement a digital feedback controller (see Chapters 6 and 9). For example, if we want our controller to run every *Tsamp* interval, then without the use of interrupts, the program has to repeatedly read a timer to determine if one *Tsamp* interval has elapsed since the last time the controller was called. If that is the case, then the controller is called again, and the process repeats. This is not very efficient. If a timer overflow interrupt was used instead, then the MCU will automatically execute the control code whenever the timer overflows. The control code will execute as part of the ISR. There is no need for the program to read the timer to check if the specified interval has elapsed. Note that the timer overflow interval has to be set as a function of the desired controller run interval or *Tsamp*.
- Use of a digital input line change of state interrupt to indicate a change in the output of a sensor. An example would be the use of a non-contact digital proximity sensor (see Chapter 7) to indicate the arrival of a part in an assembly line or the close proximity of a part to an obstacle. It is not very efficient to keep reading the sensor output value to see if the sensor output has changed state. With the use of a digital input line change of state interrupt, a change in the state of the sensor (i.e., from low-to-high or from high-to-low) will generate an interrupt which will automatically cause an ISR to execute to handle the change of state (such as starting or stopping a motion sequence or tuning ON/OFF a valve).
- Use of a serial line character receive (*EUSART Receive*) interrupt to read a character that was received by the serial port input buffer. In serial communication between a MCU and a PC (see details in next chapter), it is not known when characters will be transmitted from the PC to the MCU. Since the receive input buffer has a limited capacity, the MCU program has to continuously check the input buffer and read a character if it is available to prevent the buffer from overfilling and thus overwriting previously received characters. With the use of *EUSART Receive* interrupt, the arrival of a character over the serial line will cause the program to automatically execute an ISR to read the received character. There is no need for the program to continuously check the input buffer.

4.8.2 Interrupt Processing

On the PIC16F690, several sources can cause interrupts. These include:

Timer0 or Timer1 Overflow

PORTA or PORTB change

EUSART Receive and Transmit

External Interrupt on pin RA2

The PIC16F690 has several registers dedicated for interrupt processing. These registers are listed in Table 4.8.

Table 4.8

Interrupt registers on the PIC16F690

Register	Name	Function
INTCON	Interrupt Control Register	To enable individual and global interrupts. Also to record individual interrupt requests from INT pin interrupt, PORTA/PORTB change interrupts, and TMR0 overflow interrupt in flag bits.
PIE1,PIE2	Peripheral Interrupt Enable Register	To enable individual interrupts through their corresponding enable bits
PIR1,PIR2	Peripheral Interrupt Request Register	To record individual interrupt requests in flag bits from the remaining interrupt sources such as A/D, Timer1 overflow, comparator, etc.

To use interrupts, one needs to do the following.

1. Set the *global interrupt enable (GIE)* bit in the INTCON Register. Without enabling this bit, none of the unmasked interrupts are allowed to happen.
2. Enable (unmask) the interrupt(s) that you need to process such as timer1 overflow interrupt by writing to the appropriate register (such as INTCON, PIE1, or PIE2). Since there are many possible interrupt sources, the specific interrupt enable bits are spread over three registers: the *interrupt control register* (INTCON) and the two peripheral interrupt enable registers (PIE1 and PIE2).
3. Write an ISR or interrupt handler to handle each interrupt source.

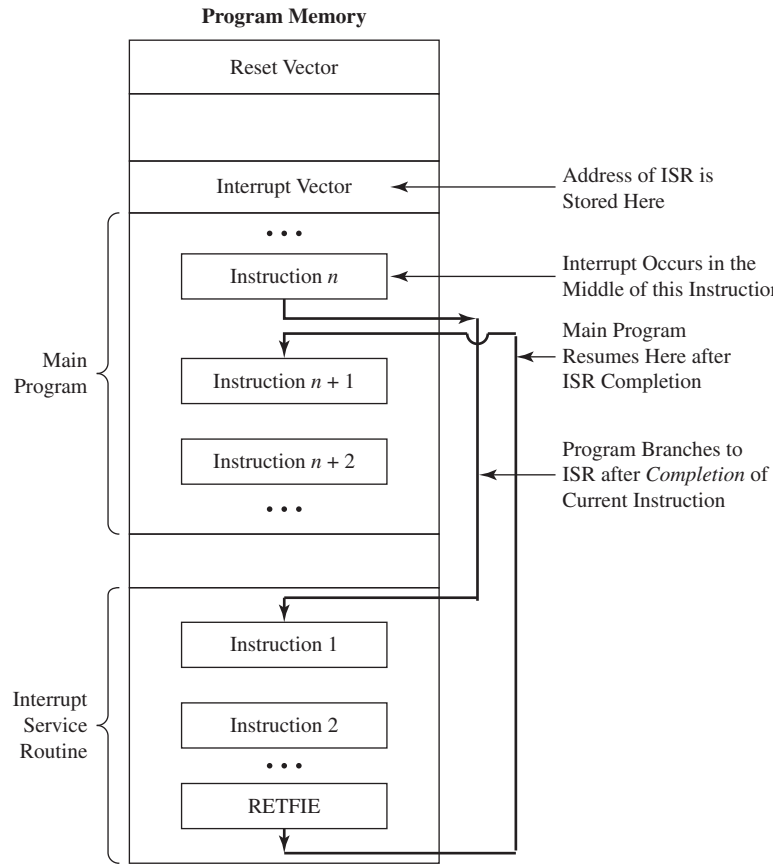
When an interrupt is serviced by the MCU, the GIE bit is cleared to disable any further interrupt. The address of the next instruction is pushed onto the stack, and the program counter is loaded with the address of the ISR which is stored in a specific memory location (0x0004 on PIC16F690). Figure 4.20 shows a graphical view of interrupt handling on a PIC MCU. It shows the interaction between the main program and the ISR in program memory.

Writing an ISR is a tricky process. If not done carefully, it will leads to unpredictable behavior. An ISR normally consists of four parts. The first part has code to save the status of the processer. In most MCUs, this means the values of the STATUS and W registers, as well as registers related to the PrC. These registers are not saved by the MCU. The values of these registers should be saved, because there is a possibility that when the interrupt executes it will modify them. Once the ISR completes its work, the original register values should be restored, and the program can continue to do what it was doing before the interrupt happened.

The second part of the ISR is to check and clear the *Interrupt Request* bit on the INTCON, PIR1, or PIR2 registers. Checking the *Interrupt Request* bit can tell which source caused the interrupt if more than one interrupt source was enabled. While some processors allow a dedicated ISR for each interrupt source, some do not allow that, and thus, checking the *Interrupt Request* bits is needed. The set

Figure 4.20

Typical interrupt handling on a PIC MCU



Interrupt Request bit should be cleared to avoid multiple interrupt requests once the interrupts are re-enabled.

The third part of the ISR is to have code to handle that specific interrupt such as incrementing a counter, initiating control action, or setting a flag. The fourth part of the ISR should restore the state of the registers, plus code to inform the processor that interrupt handling was completed. In assembly, this is done by executing the RETFIE instruction. The RETFIE instruction exits the ISR and sets the GIE bit, which re-enables interrupts.

The handling and processing of interrupts depends on the chip used. The PIC16F690 MCU supports only basic interrupt processing (such as discussed previously). Some chips (such as those in the PIC18 or PIC 24 families) have advanced interrupt features (such as the ability to setup a **priority level** for each interrupt and the use of interrupt vectors). In the PIC18F4550 MCU, each interrupt source can be assigned a high-priority level or a low priority level. A high-priority interrupt source can interrupt a low-priority interrupt. Also, low-priority interrupts are not processed, while high-priority interrupts are being serviced. Due to the use of priorities, each interrupt source has three bits to control its operation. These include the flag bit and the enable bit that were discussed before, plus a priority bit to select the priority level. There are also two global interrupt enable bits: GIEH to enable all interrupts with a high-priority levels, and GIEL to enable all interrupts with a low-priority level. When an interrupt occurs, the ISR will execute at one of two predefined addresses, depending on the priority bit setting. If the interrupt priority feature is disabled (default

setting), interrupt processing is compatible with the PIC mid-range devices (such as PIC16F690), and all interrupts branch to just one address.

4.8.3 PIC-C Interrupts Handling

The CCS compiler provides several functions for interrupt processing. These include *disable_interrupts()*, which disables a specified interrupt, and *enable_interrupts()*, which enables a specified interrupt. The #*int_xxx* directive (where *xxx* is the specified interrupt name) is placed before the code listing for the interrupt service routine to inform the compiler to use the following function with interrupts associated with the specified interrupt name. We will show in this section the PIC-C code listing for two different types of interrupts. Figure 4.21 shows

Figure 4.21

Code listing for Timer0 overflow interrupt using the PIC-C compiler

```
////////////////////////////////////////////////////////////////////////////
//              Timer0int.c
//
// A program to illustrate timer0 overflow interrupt
// Compiler: PCWH from CS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////
#include <16F690.h>
#fuses INTRC_IO,NOWDT,NOPROTECT,NOMCLR
#use delay(clock=8M)
#define INT_PER_2SECONDS 61      // ((8000000*2)/(4*256*256))
int8 seconds;                    // Seconds counter
int8 int_count;                  // Number of interrupts left before 2 seconds has elapsed
int8 LED_state = 0;              // Flag to keep track of the LED state
#int_timer0                      // This ISR function is called every time
void clock_isr() {               // timer0 overflows (255->0).
                                 // For this program this happens 30.5 times/sec (or 61 times/2 sec)
   if(--int_count==0)            // Check if interrupt counter is zero
     {
     seconds = seconds + 2;      // Increment seconds counter
     if (LED_state == 0)         // Turn on LED if it was off
     {
     output_high(PIN_C0);
     LED_state = 1;
     }
     else
     {
     output_low(PIN_C0);         //Turn off LED if it was on
     LED_state = 0;
     }
     int_count= INT_PER_2SECONDS;//Reload number of interrupts per 2 second
   }
}

void main()
{
  int_count=INT_PER_2SECONDS;
  set_timer0(0);                 // Initialize timer0 to zero
  setup_timer_0(RTCC_INTERNAL | RTCC_DIV_256 );   // Set timer0 to use internal clock with a prescale of 256
  enable_interrupts(INT_TIMER0); // Enable timer0 interrupt
  enable_interrupts(GLOBAL);     // Activate timer0 interrupt

  while ( 2> 1)                  // Start an infinite loop
  {
  ;                               // Do nothing
  }
}
```

a code listing for **Timer0 overflow interrupt**. The code alternatively flashes an LED ON and OFF every two seconds. The timing interval is kept by a variable that is decremented whenever the interrupt occurs. The frequency of the interrupts is controlled by setting the timing parameters for Timer0. Note that Timer0 is also referred to as the real-time clock (RTTC). In the code shown in Figure 4.21 for an 8 MHz clock and a prescale value of 256, the Timer0 clock frequency is 7812.5 Hz. Since the interrupt occurs when the 8-bit Timer0 overflows, then the interrupts are generated at the rate of 30.52 interrupts per second or (7812.5/256).

Figure 4.22 shows a code listing for **RA2/INT external interrupt**. This interrupt is edge triggered, occurring on the rising or falling edge of the signal connected to the RA2/INTpin. In the code, the interrupt is set to occur on the falling edge. The ISR for this interrupt makes an LED (pin RC0) flash whenever the interrupt is detected. Obviously, this code can be replaced by some other action that needs to be performed.

Figure 4.22

Code listing for RA2/INT external interrupt using the PIC-C compiler

```
////////////////////////////////////////////////////////////
//                Interrupt_EXT.c
//
//  This program illustrates external interrupts on INT/RA2 pin
// Compiler: PCWH from CS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////
#include <16F690.h>                 // Include file for the particular chip used
#use delay(internal=8M)             // Use Internal 8 MHz- clock

#fuses INTRC_IO, NOMCLR             // Set clock mode to internal ocsillator with
                                    // no clkout. Master clear pin is used for I/O

#INT_EXT                            // The ISR function
void ext_isr()
{
    output_high(PIN_C0);            // Set the selected pin to high
    delay_ms(500);                  // Set a time delay of 500 ms
    output_low(PIN_C0);             // Set the selected pin to low
    delay_ms(500);                  // Set a time delay of 500 ms
}

// Main program
void main()
{
  ext_int_edge(H_TO_L);             // Set interrupts to occur on H_TO_L
  enable_interrupts(INT_EXT);       // Enable interrupts for external INT pin
  enable_interrupts(GLOBAL);        // Activate interrupts


  while(TRUE)                       // Start an infinite loop
  {
    ;                               // Do nothing
  }
}
```

Note that due to the use of C-language to handle the interrupt, all low-level interrupt processing activities (such as checking and resetting of the interrupt flag bits) are handled by the compiler and are not explicitly shown in the code.

4.9 Assembly Language Programming

As mentioned before, assembly language is a low-level programming language that is specific to the microcontroller used. It is more difficult to program the code in assembly language, but the user has better control on the execution timing of the code, and can also get more compact code in terms of file size in hex.

4.9.1 Assembly Instructions

For the PIC16F690, each assembly instruction is 14-bit word that consists of two parts:

Opcode: Specifies the instruction type.

Operand: One or more values that specify the operation of the instruction.

Table 4.9 shows a list of some assembly instructions along with an explanation of the operation of these instructions. Note that many of the assembly instructions make reference to the W-register and the f-register. The **W-register** is the accumulator register or the working register. There is only one present in the system. The **f-register** refers to a 7-bit file register address. The file register is a designation for any data (a general purpose register) or any special function register (such as the STATUS or TRISB registers). The term 'literal' is also used in some of the commands, which is the term used to designate a numerical value or label (i.e., 10 or 207).

Some of the assembly instructions change the status of certain bits in the STATUS register. These **status bits** are defined below:

C—Carry bit (bit 0 of STATUS Register): This bit is set to 1 when a carry-out occurs from the most significant bit of the result and is set to 0 when no carry-out occurs.

DC—Digit carry bit (bit 1 of STATUS Register): This bit is set to 1 when a carry-out occurs from the 4th low-order bit of the result and is set to 0 when no carry-out occurs.

Z—Zero bit (bit 2 of STATUS Register): This bit is set to 1 when the result of an arithmetic or logic operation is zero and is set to 0 when the result is not zero.

$\overline{\text{PD}}$—Power down bit (bit 3 of STATUS Register): This bit is set to 1 after power-up or after executing a clear watchdog timer instruction (CLRWDT) and is set to 0 by execution of the SLEEP instruction.

$\overline{\text{TO}}$—Time out bit (bit 4 of STATUS Register): This bit is set to 1 after power-up, CLRWDT, or SLEEP instruction. The bit is set to 0 after a time-out from the watchdog timer.

The status bits are useful in performing comparison operations or getting status information. For example, after performing a subtraction operation using the SUBWF instruction, we can check the value of the carry bit (C) to determine which of the two values that are being subtracted is larger than the other. Similarly, the 0 bit (Z) can be used, for example, to indicate if a variable that is being incremented (using the INCF instruction) has overflowed.

4.9.2 Assembly Language Programming Examples

To illustrate assembly language programming, consider the operation of adding the values of two variables and storing the result in one of the variables. If we are using VB or C-language, we can write this operation as

```
VALUE2 = VALUE1 + VALUE2
```

Table 4.9

A listing of some of the assembly instructions for the PIC16F690

Assembly Instruction Syntax	Description	Example Syntax	14-bit Code MSB LSB	Status Bits Affected
ADDWF f,d	Add the contents of the file register 'f' to the W-register. If 'd' is 0, store the result in the W-register. If 'd' is 1, then result is stored in the f-register.	ADDWF VALUE,0	00 0111 dfff ffff	C, DC, Z
BCF f,b	Clear bit 'b' in file register 'f'.	BCF STATUS,5	01 00bb bfff ffff	
BSF f,b	Set bit 'b" in file register 'f'.	BSF TRISA,0	01 01bb bfff ffff	
BTFSC f,b	Test bit "b" in file register "f". If bit "b" is "1", the next instruction is executed. If bit "b" is "0", the next instruction is skipped and a NOP is executed making this a two-cycle instruction.	BTFSC STATUS,2	01 10bb bfff ffff	
CALL k	Call subroutine defined by 11-bit variable k.	CALL 2000	10 0kkk kkkk kkkk	
CLRF f	Clear the contents of the file register 'f'. The Z bit in the STATUS register is set after this operation.	CLRF PORTA	00 0001 1fff ffff	Z
DECFSZ f,d	Decrement the contents of the file register 'f'. If 'd' is 0, store the result in the W-register. If 'd' is 1, then the result is stored in the f-register. If the result is '1', the next instruction is executed. If the result is '0', then a NOP is executed instead. In this case, it becomes a two-cycle instruction.	DECFSZ VALUE,1	00 1011 dfff ffff	
GOTO k	Perform an unconditional branching to a label defined by 11-bit variable k.	GOTA 1100	10 1kkk kkkk kkkk	
INCF f,d	Increment the contents of the file register 'f'. If d' is 0, store the result in the W-register. If 'd' is 1, then the result is stored in the f-register.	INCF VALUE,1	00 1010 dfff ffff	Z
MOVF f,d	Move the contents of the file register 'f'. If 'd' is 0, the destination is the W-register. If 'd' is 1, the destination is the file register 'f'-itself.	MOVF VALUE,0	00 1000 dfff ffff	Z
MOVLW k	Move an 8-bit literal 'k' to the W-register	MOVLW 100	11 00xx kkkk kkkk	
MOVWF f	Move data from the W-register to the file register 'f'	MOVWF VALUE	00 0000 1fff ffff	
NOP	No operation	NOP	00 0000 0xx0 0000	
SUBWF f,d	Subtract using 2's complement method the contents of the W-register from the f-register. If 'd' is 0, store the result in the W-register. If 'd' is 1, then result is stored in the f-register.	SUBWF VALUE,1	00 0010 dfff ffff	C, DC, Z

Assembly Instruction	Comments
MOVF VALUE1, 0	Move the contents of the file register, *VALUE1*, to the W-register
ADDWF VALUE2, 1	Add the contents of the file register, *VALUE2*, to the W-register, and stores the result in the file register, *VALUE2*

Table 4.10

Listing of assembly code to add two variables

where the two variables are defined as VALUE1 and VALUE2. In assembly language, this operation is programmed as shown in Table 4.10 (we are assuming each variable to be an 8-bit in size). Here, the addition of the two numbers is performed after one of the variables was transferred to the W-register, which is part of the CPU of the microcontroller.

As another example, consider the following Visual Basic Express (VBE) code, which compares the values of two variables and performs a call to subroutine *Sub1* if the value of one of the variables is equal to the other.

```
IF (VALUE2 = VALUE1) Then
   Call Sub1
ENDIF
```

This code can be programmed in assembly language, as shown in Table 4.11. In the code listing, after performing the subtraction operation, the 0 bit in the status register is set to 1 if *VALUE2* is equal to *VALUE1* and to 0 if otherwise. Since the 0 bit is set to 1 only if *VALUE2* = *VALUE1*, the CALL statement is not skipped if *VALUE2* = *VALUE1*. In that case, the program calls subroutine *Sub1*.

Assembly Instruction	Comments
MOVF VALUE1, 0	Move the contents of the file register, *VALUE1*, to the W-register
SUBWF VALUE2, 0	Subtract the contents of the W-register from the variable *VALUE2* (i.e. perform *VALUE2-VALUE1*)
BTFSC STATUS, 2	Perform a test on the zero bit, and skips the next statement if the bit is clear.
CALL Sub1	The code will execute this statement if the zero bit is set

Table 4.11

Listing of assembly code to perform comparison and branching

Assembly language does not have a command to perform a *Do-Loop* for a certain number of operations. A *Do-Loop* is implemented using a combination of several instructions. For example, the following VBE *Do-Loop* is implemented in assembly using the code listed in Table 4.12.

```
For I = 1 to 200
   Call Sub1
Next I
```

The first two instructions place the loop counter (200 in this case) in the variable *I*. In each iteration of the code that follows the *Loop1* label, the variable *I* is decremented by 1. If the result of the decrement operation is not 0, the code branches to label *Loop2*, which calls the subroutine *Sub1*. When *I* reaches 0, the code branches to the label *ExitLoop*, at which point the *Do-Loop* code is no longer executed.

Table 4.12

Listing of assembly code that performs a *Do-Loop*

Assembly Instruction	Comments
MOVLW 200	Move the number 200 to the W-register
MOVWF I	Move the contents of the W-register to the variable I (loop counter)
Loop1:	Label Loop1
DECFSZ I,1	Decrement the contents of the variable I and place the result back in the I variable.
GOTO Loop2:	Branch to label Loop2 if the result of the previous instruction is not zero.
GOTO ExitLoop:	Branch to label ExitLoop when the I variable becomes 0.
Loop2:	Label Loop2
CALL Sub1	Call Sub1
GOTO Loop1:	Branch back to Loop1
ExitLoop:	Label ExitLoop

Figure 4.23

Listing of an assembly program that turns on an LED

```
bsf     STATUS,5        // Select Register Page 1 by writing to the RPO bit
bcf     TRISC,0         // Make IO Pin C0 an output
bcf     STATUS,5        // Back to Register Page 0
bsf     PORTC,0         // Turn on LED C0
```

To illustrate assembly language for performing I/O operations, consider the code shown in Figure 4.23. The figure has a listing of a program that turns on an LED that is connected to pin 0 on port C. Note that in this example, because the TRISC register and the PORTC registers are on different memory banks (see data sheet), the appropriate memory bank is first selected before writing to the particular register on that bank (we do not have to worry about this if we used C-language). This is due to the limitation that only 7 bits are allowed to define a file register address, while there are possibly 512 file register locations spread over four pages (or memory banks).

In the PIC16 family, all assembly instructions take either one or two instruction cycles to execute, where as mentioned before, the instruction cycle frequency is one-fourth of the clock frequency. Conditional instructions (such as *BTFSS* or *DECFSZ*) take two instruction cycles to execute when the conditional test is true and one instruction cycle when the conditional test is false. When the conditional test is true, the NOP instruction is executed in the second cycle, while the PrC loads the address to be branched to.

4.9.3 Integrating C and Assembly

There are two ways to incorporate assembly code. In the first way, you write the assembly code in a separate file (with the *.asm extension), compile with an appropriate compiler (such as MPLAB), and then download the hex file to the chip. In the other way, you can add assembly instructions to a C or VB code, because many compilers allow the integration of assembly commands with a high-level programming language in the same file.

```
////////////////////////////////////////////////////////////////////////////////////////
///                    AssemblyForLoop.c
///
///    This program illustrates the incorporation of assembly code
///    into c-program. The code below turns on pin_c0 when I reach 0
///    Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////////////
#include <16F690.h>
#fuses INTRC_IO,NOWDT,NOPROTECT
#use delay(internal=8M)

// main program that executes assembly routine to decrement the variable I
void main()
{
int8 I;

#asm                          // Start assembly code
       MOVLW   200            // Move 200 into the W-register
       MOVWF    I             // Move the contents of the W-register to variable I
 Loop1:  DECFSZ I,1           // Decrement I and place the result back in I variable
       GOTO  Loop2            // Branch to label Loop2 if the result of the previous instruction is not zero
       GOTO ExitLoop          // Branch to label ExitLoop when I becomes 0
 Loop2:  NOP                  // Do nothing (NOP operation) to simulate a function call
       GOTO Loop1             // Branch back to Loop1

 ExitLoop:                    // Label ExitLoop
#endasm                       // End assembly code

  if (I == 0)                 // Turn on pin C0 when I reaches zero
       output_high(Pin_C0);
}
```

Figure 4.24

PIC-C code incorporating assembly code to perform a *Do-Loop* operation

In the PIC-C compiler, assembly instructions are added to a C-file by placing them between the *#asm* and *#endasm* directives. As an example, the assembly code listing for the *Do-Loop* operation of Table 4.12 is incorporated into a C-code, as shown in Figure 4.24. The variable *I* is declared in the C-code, as an 8-bit integer. In the assembly code, *I* is assigned a value of 200 and then gets decremented 200 times. The NOP instruction (used to simulate a function call) is executed in each iteration of the loop when *I* is greater than 0. To check the result of the *Do-Loop*, bit C0 is turned ON in C-code that follows the assembly code if *I* is equal to 0. One feature of the PIC-C compiler is that it also can provide an assembly listing for any C-program. The assembly listing is accessed in the compile menu under the C/ASM tab.

4.9.4 PIC18 Assembly Instructions

In the PIC18 family, most of the assembly instructions are 16-bit wide, while a few of them are 32-bit wide. In addition, there many more instructions in the PIC18 family (about 70) than in the PIC16 family (about 35). Table 4.13 lists some of the assembly instructions that are only available in the PIC18 family. The additional instructions include those to perform a comparison of the f- and W-registers (such as CPFSEQ), to multiply the contents of the WREG and the f-registers (MULWF), to perform branching (such as BC, BNC, and BRA), to perform software reset (RESET), and to perform stack operations (POP and PUSH). In addition, the PIC18 instruction set includes instructions to perform program memory read and write operations (such as TBLRD and TBLWT).

Table 4.13

List of additional assembly instructions in the PIC18F family

PIC18 Instruction	Operation
CPFSEQ	Compare f-register with WREG, skip if equal
CPFSGT	Compare f-register with WREG, skip if greater than
MULWF	Perform unsigned multiplication of the contents of the WREG and f-registers
NEGF	Negate f-register
BTG	Bit toggle the f-register
BC	Branch if the carry bit is 1
BN	Branch if the negative bit is set to 1
BNC	Branch if the carry bit is 0
BRA	Unconditional branch
RESET	Perform a software reset. It performs the same action as an MCLR Reset.
POP	Pop top of return stack
PUSH	Push top of return stack
TBLRD	Table read. It reads the contents of program memory.
TBLWT	Table write. It writes to program memory

4.10 Chapter Summary

This chapter focused on PIC microcontrollers. A microcontroller is a *single* chip device that has a processor, memory, and interface devices located on the same chip. The chapter focused on the PIC16F690 microcontroller but also discussed some of the features of the PIC18 family of microcontrollers. This chapter started by discussing the different numbering systems that are used in programming and interfacing of microcontrollers. It then covered the basic elements of a microcontroller (such as clock sources, different memory areas) and basic interface devices (such as digital I/O and A/D convertor). The chapter presented the PIC C-programming language from CCS, Inc., a high-level programming language to program PIC microcontrollers. A high-level programming language simplifies the programming of microcontrollers, since all of the low-level hardware details are taken care of by the compiler. This chapter also discussed methods to download programs to the MCU using the PICKit 2 or PiCKit 3 programmer. In addition, the chapter covered many of the features and devices on a PIC MCU. These features include timing, EEPROM memory, PWM actuation, comparator, watchdog timer, power saving, and serial interfacing. This chapter also covered interrupt processing as well as the use of assembly language in programming a microcontroller. Interrupts are typically used in time critical applications to make the CPU take immediate action in response to situations such as alarm conditions. Interrupts are also used in order to not waste the computing resources in checking if an event occurs. Assembly language is a low-level programming language that is specific to the microcontroller used, which gives the user better control on the execution timing of the code and more compact code.

Questions

4.1 What distinguishes a microcontroller from a microcomputer?

4.2 Where are program instructions stored in a microcontroller?

4.3 Which bus handles the transfer of data between CPU and memory in a PIC microcontroller?

4.4 How many instructions can the PIC16F690 MCU store in program memory?

4.5 What are the minimum connections needed for a PIC MCU to operate?

4.6 How is a PIC MCU 'programmed'?

4.7 Name three external oscillator sources.

4.8 Why it is desirable to operate an MCU at a high oscillator speed?

4.9 What is the advantage of storing data in program memory?

4.10 What register controls the direction of I/O operations in a PIC MCU?

4.11 What is the difference between a prescaler and a postscaler?

4.12 What feature on a PIC MCU allows the comparison between two input signals?

4.13 What happens after a power-on reset?

4.14 What is the purpose of a watchdog timer?

4.15 What happens when a PIC MCU goes to 'sleep'?

4.16 Why can't an MCU be connected directly to the serial port on a PC?

4.17 What is an interrupt?

4.18 What are the advantages of programming in assembly language?

PROBLEMS

P4.1 Convert the following decimal numbers to hexadecimal and binary. Do not use a built-in function to do the conversion.

a. 22

b. 184

c. 630

P4.2 Find the 2's complement representation for the following numbers using an 8-bit field.

a. −1

b. −43

c. −121

P4.3 Perform the following binary operations.

a. 1011 + 0010

b. 00011101 + 01001111

c. 1010 − 0011

P4.4 Find the binary representation for the following numbers using the IEEE 742 standard.

a. 0.078125

b. −0.5

c. 10.5

P4.5 Using the Microchip website, select one or more 8-bit MCUs with a small number of pins that is suitable for the following applications.

a. Monitoring of four digital input lines and writing to five digital output lines.

b. Monitoring and updating 10 digital I/O lines, reading four analog inputs, and communicating with a PC using RS-232 protocol.

c. Same as part 'b' but also using 3 PWM lines.

P4.6 Using the PIC16F690 MCU, draw an interface diagram for an application that requires the following.

a. Reads four digital I/O lines

b. Writes to four digital I/O lines

c. Reads two analog signals

d. Uses the internal clock as the clock source

e. Uses the comparator feature to compare two signals and to turn ON an LED if one is larger than the other

Be sure to identify and label all the pins on the chip that need to be used.

P4.7 Using the PIC16F690 MCU, draw an interface diagram for an application that requires the following.

a. Reads two digital I/O lines

b. Writes to four digital I/O lines

c. Reads two analog signals

d. Sends one PWM signal

e. Uses RS-232 serial communication

f. Uses crystal resonator as the clock source

Be sure to identify and label all of the pins on the chip that need to be to be used.

P4.8 Determine the maximum counting interval for Timer1 on the PIC16F690 using 1:4 prescale factor and a 20 MHz external clock.

P4.9 Determine the parameters of Timer2 on the PIC18F4550 to enable a PWM operation at a frequency of 5 kHz and a duty cycle of 25%. Assume a clock frequency of 10 MHz.

P4.10 A simplified block diagram of PWM operation on some PIC MCUs is shown in Figure P4.10. Using this diagram and discussion about PWM operation in this chapter, explain how the PWM signals are generated.

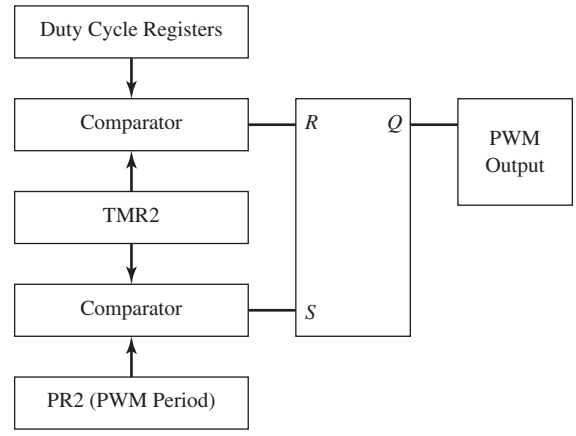


Figure P4.10

P4.11 Determine the parameters of Timer0 on the PIC16F690 so that the Timer0 overflow interrupts will occur approximately every 1 ms. Assume the microcontroller is used at a clock frequency of 8 MHz.

P4.12 Draw a circuit for interfacing a digital I/O on a PIC MCU to a MOSFET transistor that switches a small 12 V motor ON and OFF.

P4.13 Draw a circuit for interfacing the PIC16F690 to the following components.

a. An LED that is turned ON/OFF by the MCU

b. A NO push-button switch that is read by the MCU

c. A rotary potentiometer that is used to set the desired operating value for a control system

P4.14 Research and identify three features of some PIC microcontrollers for saving power. Explain how each feature saves power.

P4.15 Write out the result of separately performing each of the two assembly commands: *complement* and *negate* on the following register values.

a. 0x2a

b. 0x7d

P4.16 Write assembly code that shows how to implement a timing delay by using loops combined with the NOP statement.

LABORATORY/PROGRAMMING EXERCISES

L/P4.1 This problem requires the availability of a development board with built-in LEDs and a rotary pot (such as Microchip low pin-count board, Microchip PIC18 Explorer board, or Olimex PIC-STK-USB board). Develop and download a C-program for the microcontroller that does the following.

a. Turn ON and OFF the LEDs on the board in a particular pattern (one ON and the next OFF, two ON and the next two OFF, etc.). You can create any pattern you like.

b. Use the A/D reading from the pot input (for example, Channel RA0 on the low pin-count board) to vary the timing speed of the pattern. Turning the pot CW (as seen from above) should cause the pattern to turn ON and OFF more rapidly. Use a delay function (such as *delay_ms*() in PIC-C compiler) function to create the timing delay.

L/P4.2 Write a simple program to test the PWM feature on the PIC16F690 or any other PIC MCU. Connect the output of the PWM to a scope and monitor the output. What are the minimum and maximum PWM frequencies that can be achieved using the 8 MHz internal clock? Add a loop in the code to generate PWM signals with increasing or decreasing duty cycles.

L/P4.3 Using code that accesses Timer0 on any PIC board, write a C-program to time the turning ON and OFF of an LED with intervals ranging from 1 to 5 seconds. Do not use any of the built-in delay functions to solve this problem.

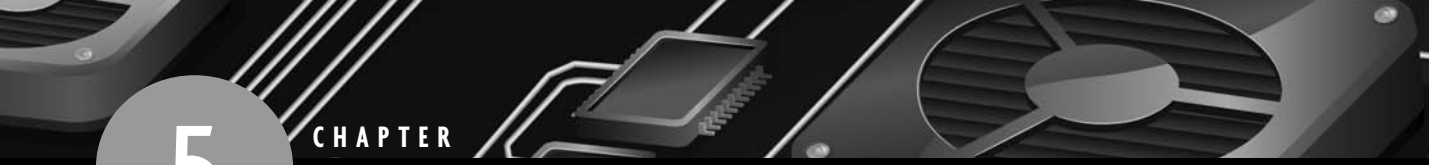
L/P4.4 Redo Lab Programming Exercise 4.3, but use a timer overflow interrupt to handle the timing (i.e., do not poll the timer to check if the desired timing interval has elapsed).

L/P4.5 Write a C-program that integrates assembly code to check the following relationships between V_1 and V_2, where V_1 and V_2 are 8-bit variables defined in the C part of the code. If the condition is true, turn on an LED to indicate that.

a. V_1 is equal to V_2

b. V_1 is less than V_2

L/P4.6 Design and build a circuit to interface a PIC16F690 or another MCU to an LED, a push-button switch, and a rotary potentiometer. Specify any resistors needed plus the wiring of all needed pins. Test your circuit by writing a simple program to turn ON the LED if the value read by the rotary potentiometer exceeds a specified value. Your code should turn OFF the LED whenever the push button is pressed.



CHAPTER 5

Data Acquisition and Microcontroller/PC Interfacing

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- Explain the difference between analog and digital signals
- Determine the requirements for proper sampling of analog signals
- Determine the voltage resolution, digitizing accuracy, and input and output values of an analog-to-digital convertor
- Determine the voltage resolution, digitizing accuracy, and input and output values of a digital-to-analog convertor
- Explain how to set or read a particular bit in a parallel port
- Explain RS-232 communication and develop code using this interface method
- Outline the I^2C and SPI interfacing methods and develop code using these interfacing methods
- Explain the serial-client model for Internet connection and develop a PC-based application that uses Internet interfacing
- Explain the USB Interface and USB code for PIC microcontrollers using the CDC class

5.1 Introduction

The heart of any mechatronics system is a computer or an embedded processor that is connected to the actuators and sensors that are part of that system. To do any useful work, data must be transferred back and forth between the processor and these components. The data can be in the form of analog or digital signals. Analog signals are continuous signals that can have any value over a certain range, while digital or discrete signals are discontinuous signals that have few specific values. The term 'analog' means analogous or similar, so a pressure transducer with an analog output maps the time-varying pressure measured by the sensor to an analogous time-varying voltage signal. Examples of analog voltage signals include the voltage signals supplied from a power company, the voltage applied to drive DC-type electric motors (see Chapter 8), and the voltage output of sensors such as thermocouples and tachometers (see Chapter 7). An example of a digital signal is the output from a digital displacement measurement sensor (such an encoder, see Chapter 7) which produces two output levels. Digital signals are found in all microprocessor circuits. Digital signals are better in handling noise which can affect the resolution of analog signals. Converting a signal from the analog domain to the digital domain

requires the use of an analog-to-digital converter (A/D). Similarly, converting a signal from the digital domain to the analog domain requires the use of a digital-to-analog (D/A) converter.

This chapter discusses techniques to interface a processor to the outside world using different interface devices (such as analog-to-digital converters, digital-to-analog converters, parallel ports, asynchronous and synchronous serial ports, network connection, and USB). In Chapter 4, we discussed interface devices on a PIC microcontroller (such as the A/D converter, the USART, and the I^2C/SPI). In this chapter, more information is given on the operation and programming of these devices. Interfacing is important for the operation of control systems, because a control system interacts with sensors and actuators through these interface devices.

5.2 Sampling Theory

In converting an analog signal to a digital signal, the analog signal is 'sampled' to obtain the digital values. By sampling, we mean that the analog signal is read at defined time instances and the continuous-time signal is replaced by a sequence of numbers [13]. We must be careful in performing the sampling operation so that the analog signal characteristics do not get distorted in the sampling process. The requirement for proper sampling is given by Shannon's sampling theory [14], which states that the sampling frequency should be at least twice that of the highest frequency in the signal. Otherwise, distortions (or aliasing) in the sampled signal will occur. Thus, a 1000-Hz sinusoidal analog signal should be sampled at a frequency of 2000 Hz or higher. In practice, a sampling rate of at least five times higher than the highest frequency in the signal is typically used.

To illustrate signal aliasing, Figure 5.1 on the next page shows a 1-Hz sinusoidal signal and the corresponding sampled signals at three different frequencies: 5 Hz, 2.1 Hz, and 1.25 Hz. Note the distortion in the signal sampled at 1.25 Hz, since it is below the minimum frequency specified by Shannon's sampling theory.

5.3 Analog-to-Digital Converter

An analog-to-digital converter is a hardware device for converting analog signals to digital signals. To prevent variation in the input signal from affecting the output while the conversion is taking place, the analog signal is first passed through a **sample and hold circuit** (which holds the input voltage) before it is converted.

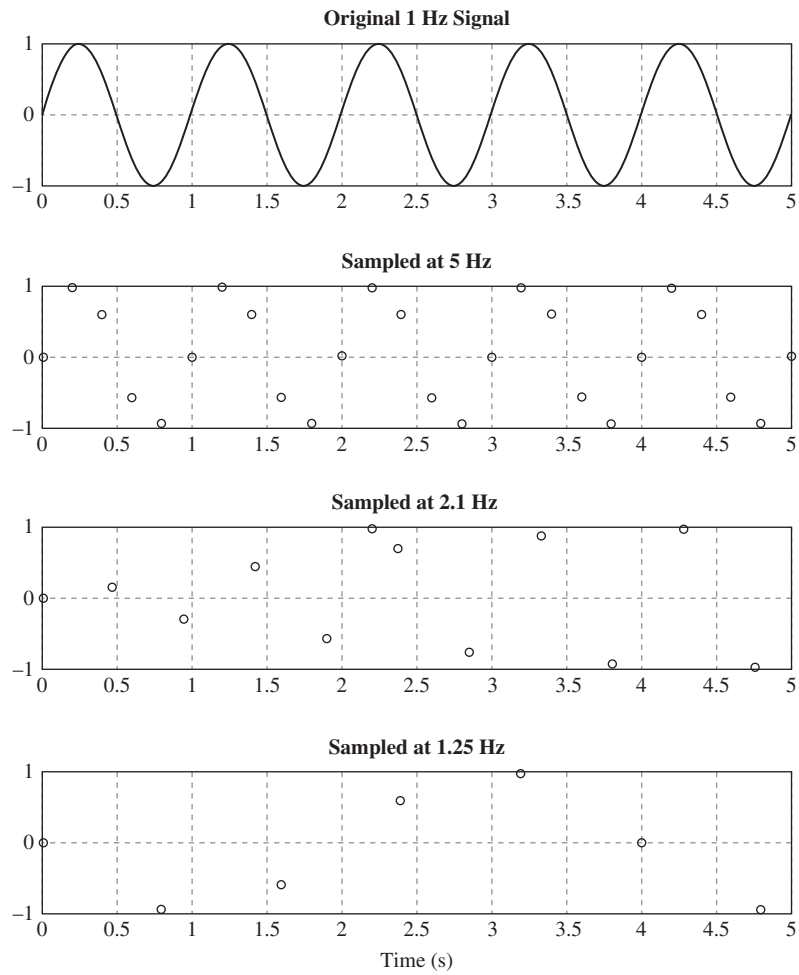
Many microcontrollers have several A/D channels. On a PC, the A/D converter is packaged with a digital-to-analog converter and a parallel port to form a data acquisitions card. The card is placed in one of the available computer slots, and a cable then is used to connect the card to an interface board commonly known as a distribution panel or screw terminal.

5.3.1 A/D Characteristics

The most important characteristics of an A/D converter are its conversion rate, voltage range, bit resolution, and quantization error. **Conversion rate** refers to how many conversions are performed in a unit of time. On a microcontroller, the conversion rate is dependent on the choice of the clock signal, its speed, as well as the bit range. A/D devices on a PC data acquisition card have conversion rates of less than 100 K conversions per second for low-end devices, while high-end devices have ranges in excess of one mega conversions per second. These numbers are the

Figure 5.1

Illustration of signal aliasing



upper limits on the performance of the A/D device by itself. When an A/D converter is used as part of a digital feedback control system, the effective conversion rate is lower and is highly dependent on processor speed.

The **voltage range** of an A/D converter refers to the analog voltage range that the device can handle. On a microcontroller, the range is 0 to V_{DD} (the supply voltage) unless an external reference voltage (V_{ref+} and/or V_{ref-}) is used, in which case the range is 0 to V_{ref+} or V_{ref-} to V_{ref+}. On a PC data acquisition card, most A/D devices allow both unipolar and bipolar ranges ranging from 0.05 to 10 V, but can tolerate an overload voltage of up to +/− 30 V. The voltage conversion range is normally set by a software call to the device controller. The **bit resolution** of the A/D device is quoted as the number of bits that the converted analog signal is mapped into. Common bit sizes are 12 to 16 bits, but many microcontrollers have A/D devices that have only a 10-bit range. The bit resolution affects the **voltage resolution** or increment of the device, which is defined as

$$\text{Voltage resolution} = \text{range}/2^n \tag{5.1}$$

where n is the bit resolution of the A/D convertor. To understand the relationship between voltage range and voltage resolution, let us consider a 12-bit A/D device with a −10 to 10 V range. In this case, a 20-V range (−10 to 10 V) is mapped

into 2^{12} binary combinations. Thus, the device can map voltage values to discrete values at increments of

$$\text{Range}/2^n = 20/4096 = 4.88\ \text{mV}$$

This means that the input voltage can increase by up to 4.88 mV without changing the output value of the A/D converter. For example, if the input voltage that was measured by this device was the output of an analog temperature sensor with a sensitivity of 10 degrees per volt (°/V), then the device will not be able to measure temperature changes that are less than 0.0488°. The discrete output of an A/D converter subjected to an input voltage V_{in} is given by

$$\text{Digital output} = \text{ceiling}((V_{\text{in}} - V_{\text{ref}-} - \text{voltage resolution})/\text{voltage resolution}) \quad \textbf{(5.2)}$$

Along with the voltage resolution of the device, the **quantization error** (or digitization accuracy) of the A/D converter is also an important performance parameter and is directly related to the bit range of the device. The digitization accuracy refers to the uncertainty in the discretized voltage value, and it is +/− one-half of a bit. To further understand the concept of digitization accuracy, let us consider an ideal 2-bit A/D device with a range of 0 to 5 V. This A/D device maps a 5 V analog range (0 to 5 V) into four different binary values (2^2 or 0, 1, 2, and 3). As seen in Figure 5.2, the output of the A/D converter will be 0 if the input analog voltage happens to be in the range of 0 to 1.25 V, will be 1 if the input analog voltage happens to be in the range of 1.25 to 2.5 V, and so forth. Notice that the maximum digital output level (3 in this case) is reached before the input reaches full scale or 5 V. Nominally, we say that if the A/D converter outputs a value of 1, then this corresponds to a nominal analog voltage of 1.875 V. The input voltage can change up to +/−0.625 V (or +/− one-half of a bit) without any change in the output of the A/D. Thus, at any of the discrete output values of the A/D converter, we say that we have an uncertainty of +/− half the voltage increment of the A/D device. Some A/D converters are built with an intentional offset of −1/2 bit. The staircase output curve of such a device will be shifted to the left and will start at 0.625 V for the 2-bit A/D example. The input/output relationship of an A/D converter is further illustrated in Example 5.1.

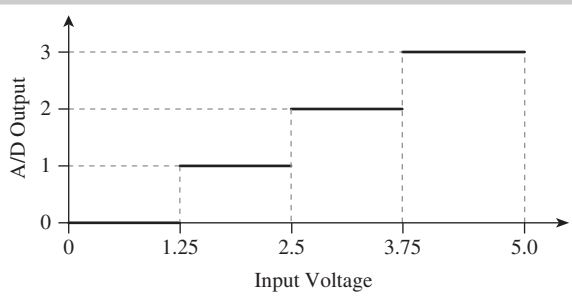


Figure 5.2
2-bit A/D mapping

Example 5.1 A/D Converter

Determine the voltage resolution and digitization accuracy of an ideal 12-bit A/D converter with a 0 to 10 V range. Determine the output level if the input voltage is 6.5 V. Also, determine the corresponding analog input voltage at the following digital output values: 0 and 1000.

Solution:

From Equation (5.1), the voltage resolution of this A/D device is $10/2^{12} = 2.441$ mV. Thus, the digitization accuracy of the conversion is +/− 1.220 mV.

From Equation (5.2), the discrete output level is given as the integer ceiling of ((6.5 V − 2.441 mV)/2.441 mV) or 2662.

This 12-bit A/D converter maps the 0 to 10 analog voltage range into 0 to 4095 digital values. When the A/D converter outputs a value of 0, the analog input voltage of this ideal A/D converter is in the range of 0 × 2.441 mV to 1 × 2.441 mV or 0 to 2.441 mV. Similarly, when the A/D converter outputs a value of 1000, the analog input voltage is in the range of 1000 × 2.441 mV to 1001 × 2.441 mV or 2.441 to 2.443 V.

5.3.2 A/D Operation

Most A/D converters are built to operate on the principle of successive approximation. In the **successive approximation method**, an internal D/A converter and a comparator circuit are used to converge on the digital signal that is closest to the sampled analog signal. Starting with the MSB, the bits in the D/A converter are set/reset one at a time until the sampled analog signal matches the output signal from the D/A converter to within the least significant bit. The binary pattern of the D/A converter is then the digital input signal. This conversion technique is a good compromise between speed, resolution, and cost. Example 5.2 illustrates the operation of a successive approximation 3-bit A/D converter with 0 to 10 V analog input range. Other types of A/D converters include flash/parallel, integrating, and digital ramp. To set up an A/D converter in a PC data acquisition board, the user has to make certain important selections, including the input range and the input signal configuration. The input range is normally set using software.

Example 5.2 Successive Approximation A/D

Illustrate the operation of a 3-bit, 0 to 10 V successive approximation A/D subjected to an analog input voltage of 8 V.

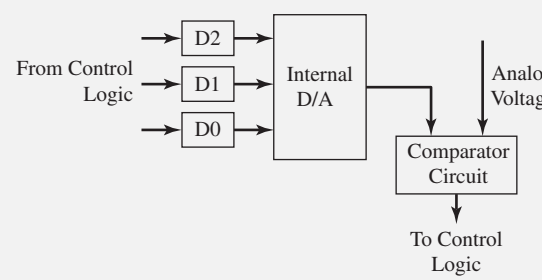


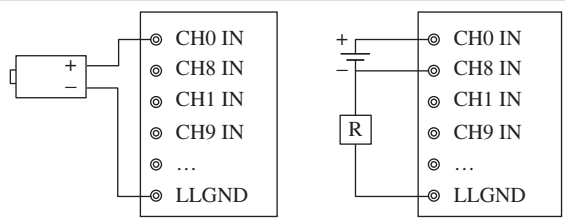
Figure 5.3

Solution:

With reference to Figure 5.3, the control logic of this A/D device will first turn the most significant bit (bit D2) of the internal D/A device associated with this A/D. The analog output voltage corresponding to this bit is 5 V (see table in Example 5.3). The comparator circuit will then compare the analog output of the internal D/A with the supplied analog input (8 V). Since the output of the internal D/A is smaller than the supplied voltage, bit D2 remains on, and the next bit (bit D1) is turned ON. The analog output (7.5 V) of the internal D/A is still less than the supplied voltage, so bit D1 remains ON. When bit D0 is turned ON, the analog output is 8.75 V, which is greater than the supplied voltage. Thus, bit D0 is turned OFF, and the output of the A/D will be 0x06, which is the same digital pattern on the internal D/A.

5.3.3 A/D Input Signal Configuration

The A/D input signal configuration refers to either single-ended input or differential input. In a **single-ended input mode**, the input signal is referenced to the A/D board's signal ground. The signal is connected using two wires, as shown in Figure 5.4(a), where the wire that carries the signal is connected to any one of the input channels terminals and the other wire is connected to the board's signal ground (called low-level ground or LLGND). In this configuration, the A/D converter measures the difference between the signal and the ground at the board. A single-ended connection is sensitive to noise, since the signal wire can act as an antenna picking up electrical noise. Note that the single-ended configuration should be used only with a floating signal source (i.e., one that does not have any connection to ground at the signal source), otherwise ground loops can be formed.



(a) Single-Ended Wiring (b) Differential Input Wiring

Figure 5.4

Data acquisition board wiring for single-ended and differential input mode

In **differential input mode**, the high (or positive) input signal is measured with respect to the low input (or negative) signal. The input signal is normally connected to the board using three wires. The wire that carries the high-input signal to be measured is connected to any of the 0, 1, 2, . . . , 7 A/D input channel terminals (assuming we have 16 single-input channels). The low signal is connected to an adjacent channel (such as channel 8 in Figure 5.4(b) if the signal was connected to channel 0). The low signal wire is also connected to the board's signal ground through a resistor. When the board is set for differential input mode, the number of available input channels is halved (i.e., a board that has 16 single-ended input channels will have eight differential mode channels).

The differential input mode is a better configuration for handling noise than the single-ended mode. Any electromagnetic interference induced in one lead of the signal is usually induced in the other lead. Since the A/D convertor in differential input mode measures the difference between the high and low ends, any voltage common to the high and low ends is removed in this mode. Differential input mode should be used to read the output of analog sensors (such as thermocouples and strain gages, see Chapter 7) which are susceptible to noise. A measure of the ability of an A/D converter used in differential input mode to eliminate the common voltage is called the **common mode rejection ratio (CMRR)**. In an ideal A/D converter, any voltage common to both signal wires will be completely cancelled. In a real A/D converter, a perfect cancellation does not occur, and a fraction of the common voltage will show. The CMRR, which is expressed in decibels (dB), is the reciprocal of the voltage fraction that is passed. It is desirable to have an A/D convertor with a high CMMR ratio.

Note that the PIC16F690 and the PIC18F4550 MCUs do not support differential mode A/D input, but Microchip manufactures special chips that support differential A/D input. An example is the MCP3301 chip, which is a dedicated 13-bit differential input A/D converter chip.

5.4 Digital-to-Analog Converter

5.4.1 D/A Characteristics

A D/A converter is a device that converts digital signals to analog signals. Most microcontrollers do not have a D/A converter, but the function of the D/A converter is approximated on microcontrollers using the PWM output feature (see Section 4.7.5). All of the performance parameters of an A/D (such as conversion rate, voltage range, bit resolution, and digitization accuracy) that were discussed before are similarly applied here for the D/A converter, so they will not be repeated, but we include an example that discusses the mapping between digital input values and analog output values.

Example 5.3 D/A Mapping

A 3-bit D/A converter is set for 0 to 10 V output range. Map all of the possible digital input values to their corresponding analog output values.

Solution:

A 3-bit D/A converter has 2^3 possible digital input values (0 to 7 decimal or 000 to 111 binary). The voltage resolution is $10/2^3 = 1.25$ V. The corresponding analog output values for each possible digital input value are shown here:

Binary Input	Analog Output (V)	Binary Input	Analog Output (V)
000	0.00	100	5.00
001	1.25	101	6.25
010	2.50	110	7.50
011	3.75	111	8.75

Note that while this D/A converter's nominal range is 0 to 10 V, the maximum output analog value is only 8.75 V due to the coarseness of its resolution. If the bit resolution was 10 instead of 3, then the maximum analog output voltage would be 9.990 V, or to generalize:

$$\text{Maximum output} = \text{range} - \text{resolution} = 10 - 10/2^{10} = 9.990 \text{ V}$$

The digital input that gives a certain analog output voltage V_{out} is given by

$$\text{Digital input} = \text{ceiling}((V_{\text{out}} - V_{\text{min}} - \text{voltage resolution})/\text{voltage resolution}) \quad (5.3)$$

where V_{min} is the minimum voltage supplied by the D/A. Using the data from Example 5.3, the digital input needs to be 6 for an output of 8 V. Due to the coarse resolution of this A/D converter, the analog output will only be 7.5 V (6×1.25). If this was a 10-bit D/A converter instead, then the output would be 7.998 V at a digital input of 819. Most commercial D/A converters have a 12-bit output resolution.

5.4.2 D/A Operation

To illustrate the operation of a D/A converter, let us consider the **weighted resistor summing amplifier circuit** shown in Figure 5.5. The digital input acts as an electronic switch in this circuit, providing a connection between V_R and the respective resistor if the binary value is 1. For the digital input 1011 shown in the figure, the output of this circuit is $-11\ V_R$. This circuit is not used in practice, because it

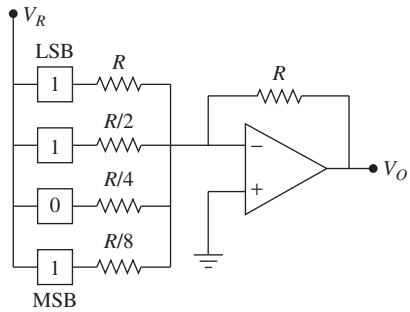


Figure 5.5

Weighted resistor summing amplifier circuit

requires resistances of certain ratios which are difficult to satisfy with good accuracy. This is especially true for a D/A convertor with more than 4 bits. For example, a 12-bit D/A would require that the 11th bit resistor have a resistance of 1/2048 of the 0th bit resistor.

A commonly used circuit for D/A conversion is the ***R/2R* ladder resistor network** [15]. Unlike the weighted resistor summing amplifier circuit, the *R*/2*R* ladder circuit requires only two resistor values *R* and 2*R* regardless of the number of bits used. The *R*/2*R* ladder circuit is shown in Figure 5.6. It consists of a repeating pattern of 2*R* and *R* resistors arranged in a ladder form. The 2*R* termination resistor is connected to ground and is used to make the Thevenin resistance of the network at each ladder leg equal to *R* when all the bits are grounded (see Figure 5.6).

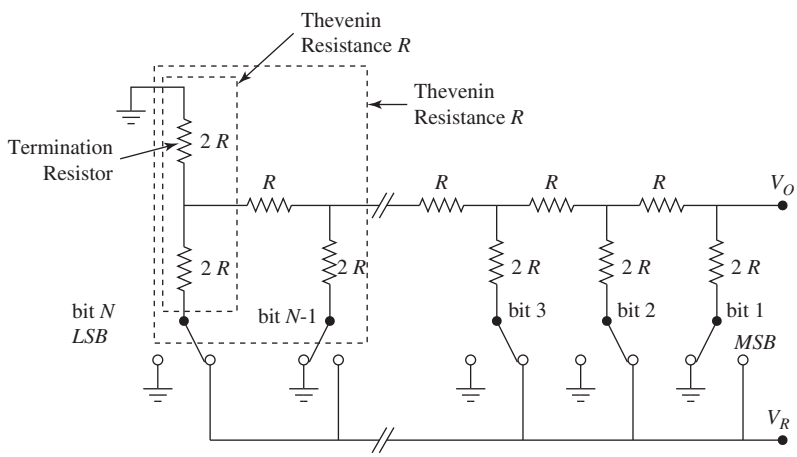


Figure 5.6

R/2*R* ladder resistor network

The output voltage V_O when a voltage V_R is connected to a bit i ($1 \leq i \leq N$) with all the other bits grounded is given as

$$V_0 = \frac{V_R}{2^i} \tag{5.4}$$

If more than one bit is connected to V_R, then the principle of superposition applies. Thus for example, if we have a 3-bit D/A with all the bits connected to V_R, then the analog output is equal to $V_R/2 + V_R/4 + V_R/8 = 7/8\ V_R$. If V_R is equal to 10 V, then the output is 8.75 V, as was the case in Example 5.3. The *R*/2*R* network provides the most accurate method of digital-to-analog conversion.

A/D and D/A converters on data acquisition boards have different data transfer modes. These include **direct memory access** (or DMA) and programmed I/O.

In DMA, the data is transferred directly between the memory and the data acquisition board without using the system processor to perform the operation. In **programmed I/O**, the system processor directly controls the transfer of data between the memory and the board. While DMA allows fast data transfer rates (300 KHz or higher), extra set-up is needed for DMA operation, which makes it advantageous only in transferring large amounts of data.

Practical D/A converters employ a **zero-order hold circuit**. This means that the current output of the D/A device will remain constant until a new output is sent to the device. In using a D/A converter in a sampled feedback control system, the output will remain constant between updates.

5.5 Parallel Port

Many data acquisition cards (DACs), as well as microcontroller chips, allow digital I/O signals to be sent back and forth through what is called a **parallel port**. The name parallel port comes from the fact that all the data that is presented to the device is transmitted simultaneously, thus the name parallel. The printer port, which is available in some old PCs, is an example of a parallel port. Common configuration of a parallel port on PC data acquisition cards is four 8-bit ports, such as those that are available with the Measurement Computing DAC. Each port can be configured through software to be an input type or an output type, or even a combination of the two. Most parallel ports are constructed using transistor-transistor logic (TTL) family chips, which have different voltage ranges for input and output. These ranges are shown in Table 5.1.

Table 5.1
TTL input and output levels

Operation	Low State Voltage Range	High State Voltage Range
Input	0–0.8 V	2.0–5.0 V
Output	0–0.5 V	2.7–5.0 V

The input parallel port is commonly used to read data from switches and on/off type sensors, such as proximity sensors and limit switches. The output parallel port is normally used to activate lights, solenoids, and relays. While the software that comes with most data acquisition boards provides functions to access the parallel port, it does not provide means to read a single bit on the card or set a particular output bit without disturbing the rest of the bits. These operations can be done by using the bit-wise logical operators that are available with VBE or C. Example 5.4 illustrates this using VBE syntax.

Example 5.4 Parallel Port

Illustrate how bit #5 of an 8-bit parallel port can be set either high or low without changing the current output on the port.

Solution:
To enable us to perform this operation, we need to have a variable that stores whatever was sent to the port. Let us call this variable *PortValue*. To set bit #5 to high, we simply perform a bitwise OR operation between the variable *PortValue* and the value &H20. We can write it as

PortValue = PortValue OR &H20

Notice that in using this operation, the value of each bit, other than bit #5, is not changed from its current value, since 'oring' a bit with 0 does not change its value. To set bit #5 to low, we need to perform a bitwise AND operation with the value *&HDF*, which has a value of 1 in all its bits except bit #5, which has a value of 0.

PortValue = *PortValue AND &HDF*

The updated value of *PortValue* is then sent to the port.

5.6 Data-Acquisition Board Programming

To illustrate programming issues in using a data-acquisition card, we will discuss a data-acquisition card made by Measurement Computing of Norton, MA. The particular card is the **PCIM-DAS1602/16** which has sixteen 16-bit A/D channels, two 12-bit D/A channels, thirty-two digital input/output lines, and three 16-bit counters. The card is installed on one of the available slots on the PC, and using a ribbon cable, the interface pins are brought to a screw terminal where the user can conveniently wire up signals to the card. (See Figure 5.7).

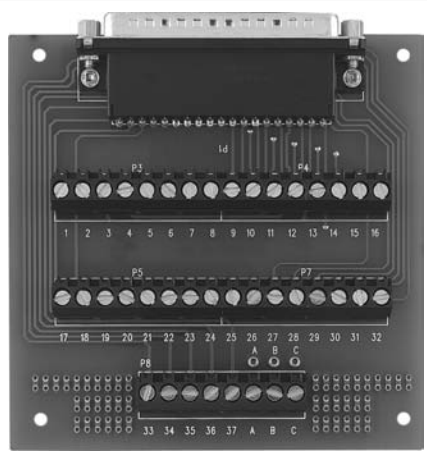


Figure 5.7

Screw terminal for a data acquisition card
(Courtesy of Measurement Computing, Norton, MA)

The user has to develop a set of interface functions to use the card. An example of such interface functions (in Visual Basic Express) are provided here (the complete code is provided on the text website).

Sub ***Write_DA****(ByVal Chan%, ByVal DataVolts!)*—sends a voltage signal (−10 to 10 V) to D/A channel 0 or 1

Function ***Read_AD****(ByVal Chan%) as double*—reads a voltage signal (−10 to 10 V) from channel 0, 1, . . . , 15

Function ***Read_IO****() as UShort*—reads a byte from the parallel port B

Sub ***Send_IO****(ByVal DataValue%)*—sends out a byte to parallel port A

Note that in the above A/D and D/A interface functions the range is set for −10 V to 10 V, but the range can be conveniently changed in software by accessing the *Range* enumeration, which has a wide variety of bipolar and unipolar (such as

0 to 1 V) range settings. This card, similar to many other cards from other vendors, has a built-in function (called *DaqBoard.ToEngUnits*) to convert the read digital A/D value (raw data) to engineering units (volts). This relieves the user from doing the conversion.

The 32 digital input/output lines on the card are split into four 8-bit ports. These ports can be set in software to operate as digital input, digital output, or a combination of the two. This gives the user flexibility in configuring the port.

5.7 USART Serial Port

A parallel port is appropriate to use when the data is transmitted over a short distance. When data needs to be transmitted over longer distances, a **serial port** is more suited to use, as it is more immune to noise. Furthermore, a serial connection normally requires fewer wires than a parallel connection. In addition, a serial port has the means to set up communications in asynchronous fashion, which is not the case with parallel ports. A serial port is an input/output device that takes data in a parallel form and transmits it in a serial fashion. Terminals, modems, mice, and keyboards are examples of devices that connect to a PC or an MCU through a serial port. In a serial port, the data is transmitted one bit at a time, rather than simultaneously. The data to be transmitted is broken up into packets, each packet is made up of a number of bits, and then the bits in these packets are transmitted sequentially, thus the name serial port. The speed of transmission or baud rate refers to the number of bits per second that can be transmitted. The baud rate can range from 300 to several hundred thousand.

Serial data can be transmitted in an asynchronous or in a synchronous fashion. **Asynchronous** transmission is used when the transmitter and the receiver operate independently, each using its own clock signal, while **synchronous** transmission is used when the transmitter and receiver have a common clock signal. Asynchronous transmission is the default on PC's, since the required hardware for synchronous transmission is normally not present on PC's.

In asynchronous transmission, means must be provided to inform the receiver of the start and end of a data packet, since the timing of the transmission of the packet is not known in advance. This is accomplished by structuring the **data packet** to include a **start bit** at the beginning of the packet to inform the receiver of the start of the packet and a **stop bit** at the end of the packet to indicate its completion. The 10-bit serial data packet is thus structured to include one start bit (the first bit that is transmitted), 7 or 8 bits of data representing the character to be transmitted, an optional **parity bit** for 7-bit data, and one stop bit (or two stop bits for 7-bit data with no parity bit) at the end of the packet. The start bit is always low (space) while the stop bit is always high (mark). Figure 5.8 shows the packet structure for one start bit, seven data bits, one parity bit, and one stop bit.

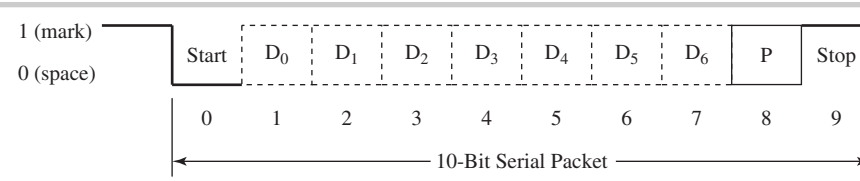


Figure 5.8
Serial packet structure

Since the receiving end does not know ahead what data is transmitted, the parity bit can be used to provide a crude method of error checking. The different **parity methods** are listed here.

Even: Means that the total number of one bits in the packet (excluding stop bit) is even. Thus, the value of parity bit is set to 1 or 0 to make the total number of one bits even.

Odd: Means that the total number of one bits in the packet (excluding stop bit) is odd. Thus, the value of parity bit is set to 1 or 0 to make the total number of one bits odd.

Mark: Parity bit is always set to logical 1 (mark).

Space: Parity bit is always set to logical 0 (space).

None: No parity bit is sent at all.

For even/odd parity, the parity method of error checking works by counting the total number of one bits in the packet (excluding the stop bit) that is received. If even (odd) parity was selected, then a transmission error has occurred if that number is not even (odd). For mark or space parity, the parity bit in the received packet is checked to see if it matches the parity set mode. The parity method can detect a single-bit error but is not guaranteed to detect multibit errors. The seven or eight bits of data are converted using the ASCII code (See Appendix D for a list of the codes). Example 5.5 illustrates the serial packet structure.

Example 5.5 Serial Packet Structure

Show the 10-bit serial packet for the letter 'B' using a 7-bit data, one start bit, one stop bit and a parity bit. Illustrate for both even and odd parity.

Solution:

The 7-bit ASCII code for the letter 'B' is 0x42 or b100 0010 (see Appendix D). Thus, for even parity, the 10-bit serial packet is

```
0 + 0100001 + 0 + 1
Start + 7-bit ASCII + parity + stop bit
```

or '0010000101'.

For odd parity, the 10-bit serial packet is

```
0 + 0100001 + 1 + 1
Start + 7-bit ASCII + parity + stop bit
```

or '0010000111'. Note that the start of the packet is from the left end.

Serial ports installed on PCs and laptops take the form of a 9-pin male 'D' connector (older PCs have a 25-pin male connector but only nine pins of these were actually used). In a serial connection, the data can be sent in full-duplex mode or half-duplex mode. **Full-duplex mode** means that the data between the two devices in communication can be transmitted simultaneously in both directions. In **half-duplex mode** (which is outdated and not commonly used), data is transmitted in one direction at a time, but the direction can be changed. For full-duplex mode, a serial connection between two ports requires (physically) a minimum of three wires if no hardware flow control is used. One wire is used for sending the data, the

second wire is used for receiving the data, and the third is used as ground. The remaining pins in the connector are used for control purposes.

There are several protocols for serial interface. The most common is the RS-232C protocol (now called EIA-232). **RS-232** stands for recommend standard number 232, and C is the latest revision of the standard. A subset of the RS-232C standard is used in the serial ports on most computers. The RS-232 protocol has limits on speed, cable distance, and device support. Another protocol is the RS-422, which permits longer cable distances at higher cost, since each signal is carried by two wires due to the use of differential mode in signal transmission.

To prevent overflow of the buffer that receives the data, handshaking or **flow control methods** can be used. Software-based flow control, hardware-based flow control, or a combination of the two can be used. The XOnXOff is a software flow control method in which these characters are sent from the receiver to the transmitter to control when data can be sent. XOnXOff stands for transmit on/transmit off. If the receiver is ready to accept characters, it will send an XON character (typically 0x11) to the transmitter. If the receiver buffer is full, it will transmit an XOFF (typically 0x13) character to the transmitter to stop the transmission of data. Note that this method of handshaking is software based, and thus can be used with a three-wire serial cable.

An alternative method of handshaking is the use of the *Request-to-Send* (RTS) and *Clear-to-Send* (CTS) signals, but it is hardware based. If the input buffer is not full, the RTS line will be set to true indicating that the receiver can accept characters. If the input buffer becomes full, the RTS line will be set to false. Both the RTS hardware control and the XON/XOFF software controls can also be used at the same time.

The common method of using a USART on a PIC MCU is to implement full-duplex asynchronous serial communication using the RS-232 protocol. The USART has a two-character input buffer and a single character output buffer on the PIC16F690. It allows 8-bit or 9-bit character length and has means to detect input buffer overrun errors and received character framing errors. A **framing error** is defined as a serial packet that is not in the expected format (such as having a wrong number of bits). With the use of a 9-bit character length, the serial packet becomes 11 bits in length (with the addition of the start and stop bits). The USART allows a range of **baud rate settings** that are depended on the oscillator clock frequency. The SPBRG register controls the period of the baud rate generator (BRG) which is implemented as a free-running, 8-bit timer. Actually, the PIC MCU allows two ranges of baud rates: low and high. The high range is obtained when the BRGH bit of the TXSTA register is set to 1, and the low range is obtained when that bit is set to 0. The formula for the desired baud rate for asynchronous transmission is

$$\text{Desired baud rate (low speed)} = F_{OSC}/(64(X + 1)) \quad (\text{BRGH} = 0) \tag{5.5a}$$

$$\text{Desired baud rate (high speed)} = F_{OSC}/(16(X + 1)) \quad (\text{BRGH} = 1) \tag{5.5b}$$

where F_{OSC} is the oscillator frequency in Hz and X (0 to 255) is the value written into the SPBRG register. Note that because X is limited to only 256 values, the actual baud rate may be different from the desired baud rate. As an illustration, consider an 8 MHz oscillator, and a desired baud rate of 9600. With BRGH = 0, we can solve the top equation for the value of X to give this desired baud rate. Doing this, we get X to be 12.02. Since X is limited to integer values, using a value of $X = 12$ gives us an actual baud rate of 9615, which is 0.16% higher than the desired baud rate. When the actual baud rate is different from the desired baud rate by more than 1 to 2%, transmission errors such as missing or wrongly received bits

could occur. Microchip provides data sheets with the appropriate value of *X* to use to obtain a particular baud rate. When using a high-level compiler (such as C or Basic), the user does not need to write to these registers to set up the baud rate. The compiler provides functions for this purpose. On the other hand, when using assembly language, the user has to explicitly write to these registers.

Figure 5.9 shows a code listing for serial communication on a PIC MCU using the PIC-C compiler. The example code has an infinite loop which continuously reads the serial port. If the read character matches a specified character, the character '1' is transmitted.

Figure 5.9

PIC-C code for serial communication

```
////////////////////////////////////////////////////////////////////////////
///                    Serial_In_Out.c
///
///     Program that demonstrates RS-232 communication
///     Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////
#include <18F8722.h>
#fuses HS, NOMCLR, NOWDT, NOPROTECT, NOBROWNOUT
#use delay (clock=10000000)
#use rs232(baud=38400, BITS = 8, PARITY = N,ERRORS, xmit=PIN_C6, rcv=PIN_C7)

char ReadSer();                 // Function prototyping in C
void WriteSer(char);            // Function prototyping in C

void main()                     // Main routine
{
char c;
while ( 2 > 1)                  // Start infinite loop
  {
      c = ReadSer();            // Read the serial port
      if (c == 'a')             // Is the read character 'a'?
        {
        WriteSer('1');          // If the read char is 'a', then send the char '1'
        }
    }
}
                                // ReadSer function
char ReadSer()                  // Read a character from the serial port
{
  if (kbhit() == 1)             // Is there a character in the buffer
    return(getc());             // Return character if available
  else
    return('0');
}
                                // WriteSer function
void WriteSer(char d)           // Send a char to the serial port
{
  printf("%c\n\r",d);           // Send a new line and carriage return
}
```

VBE includes a control for handling serial communication. The control name is *SerialPort*, and it is of the indirect type. When this control is added to a form, it does not show in the form when the program is executed.

For sending out data, VBE has several functions, including the following.

Write()—sends a string to port

WriteLine()—sends a string and a new line char to port

For receiving data, VBE has several functions, including the following.

Read()—reads a specified number of bytes

ReadExisting()—reads all available characters in the input buffer

Figure 5.10 gives VBE code listing for a serial port that is configured for eight data bits, 19200 baud rate, one stop bit, no parity, and no handshaking. It also includes code for transmitting a string using the *WriteLine* function. The code listing makes use of the 'With' keyword which can used to access multiple elements of an object using the 'dot' operator without the need to repeat the name of the object. This code writes to the 'com3' serial port. This port can be a dedicated serial port or a USB port (see next section) that is configured by the operating system (through the Windows Device Manager) to operate as a serial port.

Figure 5.10

VBE code listing for serial port setup and communication

```
'VBE 2010 code for transmitting a string data using serial port
Private Sub serial_send(ByVal data As String)

    With SerialPort1
        .PortName = "com3"
        .DataBits = 8
        .BaudRate = 19200
        .Handshake = IO.Ports.Handshake.None
        .StopBits = IO.Ports.StopBits.One
        .Parity = IO.Ports.Parity.None
        .Open()
        .WriteLine(data + vbCr)
        .Close()
    End With
End Sub
```

5.8 Serial Peripheral Interface

To provide higher communication speeds, many PIC chips have a built-in synchronous serial port (SSP) module. Both the PIC16F690 and the PIC18F4550 support the two modes of operation of the SSP: the Serial Peripheral Interface (SPI) and the Inter-Integrated Circuit (I^2C™) interface (described in the next section). The **SPI** operates in full duplex and at speeds of 1 Mbps or higher. It is simple to implement (needs only four wires) and uses the concept of master/slave. These four wires are given here.

Clock Signal (SCK pin): This is the clock pulse signal that the master sends to the slave. One bit of data is transmitted for each clock pulse.

Master Out Slave In (SDO pin): Output data from master to slave.

Slave Out Master In (SDI pin): Output data from slave to master.

Slave Select (SS pin): This signal is used to select the particular slave in the case of one master and several slaves.

Each end of an SPI consists of a buffer and a shift register. A **shift register** is a grouping of flip-flops connected in a chain in which a binary number can be

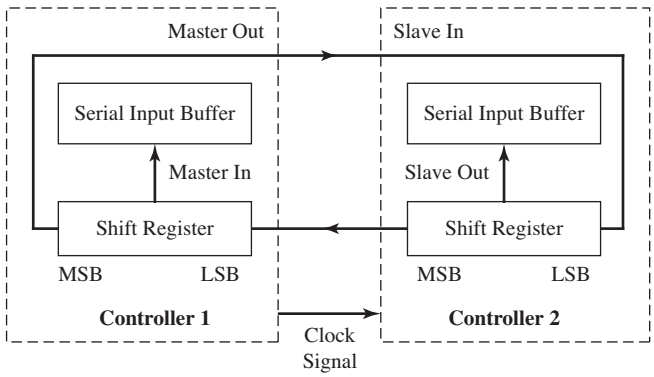


Figure 5.11

Illustration of SPI Interface

stored. The number can be shifted to the right or the left when a shift pulse is applied. Data is read/written to the SPI through reading and writing to the buffer register on each end of the SPI. Data is then copied to the shift register and transmitted by shifting out one bit at a time on each clock pulse. To illustrate the operation of an SPI, consider Figure 5.11. It shows the connections between two SPI ends. On each clock pulse, one bit is shifted out from the shift register on either end on the MSB side. The shifting out of the MSB on the master shift register allows the MSB from the slave shift register to be placed in the LSB bit location on the master side. After eight clock pulses, the contents of both shift registers are exchanged. At this instant, the contents of the shift register are copied to the buffer register if the SPI was performing a read operation, or the contents of the buffer register are copied to the shift register for next transmission if the SPI was performing a write operation.

In creating an SPI interface between two devices, one device should be designated as the master and the other as the slave. The master sends out the clock pulses, while the slave does not send out the clock pulses but receives them from the master. The clock rate is set in the master. In PIC MCUs, the SPI clock rate can be set to $F_{OSC}/4$, $F_{OSC}/16$, $F_{OSC}/64$, or Timer2-output/2, and the maximum allowed data rate is typically less than 10.0 Mbps. In addition to specifying the clock rate, the user also needs to specify whether the output data is sent on the rising or the falling edge of the clock signal.

The PIC-C compiler has built in-functions to set and access the SPI interface. They include the ***setup_spi*()** function to configure the SPI device (i.e., master or slave, clock rate, and clock edge), and the ***spi_read*()** and ***spri_write*()** functions to read and write to the interface, respectively. Calling the *spi_write*() function causes the clock pulses to be generated and a byte to be sent to the SPI interface. As the data is being sent, the incoming data is clocked in and stored in the buffer. The *spi_read*() function can be called with or without a data argument passed to it. If called without an argument, the function will read the data received from a previous *spi_write*() operation or wait for the data if no data is ready. Calling the *spi_read*() function with an argument causes the data to be clocked out and the incoming data to be received.

Figure 5.12 shows a PIC-C code listing for reading and writing to an EEPROM (Microchip 25LC256 EEPROM) using the SPI interface. The particular sequence of commands in the *write_ext_eeprom()* and *read_ext_eeprom()* functions correspond to the details of accessing this EEPROM as defined in the manufacturer data sheet.

Figure 5.12

PIC-C code listing for reading and writing to an EEPROM using the SPI interface

```
////////////////////////////////////////////////////////////////////////////////
///    A collection of routines for SPI communication with the 25LC256 EEPROM
///
///    Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////
#define EEPROM_SELECT PIN_A3
#define EEPROM_ADDRESS long int
#define READ        0x03
#define WRITE       0x02
#define WREN        0x06

void init_ext_eeprom()                      // Initialize EEPROM function - called once
 {
  output_high(EEPROM_SELECT);               // Make EEPROM_SELECT line high
  setup_spi(SPI_MASTER |SPI_XMIT_L_TO_H| SPI_CLK_DIV_16 ); // Set SPI mode
}

void write_ext_eeprom(EEPROM_ADDRESS address, BYTE data)
{
  output_low(EEPROM_SELECT);                // Make EEPROM_SELECT line low
  spi_write(WREN);                          // Send code to enable writing
  output_high(EEPROM_SELECT);               // Make EEPROM_SELECT line high

  output_low(EEPROM_SELECT);                // Make EEPROM_SELECT line low
  spi_write(WRITE);                         // Send code to start writing
  spi_write(address>>8);                    // Send MSB byte first
  spi_write(address);                       // Send LSB byte
  spi_write(data);                          // Send data
  output_high(EEPROM_SELECT);               // Make EEPROM_SELECT line high
  delay_ms(6);                              // Delay to complete the erase and writing of data
}

BYTE read_ext_eeprom(EEPROM_ADDRESS address)
{
  BYTE data;
  output_low(EEPROM_SELECT);                // Make EEPROM_SELECT line low
  spi_write(READ);                          // Send code to start reading
  spi_write(address>>8);                    // Send MSB byte first
  spi_write(address);                       // Send LSB byte
  data=spi_read(0);                         // Read data
  output_high(EEPROM_SELECT);               // Make EEPROM_SELECT line high
  return(data);                             // Return data to calling function
}
```

5.9 Inter-Integrated Circuit Interface

The I^2C interface (pronounced I-Squared-C) is a synchronous serial communication protocol that was developed by Philips Semiconductor. The I^2C or the inter-integrated circuit (I^2C™) interface uses just two wires—one for data transmission and the other for the clock signal—for the interface between two devices. On the PIC16F690, the data line is the SDA pin (pin 13), and the clock line is the SCL pin (pin 11). Figure 5.13 shows the wiring between an I^2C master (i.e., a PIC MCU) and an I^2C slave (such as a serial EEPROM). Note that the SDA and SCL lines are open-collector types, and a pull-up resistor is needed on each line.

While there are several modes of I^2C interface, the most common one is that of a **single master and a single slave**. In this mode, the master controls the communication between the two I^2C devices. The master starts the communication by sending a start bit followed by the slave address and a bit indicating whether it wants to perform a write or read operation. If the sent address matches the internal address of the slave, then the slave will send back an acknowledgment bit to the

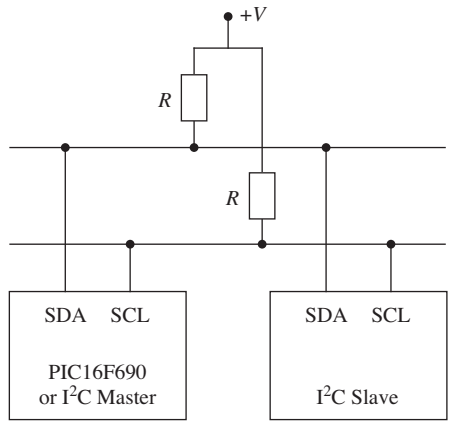


Figure 5.13

I²C wiring

master. Upon receiving the acknowledgment bit, the master will send out a data byte to the slave if it was performing a write operation, or will read a data byte from the slave if it was performing a read operation. The master terminates the communication by sending a stop bit to the slave. Note that after each data write (read) operation, the slave (master) will send an acknowledgement bit to the master (slave). The I²C standard interface speed is 100 kHz, but higher speeds (400 kHz and 1 MHz) can be used with the understanding that the PIC MCU I²C interface does not conform to high-speed (above 100 kHz) I²C specification in all details. Note that the I²C interface speed is less than the maximum SPI interface speed.

The PIC-C compiler has several functions for accessing the I²C interface. They include *i2c_start*() to issue a start condition, *i2c_stop*() to issue a stop condition, *i2c_poll*() to check if the hardware has received a byte in the buffer, *i2c_read*() to read a byte from the I²C interface, and *i2c_write*() to write a byte to the I²C interface. Figure 5.14 shows the code listing in PIC-C language that uses the

Figure 5.14

PIC-C code listing for I²C interface functions

```
////////////////////////////////////////////////////////////////////////////////////////////
///     Code for accessing an I2C RAM chip
///
///     Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////////////////
void write_ext_ram_byte(long int address, byte data)
{
   i2c_start();                        //Issue a start condition
   i2c_write(0xa0);                    //Set mode for data transfer from master to slave
   i2c_write((byte) (address>>8));     //Address MSb
   i2c_write(address);                 //Address LSb
   i2c_write(data);                    //Send out data
   i2c_stop();                         //Issue a stop condition
}

byte read_ext_ram_byte(long int address)
{
  byte data;
   i2c_start();                        //Issue a start condition
   i2c_write(0xa0);                    //Set mode for data transfer from master to slave
   i2c_write((byte) (address>>8));     //Address MSb
   i2c_write(address);                 //Address LSb
   i2c_start();                        //Issue a start condition
   i2c_write(0xa1);                    //Set mode for data transfer from slave to master
   data=i2c_read(0);                   //Read data without sending an acknowledgement
   i2c_stop();                         //Issue a stop condition
   return(data);                       //Return data from function
}
```

I^2C interface to communicate with an external RAM chip (RAMTRON FM24C256). The figure shows two routines: one for writing a byte and the other for reading a byte. Note that in the *read_ext_ram_byte()* routine, all of the code above the second *i2c_start()* statement is used to go to the specified address on the chip, but instead of performing a write operation, it is followed by code to perform a read operation.

5.10 USB Communication

A USB port or Universal Serial Bus is an external bus interface that is available on almost all PCs and laptops that were made in the last few years. It is designed to connect to external devices that connect to a PC (such as external hard drives, mice, scanners, printers, digital cameras, and DVDs). The term 'Universal' is used since the port can communicate with many types of devices. The 'serial' term refers to the flow of information on the bus. A USB is built with a 'bus' architecture, which provides an organized method to move information from many devices into and out of a computer system. One advantage of a USB port is that it allows a device to be connected or disconnected to a computer without powering down or rebooting the computer.

The 'Universal' aspect of the USB port stems from the requirements placed on developers of USB devices by the USB Implementers Forum (USB-IF). Devices that plug into a USB port are classified into one of several classes. Examples of **device classes** include the USB human interface device (HID) class, which includes USB mice and keyboards, and the mass storage device class, which includes USB flash drives. When a standard USB device that fits into one of the defined classes (such as a USB mouse) is plugged into a PC, the operating system should be able to automatically identify the device. In addition, a Windows application can communicate with that device using a driver that is supposed to be provided by the operating system without loading any additional software. For cases where the device does not belong to one of the device classes specified by USB-IF, a custom driver needs to be developed to access the USB.

USB communication is quite an involved process, and this section gives only a brief outline of the process of USB communication. For further information on this topic, the reader is encouraged to read books with details on this topic, such as [16].

5.10.1 USB Standards and Terminology

There are several standards for USB communication; the most recent is the USB 3.0 specification, which supports speeds up to 5 Gbps or SuperSpeed. This standard is not yet widely available in PCs and devices, so we restrict most of our discussion to the USB 2.0 standard. This standard supports data transmission rates of up to 480 Mbps, which is forty times faster than the rate allowed by the previous USB 1.1 standard. The USB 2.0 standard supports three speeds: high speed (480 Mbps), full speed (12 Mbps), and low speed (1.5 Mbps).

In USB terminology, the PC is known as the **host** that communicates with devices that are attached to the host. The host has a *host controller* which formats the data that is transmitted on the bus and also manages the communication on the bus. The host also has a *root hub* which has one or more connectors for attaching devices. All USB communication is between a host and a device (exception in USB 3.0), and direct communication is not allowed between hosts or between devices. Up to 127 devices or hubs can connect to a single host controller at one time, either through the installed USB ports on the PC or through external hubs, which can

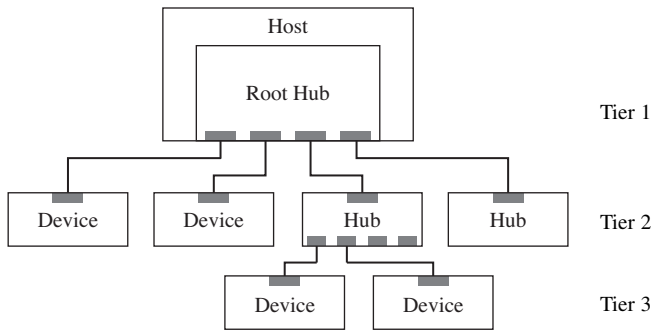


Figure 5.15

Physical connection structure with USB communication

connect between two to seven devices to one port on the PC. Modern PCs have several host controllers, each controlling an independent bus, to improve the communication bandwidth. Figure 5.15 shows the physical connection structure between a host and hubs or devices. This structure is referred to as a tiered star structure. Note that the physical connection between a host and a device (i.e., through more than one hub or not) does not affect the programming between host and device.

A ***device*** is a physical or logical unit that performs a particular function. Devices include hubs and physical devices (such as printers and keyboards). The host assigns a unique address to each device on the bus. This is needed because multiple devices can share the data path on the bus. Every device that supports USB communication has a controller chip to manage the communication. Also, every USB device has identifiers (*Vendor ID* and *Product ID*) that identify the device to the operating system.

A USB cable has four shielded wires. Two of these wires are used for power (+5 volts and ground), and the remaining two carry the data. In the USB 2.0 standard, the bus carries the data in one direction and at one speed at a given time. The cable end can be either of the A or B forms (see Figure 5.16). The A form is designed to connect to the computer side, while the B form is used to connect to the device side. Cable length is limited to about 4 m. Smaller size USB plugs and receptacles called mini-B and micro-USB are also available.

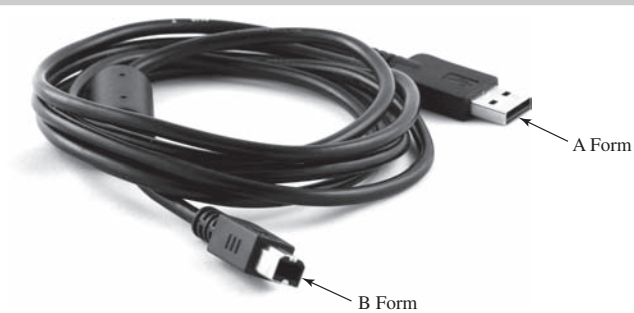


Figure 5.16

A and B forms of USB connector

(© Nenov Brothers/ Shutterstock.com)

One nice feature of the USB interface is that a device can draw its power from the USB bus instead of using a dedicated power supply. The current limit is 500 mA for USB 2.0 and 900 mA for USB 3.0.

The operating system prevents applications from directly accessing the USB hardware. Applications need to access the USB hardware through a driver that the operating system assigns to the device connected to a particular port. The driver in turn communicates with lower-level drivers that manage communication on the

bus. Developing a driver for a USB device is not a trivial task, as it requires detailed knowledge about USB protocols and buses. VBE (or even VB.NET) does not provide built-in code to directly access a USB port. While a USB connector on a PC is called a port, it differs from other ports on a PC (such as a serial port). The primary difference is that a USB port is part of a bus system, while a serial port is an I/O port with a specific location address. Note that for PCs that do not have a serial port, the USB can be used as a standard RS-232 port. This is accomplished by software/hardware that makes the USB device look to the host software like a COM (RS-232) port. Configuration of this port is done using the Windows Device Manager.

When a device first attaches itself to a USB port, the PC searches for an INF (information) file, which specifies the driver that the PC will use with the device and will load. On every subsequent attachment of the device to a host, the host undergoes a process of **enumeration**. In it, the host requests information from the device so it can assign the appropriate driver to use when communicating with the device. In the enumeration process, the host requests information from the device, assigns an address for the device, and selects a configuration that reflects the device's power and interface requirements.

5.10.2 USB Data Transfer

In USB communication, data is transmitted between the host and devices based on requests from the host. Data communication between the host and the device is done through logical channels or **pipes** which connect the host controller's software to entities called endpoints on the device. An ***endpoint*** is a buffer that holds transmitted or received data. Physically, an endpoint is a block of memory or a register. While the host has buffers to hold transmitted and received data, they are not called endpoints. Each endpoint is defined by an address that has two components: an endpoint number in the range of 0 to 15 and a direction labeled either *IN* or *OUT*. The direction is from the perspective of the host. Thus, the IN endpoint has data to transmit to the host, while the OUT endpoint has data that was received from the host. The number of endpoints for a device is dependent on the USB speed mode with full and high-speed modes having up to 32 endpoints. Every device must have an endpoint zero, which is used for control purposes. The communication pipes are established by the host during the enumeration process. They can be removed by the host if the device is no longer attached to the host, or changed if the host asked for a new interface to the device. Figure 5.17 shows a schematic of the logical interface between a host and a device.

Data transfer between the host and an endpoint is performed using one or more transactions. There are three kinds of **transactions**: *Setup*, *In*, and *Out*. A *Setup* transaction sends control information from host to endpoint. An *In* transaction sends data from the device to the host, while an *Out* transaction sends data from the

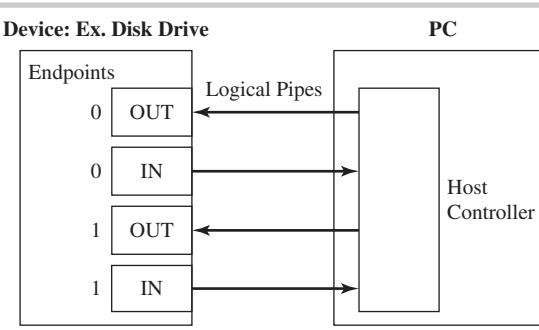


Figure 5.17

Illustration of logical channels in a USB connection between a host and a device

host to the device. A transaction consists of a series of packets, where a **packet** is a block of information with a defined structure. There are different types of packets, including *token*, *data*, and *handshake* packets. A **token** packet identifies the transaction type such as *In* or *Setup*. A **data** packet carries data or status information, while a **handshake** packet carries status code. Each transaction has a token packet that is always sent by the host and may also include a data and/or a handshake packet. Figure 5.18 shows the relationship between transfers, transactions, and packets.

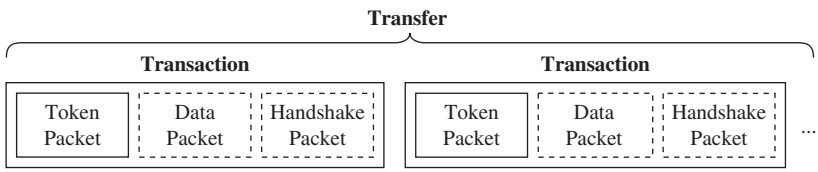


Figure 5.18

Illustration of a USB transfer

All packets begin with a **packet ID (PID)** that contains information that identifies the packet. For example, the PID name for a token packet is either Out, In, Setup, or SOF (start of frame). The SOF is used for timing purposes. A handshake packet PID name could be ACK (which means that the receiver accepts error-free data packets). The remaining entries in the packet are dependent on the packet type and may include the endpoint address, data, status information, or errors-checking bits (CRC). Figure 5.19 shows the packet format for token, data, and handshake packets. Note that there are other packet types (such as the *PRE* packet) that are not discussed here. When an endpoint receives a packet from the host, it uses the packet ID to determine what to do. For example, in receiving an *Out* token packet, the endpoint stores the data that follow in the data packet. The device hardware usually triggers an interrupt after the data is received, and software is then used to process the received data. The size of a packet varies with USB bus speed.

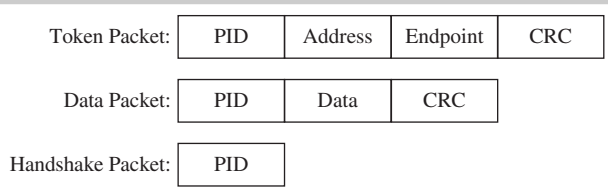


Figure 5.19

USB packet format

The scheduling of data transfer on the USB bus is handled by the host. Time is divided into 1-ms frames at low and full speed and into 125-μs microframes at high speed. A portion of the frame or microframe is allocated for each transfer. The transactions for a particular data transfer can be split over several frames or microframes, but each USB 2.0 transaction is completed within a frame or a microframe without interruption. A schematic of the timing of data transfers is shown in Figure 5.20.

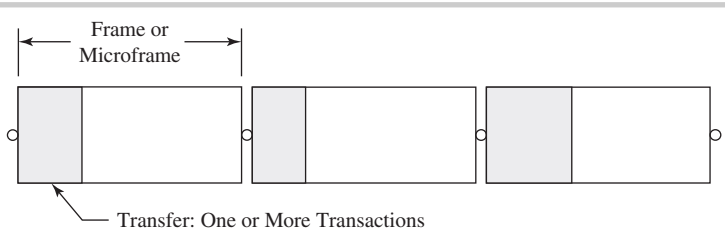


Figure 5.20

Timing of data transfers on a USB bus

5.10.3 Transfer Modes

There are four different modes of data transfer through a USB. These are control, bulk, interrupt, and isochronous. The application typically determines which mode to use. The **control mode** is used to obtain information about the device and to set the device address and configuration parameters. This mode must be supported by all USB devices. The **bulk mode** is typically used in applications where data transfer rates are not critical (such as sending data to a printer, receiving data from a scanner, or performing file access operations on a disk). The **interrupt mode** is used when data transfer must occur within a specific amount of time or latency and is typically used in mouse or keyboard interfaces and in data-acquisition applications. In **isochronous mode**, the data is transferred at a constant rate, and this mode is typically used in audio and video data-streaming applications. There is a tradeoff in the different modes in terms of error detection, recovery, and bandwidth.

In a USB connection, the data can be transmitted at three different speeds. In the slow-speed mode, the maximum data transmission rate is 1.5-M bits/s, and this speed is used for devices such as a USB mouse. In full-speed mode, the maximum data transmission rate is 12-M bits/s, and this speed is used for most devices. The high-speed mode (USB 2.0 standard) allows rates to up to 480-M bits/s, with a typical speed range of 25- to 400-M bits/s. Note that the slow-speed mode does not support bulk and isochronous data transfer modes.

5.10.4 USB Support on PIC Microcontrollers

Some of the PIC MCUs support USB communication (such as the PIC18F4550 MCU). The USB interface in these MCUs can be made to support several USB communication classes (such as HID and Communications Device Class (CDC)). We will talk about the CDC class here. The **CDC** is a class that covers a large number of telecommunication devices (such as analog phones, digital phones, cable modems, Ethernet adapters, and virtual COM-port devices). When a PIC MCU with USB support for CDC is connected to a PC, it will appear on the PC as a virtual COM-port. The PC can communicate with the device using standard RS-232 communication routines.

The PIC-C compiler has support for USB communication. It has a number of functions to handle USB communication for any class (such as *usb_attach*, which attaches a PIC device to the USB bus; *usb_put_packet*, which places a packet of data into a specified endpoint; and *usb_get_packet*, which reads a maximum number of bytes from a specified endpoint). It also has a number of routines to use exclusively with the CDC class. These routines are placed in a library called the CDC. These routines have a calling format similar to standard RS-232, and they hide from the user all of the details of USB communication. For example, the routine to get a character from the receive buffer is *usb_cdc_getc()*, and the routine to send a character is *usb_cdc_putc()*.

Figure 5.21 shows an example of code using the CDC library to communicate with a PC. The code sends back to the PC the character 's' if it was received by the MCU and ignores all other characters. The *#include <usb_cdc.h>* file has all of the needed code for CDC communication. Note that the function *usb_init_cs()* needs to be called once to initialize the USB hardware. Also, the routine *usb_task()* has to be called periodically. The *usb_task()* function monitors the USB bus for device connection/disconnection. It also enables and uses the USB interrupt. Because the CDC class makes the USB port looks like a COM-port to the PC, one also needs to set the RS232 communications parameters (done here with the *#use rs232* command).

```
////////////////////////////////////////////////////////////////////////////
///                    USB_IN_OUT.c
///
///   This program demonstrates USB communication using the CDC library
///
///     Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////
#include <18F4550.h>
#fuses HSPLL,NOWDT,NOPROTECT,NOLVP,NODEBUG,USBDIV,PLL5,CPUDIV1,VREGEN
#use delay(clock=48000000)

#use rs232(baud=38400, BITS=8, xmit=PIN_C6, rcv=PIN_C7,ERRORS)
#include <usb_cdc.h>

void main()
{
usb_init_cs();                  //Initialize USB Hardware but do not wait for device to be
                                // connected to bus
char c;

while ( 2 > 1)                  // Start infinite loop
  {
  usb_task();                   // Monitor Bus
  if (usb_cdc_kbhit() == 1)     //Check if there is a char available in the receive buffer
    c = usb_cdc_getc();         // Read char to variable c
  if (c == 's')
    {
    printf(usb_cdc_putc,"%c\n \r",c); // send the char back if it is 's'
    c= ' ';
    }
  }
}
```

Figure 5.21

Code listing that uses the CDC class in PIC-C compiler

When using a PIC MCU with a USB, one has to be careful in specifying the oscillator configuration for the chip. As mentioned in Chapter 4, in chips that support USB communication (such as the PIC18F4550), an additional clock branch is provided on the chip to give a 48 MHz clock for full-speed USB operation. For the USB operation, a phase locked loop (PLL) circuit on the MCU is used to provide the 48-MHz clock signal using as input clock sources that can range in frequency from 4 to 48 MHz. In the code example shown in Figure 5.21, we have used a board that has a 20 MHz crystal as the external clock source. Thus, in specifying the clock information using the *#fuse* settings, we defined the clock source high speed with phase locked loop (HSPLL). We also used the fuse setting PLL5, which means that the external oscillator clock frequency is divided by a prescale factor of 5. This is needed to cause the input to the PLL circuit to be at 4 MHz (20 MHz/5), which is the required input frequency to the PLL circuit. The PLL circuit gives a clock output of 96 MHz, so the fuse setting USBDIV is used to cause the output signal frequency to be divided by 2, thus giving the required 48 MHz clock.

5.11 Network Connection

A serial interface is an example of point-to-point network connection. When a PC or a MCU needs to interface to many devices, a network form of connection is more appropriate. A common method of network interfacing is the Internet. Some 8-bit PIC MCUs support Ethernet communication; for example, the PIC18F97J60

family is one which the reader can refer to for further information. This section will discuss Internet interfacing on PCs.

5.11.1 Structure and Operation

There are several protocols for Internet communication (see [17-18] for further reading). The **Transport Control Protocol/Internet Protocol (TCP/IP)** is the basis of the Internet. TCP and IP were developed by the Department of Defense (DoD) to connect a number of different networks into a network of networks (i.e., the Internet). The TCP/IP protocol is based on a stacked layered model, where each layer performs a separate function. There are two commonly referenced models. These are the Open Systems Interconnect (OSI) model, which has seven layers, and the DoD model, which has four or five layers depending on its version. The DoD four-layer version is shown in Table 5.2. The topmost layer is related to the application program, while the bottommost layer is related to the physical transport of the data. At the application layer, several protocols are used (such as **Hypertext Transfer Protocol (HTTP)** and **File Transfer Protocol (FTP)**). At the transport layer, there are several protocols including the User Datagram Protocol (UDP) and the TCP protocol. At the Internet layer, protocols such as IP route the data to the proper address. At the network access layer, several methods can be used (such as Ethernet or Wi-Fi). We will focus on the use of the TCP protocol in the transport layer. The TCP protocol assembles a message or file into a group of packets that are sent over the network and performs re-assembly of the packet when it arrives at its destination.

Table 5.2

Four layers TCP/IP model

Layer	Function	Example Protocol
4- Process/Application Layer	Applications that use network	HTTP, FTP
3- Transport Layer	Data delivery service	UDP, TCP
2- Internet Layer	Routing of data	IP
1- Network Access Layer	Access of physical networks	Ethernet, Wi-Fi

IP Address The Internet Protocol (IP) address gives the location for data transmission and sourcing on a network. IP addresses are currently unsigned 32-bit numbers that are displayed in **IPv4 dotted-quad notation**. In this notation, the address is displayed as four decimal numbers that are separated by dots with each number corresponding to one byte of the 32-bit address. The Internet regulators assign a range of addresses to different organizations. These organizations in turn assign their addresses to different departments within their organization. For example, in large organizations, the first two bytes represent the organization address, while the last two bytes represent the computer number or workstation in that organization. Using a 32-bit address space gives only a total of about 2.9 billion publicly available addresses. Some addresses are reserved for private networks or government use. To allow for a much larger number of addresses, a new format of IP addresses was proposed. The new format, called the **IPv6 colon-hexadecimal notation**, uses 16 bytes (or 128 bits) to represent the address. In the IPv6 notation, the address is displayed as eight four-digit hexadecimal numbers with colons separating each of the 16-bit blocks. In addition to providing an inexhaustible supply of addresses, the new address scheme enables a hierarchical routing infrastructure that is designed for more efficient routing. Figure 5.22 displays an IP address in both formats.

Figure 5.22

IPv4 and IPv6 addresses

IPv4 Address notation:
192.168.1.101
IPv6 Address notation
2002:C0A8:165:0:0:0:0:0

A **static address** is a permanent IP address (similar to a fixed telephone number) while a **dynamic address** is generated from a pool of available addresses by

some Internet service providers when one connects to the Internet. Dynamic addresses are used to overcome the limitation of limited IP address availability.

Server and Client The common programming model for Internet communication is the **client/server model**, where each computer or process on the network is either a client or a server. In this model, the client requests a service from the server, and the server responds by sending the requested information to the client. Many clients can be connected to the server at one time. The servers are typically powerful computers or workstations. Another model for Internet communication is the **peer-to-peer model** (P2P), where each computer has a similar capability and can initiate communication with the other computer. The P2P model is commonly used in home and small office networks where a limited number of computers want to share resources (such as printers and scanners). It is not suitable to use for shared database applications.

Nodes A **node** is a computer or some other device (such as a printer) that is connected to the network. Nodes can be a gateway type (connected at the entrance to the network) or a host type, (connected at an end-point of the network).

Sockets and Ports A **socket** is the abstract designation for a network connection on a PC. If a PC has more than one process that is interfaced with the Internet, then each process needs to have its own socket. A **port**, on the other hand, is the abstract designation for the channel that the data is sent to. When a client requests a service from a server, it informs the server of the port number to which the data should be sent. The port numbers are unsigned integer values, and there are 64K possible ports. The port numbers are split into three groups depending on the application: the *well known ports* (0 to 1023), the *registered ports* (1024 to 49151), and the *dynamic* and/or *private ports* (49152 to 65535). The port assignments are regulated by the Internet Assigned Numbers Authority (IANA). The well known and registered ports should not be used without IANA registration.

Network Access There are several models for network access. These include rotating access and collision detection. In the **rotating-access method**, each node gets a turn in writing data to the network. In the **collision-detection method**, a node checks to see if there is any activity on the network before it writes the data to the network. If no activity was detected, it starts to write the information to the network. If while writing the information the node detected information on the network from other nodes (i.e., a collision), then all the involved nodes stop transmitting. Each node then waits a random time interval to reduce the probability of another collision before attempting again.

TCP and UDP Protocols The most popular network architecture is one that uses Ethernet for the physical layer and TCP/IP for the upper layers. The Ethernet or physical layer uses the collision-detection method for network access. Ethernet is not deterministic and a given PC on the network cannot guarantee when it can use the network. The TCP protocol uses handshaking and has a built-in structure for error detection and correction. This protocol is referred to as a connection-oriented protocol.

In the **user datagram protocol (UDP)**, there is no handshaking, and there is no error detection or correction above the network access layer. This protocol is connectionless, and using it, you cannot tell if a packet did not arrive at its target destination. The UDP is a faster protocol than TCP due to the use of a smaller header in its packet structure and the lack of error detection and correction. Due to

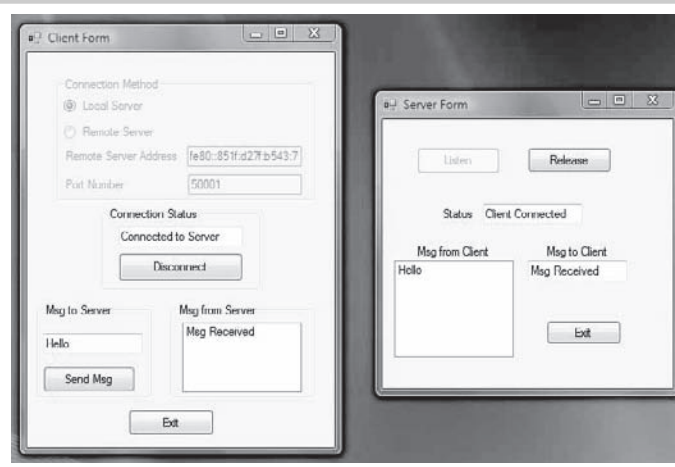
this, UDP is suitable only for private networks where network traffic is very predictable and overload is unlikely to occur. It also is commonly used to send streaming audio and video data when there is a need for high-speed transmission but the fidelity of the transmitted data is not of utmost concern.

5.11.2 VBE Programming Support

VBE provides programming interfaces for several of the protocols used on the networks today. It supports both the TCP/IP protocol and the UDP protocol that were discussed as part of the transport layer. We will discuss an example application that has two programs running: one acting as a client and the other as a server. These programs can run on the same PC or on two different PCs. A complete listing of the code for this application is provided on the text website. Figure 5.23 shows the interface screens for these two example programs. To start the server program, the user clicks on the *Listen* button. This makes the program wait for a connection to be established by the client. If the server is listening, then the client can connect to the server by pressing *Make Connection* button. Any message that the user types in the *Msg to Server* textbox is sent to server when the client hits the *Send Msg* button. The server will display the received message in the *Msg from Client* textbox. The server also replies back to the client with a message, which is the content of the *Msg to Client* textbox. The server can terminate the connection by pressing the *Release* Button. Also, the client can disconnect from the network by pressing the *Disconnect* button. The client can specify the address of the server to be connected to or can choose a local server connection. In the latter case, the server Internet address is automatically obtained by interrogating the **Domain Name System (DNS)** object for the PC on which the client and server programs are running.

Figure 5.23

Interface screen for the server and client example programs



Let us now look at the details for this example application. To implement a network connection based on the TCP/IP protocol, the *TcpClient* and the *TcpListener* objects need to be used.

A **new client** in the client program is created with the command:

```
Client = New Net.Sockets.TcpClient (server_address, port_num)
```

where *server_address* is the IP address of the server to which the client desires to communicate with and *port_num* is the desired port number on the client machine.

Once the client has been created, a stream is established for communication using the *GetStream* method with the command:

```
Stream = Client.GetStream()
```

To send data to the server, the *Stream.write()* method is used. The message to be sent has first to be parsed into ASCII format and stored in an array of bytes before being sent to the port. This is done using the command:

```
Dim Sentdata As [Byte]() = System.Text.Encoding.ASCII.GetBytes(message)
```

Similarly to reading items from the server, the *Stream.read()* method is used. In the same fashion, the data that is received in bytes need to regrouped into a message using

```
ReceivedMessage = system.Text.Encoding.ASCII.GetString(data, 0, num_of_bytes)
```

where *data* is the byte array containing the data that is received from the port and *num_of_bytes* is the array size.

Similarly, a **new server** in the server program is created using

```
Server = New TcpListener(local_add, port)

Server.Start()
```

where *local_add* is the IP address for the server machine. Once a request from the client to connect to the server is received, a client reference is created in the server program using

```
Sclient = Server.AcceptTcpClient()
```

Similar to the client program, communication with the client is done using a stream established with the command:

```
Stream = Sclient.GetStream()
```

Note that the *Stream.read()* method is blocking (i.e., it has unpredicable execution time) if no data is available. To prevent blocking code, the *Stream.DataAvailable* property should be checked to see if data is available, and the code that handles the communication in both the client and server programs should be structured as a state-transition diagram (covered in Chapter 6). Figure 5.24 shows an example of state-transition diagrams for the client and server programs. Notice that once communication has been established between the client and the server, the client program is in the *Wait* state. The program remains in this state unless data is available from the server or the client wants to send a message to the server. Similarly, the server program remains in the *Wait* state unless a request comes from the client.

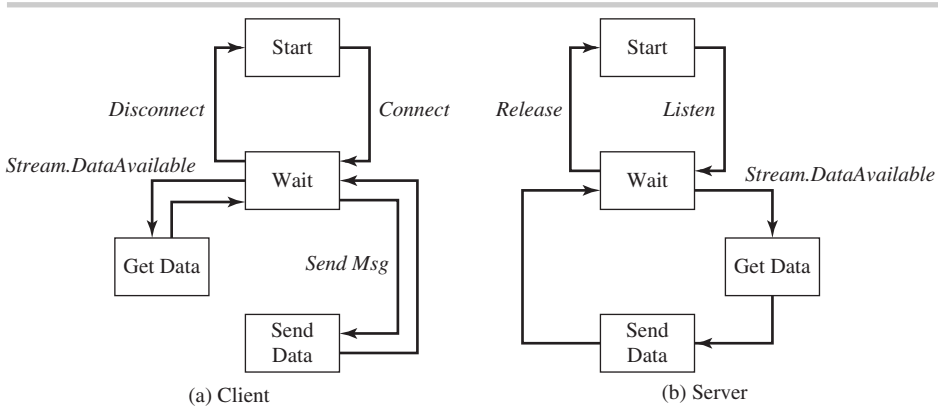


Figure 5.24

State-transition diagram for (a) client program and (b) server program

5.12 Chapter Summary

This chapter discussed means for interfacing a processer to external devices. It started with a discussion of Shannon's sampling theory, which states that the sampling frequency should be at least twice that of the highest frequency in the signal in order to prevent aliasing. This theorem applies when an analog signal is 'sampled' to obtain the digital values. It was then followed by discussion of the operating principles of an analog-to-digital converter, a digital-to-analog converter, and a parallel port. For A/D and D/A converters, the important characteristics of sampling rate, voltage range, bit resolution, and quantization error were discussed. Single and differential input modes of an A/D converter were also discussed concerning the differential input mode being a better configuration for handling noise than the single-ended mode. The features of a commercial data PC data-acquisition card were also presented. Serial interfacing techniques were covered next. The legacy RS-232 communication protocol was discussed in detail along with software support in the VBE and PIC-C compiler. To provide higher communication speeds, many PIC chips have a built-in synchronous serial port (SSP) module. Both modes of operation of the SSP (the Serial Peripheral Interface (SPI) and the Inter-Integrated Circuit (I^2C™) interface) were discussed along with example code using the PIC-C compiler. The basics of USB communication were discussed along with an example code for the PIC-C compiler that illustrates the use of the CDC class. The last section covered Internet interfacing, including both the TCP/IP and UDP protocols. The development of a client/server application using VBE was illustrated.

Questions

5.1 Explain what is meant by signal aliasing.

5.2 What affects the voltage resolution of an A/D converter?

5.3 What is the purpose of differential wiring in A/D reading?

5.4 Does the PIC16F690 MCU have a D/A converter?

5.5 What advantages does serial interfacing have over parallel interfacing?

5.6 How is a PC interfaced to analog signals?

5.7 Explain the different types of parity methods used in the RS-232 protocol.

5.8 List the differences and similarities between RS-232 interfacing and SPI/I^2C interfacing.

5.9 Which serial communication method has the fastest data transfer rate?

5.10 Name two Internet protocols.

5.11 Is USB communication allowed between devices?

5.12 Explain 'packets' and 'transactions' in USB communication.

5.13 Name two USB classes.

5.14 Name two programming models for Internet communication.

5.15 Identify a major limitation of Internet communication.

PROBLEMS

P5.1 What is the minimum sampling frequency needed to sample the following signals to prevent aliasing?

a. $f(t) = \sin(\pi t)$

b. $f(t) = 3\sin(2t) + 3\cos(2t)$

P5.2 Determine the digital output of a 10-bit A/D convertor with 0 to 5 V analog voltage range if subjected to the following analog inputs.

a. 1 V

b. 2.5 V

c. 5 V

P5.3 A PIC18 MCU with a 10-bit A/D has its A/D convertor set with $V_{ref+} = 2.5$ V, and $V_{ref-} = -2.5$ V. Determine the digital output of the A/D convertor if subjected to the following analog inputs.

a. −1 V

b. 0 V

c. 2 V

P5.4 A temperature sensor was connected to a 16-bit A/D converter with 0 to 5 V analog range. The sensor sensitivity is 10 mV/°C and the sensor output is 0 volts at zero degrees. Determine the following.

a. The temperature reading if the A/D converter output is 1000.

b. The measurement uncertainty due to the quantization error of the A/D.

P5.5 For an *R/2R* ladder resistor network similar to that shown in Figure 5.6 with $N = 8$, determine the voltage output of the network if bits 1 to 5 were connected to V_R.

P5.6 Estimate the time it takes send a file that has 20000 characters using RS232-serial interfacing if the baud rate is set at 38400 bps using a 10-bit data packet.

P5.7 Show the 10-bit serial packet for the following characters using a 7-bit data, one start bit, one stop bit, and a parity bit. Illustrate for both even and odd parity.

a. 5

b. L

P5.8 Perform research to identify three different sensors that use SPI or I^2C interfacing. For each sensor, list the manufacturer and part number, and the sensor details.

LABORATORY/PROGRAMMING EXERCISES

L/P5.1 Write a MATLAB program to show the effects of signal aliasing. Assume the input signal is sinusoidal with a frequency of 5 Hz. Plot the sampled signal if the signal is sampled at the following sampling frequencies: 4 Hz, 8 Hz, and 20 Hz.

L/P5.2 Build a circuit on a bread board to interface any PIC MCU with the DS275 or the MAX232/3 chip to enable the microcontroller to have RS-232 interfacing with a PC. Test your circuit by developing a program that allows the MCU to receive and transmit characters to a terminal program (such as *HyperTerminal* or *PuTTY*).

L/P5.3 Using a PIC development board with a built-in RS-232 interfacing (such as Microchip PIC18 Explorer board or Olimex PIC-STK-USB board), write a program that allows the MCU to receive and transmit characters to a terminal program (such as *HyperTerminal* or *PuTTY*).

L/P5.4 Using the VBE serial component, write a VBE program that sends and reads data from a COM port. For flexibility, write the code to allow the user to select a particular port from the available COM ports.

L/P5.5 With reference to the data sheet of a digital temperature sensor (such as MAXIM DS1631 sensor that uses I^2C interfacing), do each the following.

a. Build a circuit on a bread board to interface this sensor to a PIC MCU.

b. Develop a program to read the temperature measured by the sensor.

c. Display the read temperature by transmitting the temperature data to a terminal program.

L/P5.6 With reference to the data sheet of a digital potentiometer (such as Microchip MCP42050-I/P that uses SPI interfacing), do each of the following.

a. Build a circuit on a bread board to interface this potentiometer with a PIC MCU.

b. Develop a program to set the resistance of the potentiometer. Note that the resistance is set by a sending a byte to the chip. Use a multimeter to verify the set resistance.

L/P5.7 Using any PIC microcontroller development board, do each of the following.

a. Build a circuit on a bread board to interface the PIC MCU to a small DC motor with a tachometer. Use the A/D converter on the MCU to read the tachometer output and a PWM line to actuate the motor through a transistor or H-bridge driver.

b. Develop a program for the MCU to vary the speed of a DC motor by changing the duty cycle of the PWM signal sent to the transistor or the H-Bridge driver. Use the rotary pot on the development board as a speed dial to vary the desired motor speed. Set the duty cycle as a function of the desired speed, which is read from the 10-bit A/D channel connected to the rotary pot. Read the actual motor speed from the tachometer, and display it to the user on a terminal program using the RS-232 interface.

L/P5.8 Develop a VBE program to test the A/D, D/A, and digital I/O functions on a data acquisition card. For testing the A/D and D/A, the program should allow the user to specify which channel to test. In addition, the program should provide three testing modes: single reading from or writing to a particular channel, repeated testing of a particular channel (for reading only), and continuous testing of a combination of an A/D and D/A channels (for example reading a sine wave through the A/D and sending it back through the D/A). In the second testing mode, the program should allow the user to specify the number of times the individual reading need to be taken, and should display the average reading and the range of the readings taken during the test. For testing the digital I/O port, the program should be able to send a value (0 or 1) to any one of the 8 bits on the digital output port without affecting the current values on the other remaining 7 bits. For testing the digital input port, the program should be able to read the input value on any one of 8 bits. (Note: This problem assumes the availability of a data-acquisition card with a software library for accessing the A/D, the D/A, and the parallel port.)

L/P5.9 Modify the provided code for the client-server (see text website) to perform remote control or actuation. Specifically, do each of the following.

a. Run the server code on a PC that has a data acquisition card or I/O capability. Add code to allow the server program to interface with a motor and/or sensors on the server station.

b. Run the client code on another PC, and use the client program to issue commands to the physical system that is attached to the server. Examples of such commands are to start/stop the physical system or to get status information (such as speed or temperature).

CHAPTER 6

Control Software

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- Explain the concept of time and timers
- Outline how timers are implemented in different computing platforms
- Organize the operation and control of physical systems into tasks and states
- Develop state-transition diagrams for control of physical systems
- Explain code organization in state-transition diagrams
- Implement state-transition diagrams in different computing platforms
- Explain the concept of a thread
- Identify mechanisms for resource sharing
- Explain the operation of an RTOS
- Develop user interface for the PC portion of a control system

6.1 Introduction

The previous two chapters discussed microcontrollers and the means of interfacing microcontrollers or PCs. This chapter focuses on software development issues when using a microcontroller (and to a lesser extent, the personal computer) as the controller in a mechatronic system. Some of these issues include how to incorporate time into a control program, how to structure the operation and control of physical systems into tasks and states, and how to write control code that is suitable for real-time implementation. Incorporating time is essential in many software situations, especially in digital implementation of closed-loop control systems, since the sampling rate, which is the frequency at which the control signals are sent to the system, affects the response of the system.

Software development is a very important piece in the development of a mechatronic system. Without software, a MCU or PC cannot function. A software-based control system offers flexibility over a hardware-based one, since the controller structure and control logic can be changed by simply changing the code in the program. Due to the different types of control activities that need to be performed (such as feedback control or discrete event control), it is advantageous to develop a uniform control software structure that can handle a variety of control applications. This chapter presents such a structure that is based on the task/state software structure. According to Lyshevski [19], developing a control software structure is one of the most challenging problems in mechatronics system design.

In many situations, more than one control task needs to be controlled at the same time, and this chapter addresses the issue of multitasking. Multitasking brings the problem of resource sharing among the different tasks and the tools available to properly share these resources. The chapter also discusses the requirements for real-time operating systems (RTOS) and discusses two commercial RTOS for use in microcontrollers. The last section of this chapter addresses the development of graphical user interfaces of the PC portion of a control program. Approaches using Visual Basic Express (VBE) and MATLAB are discussed.

6.2 Time and Timers

Time is a very important element in the software used in mechatronic applications. Timing needs in mechatronic applications include the use of a timer to record the time of occurrence of an event (such as when the temperature of a process reaches a certain value), to implement time delays, and to schedule repeated execution of code segments (such as those used for monitoring and feedback control).

Some programs used in mechatronic applications are called **real-time programs**. A real-time program is one where the timing of the output result is as important as the result itself [20]. Not only does the program have to produce the correct output, but that output has to be produced at a certain time or within a specified interval, otherwise the output is worthless. For example, if you wrote a program to find a solution for the roots of a quadratic equation, the program operation and output is not affected by what time you start the program or how long it takes to solve for the roots (as long as it is done in a reasonable time interval). On the other hand, in a program to implement anti-lock braking, it is very important that the program output is produced in a very short interval from the instance that wheel skidding was detected, otherwise a catastrophic failure could happen. Another example where the timing of the control output is important is in the cutting of a sheet of material by a heat source such as laser. Here, the cutting operation is very dependent on how long the laser beam remains over the cutting area (controlled by the speed of the laser head relative to the material). If the exposure time is not controlled properly, then either the material will not cut due to underheating or it will burn due to overheating. Hence, in the laser-cutting example, the timing of the control output is important.

We will consider both **absolute** and **relative** timing modes as well as how to implement time in software. In some applications (such as alarm monitoring and scheduling), there is a need for absolute time, which includes the date and time information. As an example, consider a programmable thermostat to control the temperature in a house. The thermostat includes the means to allow the user to set a certain temperature value, depending on the time of the day (morning or night) and the particular day of the week. Thus, the thermostat, which is an example of an ON-OFF type of controller, synchronizes its operation with absolute time. Another application where absolute time is important is in security or alarm monitoring, where the time when an event happens (such as the opening of a door) is important and needs to be recorded.

Relative time or interval timing is needed when we need to implement a particular sampling rate in feedback control systems or to implement time delays. In interval timing, the time interval from the current instance to the future instance in which the next operation needs to be performed is controlled, but not at what hour of the day or day of the month.

Regardless of the mode of timing, a timer is implemented in a computer system using a combination of a clock and a counter. The **clock** is any device (such as a stable crystal oscillator) that can generate a uniform train of pulses at a particular frequency. These pulses are fed to a counter, which keeps track of them. To access the counter, the program simply reads the counter value through a function call. Counters can operate in two modes: count-up or count-down. In the **count-up** mode, the counter starts at 0 (or any other value), and then the count value is incremented by 1 whenever a new pulse has arrived. In the **count-down** mode, on the other hand, the counter is loaded with a starting value, and this value gets decremented with the arrival of each new pulse. Due to the finite size of counters (i.e., 16-bit or 32-bit), a count-up counter will overflow when the count exceeds the maximum number of counts for that counter. Similarly, a count-down counter will overflow when the count goes below 0. When a counter overflows, the counter is reloaded with the starting value, and the counting process repeats.

An important characteristic of a timer is its **resolution**. This refers to the smallest time change that can be measured by a timer. For a clock that generates pulses at a frequency of f pulses per second, the timer resolution in seconds is

$$\text{Timer resolution} = 1/f \tag{6.1}$$

The maximum time interval that can be measured by an n-bit counter before it overflows is given by

$$\text{Maximum interval} = 2^n/f \tag{6.2}$$

For example, a clock that operates at a frequency of 12000 Hz and a 16-bit counter have a maximum time interval of $65536/12000 = 5.461$ s.

In **relative-timing** mode, time intervals are obtained by dividing the difference between two counter readings by the clock frequency. For example, a counter operating in count-up mode has the time interval given by

$$\Delta T = (C_2 - C_1)/f \tag{6.3}$$

for $C_2 > C_1$, *and*

$$\Delta T = (2^n - C_1 + C_2)/f$$

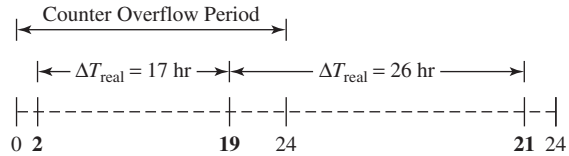
for $C_2 < C_1$, where C_2 and C_1 are the counter readings at the end and start of the interval, respectively, and n is the bit size of the counter. C_2 can be less than C_1 if the second reading of the counter was obtained after the counter has overflowed.

To determine time in absolute timing mode, the starting time is first synchronized with a reference time (such as the current time and date information). The absolute time is then determined by adding the interval between the current time and the last time the counter was read to the starting time. To have accurate absolute timing information, the counter has to be read at least once during an overflow period.

To illustrate **counter overflow problems**, let us consider a counter that overflows every 24 hours (similar to the *Timer* property in VBE, which will be discussed later). If this counter was read after one hour from the last time it overflowed, and then read again after six hours from the last time it overflowed, then by taking the time difference between these two readings, we say that five hours have elapsed. However, if the second reading of this counter was done the next day instead (29 hours after the first reading), we still get a difference of five hours between the two readings, although the two readings are actually separated by 29 hours. This is because, when the counter overflows, it resets itself back to zero.

Figure 6.1

Illustration of counter overflow



For accurate time keeping over periods longer than the counter overflow period, the counter has to be read at least once during each overflow period to prevent timing errors resulting from overflow of the counter. The successive time intervals between the current and the last time the counter was read are added to obtain the current time. The question that now arises is how to implement a scheme to check that the readings are actually done before the counter overflowed. The answer is to set a maximum read interval for the counter that is less than the overflow interval and to call any reading in which the time difference exceeds the maximum read interval as overflow. This scheme, which was proposed in [2], may not be able to detect all overflow instances, but should be able to detect the overflow error over many readings of the counter.

To illustrate this, let us consider again our counter example that overflows once every 24 hours and set the maximum read interval to 16 hours. Assume the counter readings were 2, 19, and 21 (see Figure 6.1). Then according to this scheme, we get an overflow at the second reading, since the difference between the second and the first reading is 17 hours, which is greater than the maximum read interval that we set (16). On the other hand, at the third reading of 21, this scheme does not detect an overflow error, since the difference between the third and the second counter readings is only 2, while in reality the third reading was obtained in the second day. Nevertheless, this scheme will be able to detect some overflow errors over many readings of the counter. Detection of such errors tells us that we need to revise our code to read the timer more often.

6.3 Timing Functions

Most programming languages provide functions for accessing time information that make use of the hardware timer that is available on the microcontroller or the PC. In this section, we will discuss timer implementation in PCs and PIC microcontrollers. In PCs, we discuss timer implementation in MATLAB and Visual Basic Express. We also discuss the performance counter that is available on some PCs.

6.3.1 Timer Implementation in MATLAB

MATLAB has several functions to obtain timing information. In this section, we will discuss

- TIC and TOC functions
- CLOCK and ETIME functions
- TIMER object

The **TIC** and **TOC** built-in functions are designed to be used together and use the PC clock. The TIC function starts a stopwatch timer, while the TOC function reads the current time in seconds from when the TIC function was called. To get timing information, the TIC function is called first. A subsequent call to the TOC

function returns the elapsed time since the TIC function was called. For example, if we type in the command window:

```
tic
toc
```

Then MATLAB will print:

```
Elapsed time is 2.647394 seconds.
```

The TIC and TOC functions are useful to use as an interval timer. For example, the following MATLAB code (see Figure 6.2) can be used to check if a certain time interval (such as 10 s) has elapsed.

```
tic;
start_time = toc;
while (toc - start_time) < 10

end
disp ('10s seconds has elapsed')
```

Figure 6.2

MATLAB code listing for implementation of an interval timer

The TIC/TOC functions have a sub-millisecond time resolution, but the actual resolution obtained in a given application is dependent on the computer hardware used and on what other applications are running at the same time.

The **CLOCK** function returns a six-element vector that has the current date and time information in decimal format. Typing CLOCK in the command window gives

```
clock
ans =
1.0e+003 *
2.0100   0.0010   0.0140   0.0120   0.0020   0.0128
```

where the first element is the year information and the remaining five elements are respectively: month, day, hour, minute, and seconds. To access only one element of the date and time information, the CLOCK function is assigned to a variable, and the corresponding element of that variable is used. To get the information in integer format, the clock function is called using the FIX function or *fix(clock)*. The **ETIME** function along with the CLOCK function can be used to determine the time (in seconds) that has elapsed between two values of clock vector. Thus, the following code will give the elapsed time:

```
t1 = clock;
etime(clock,t1);
```

MATLAB does not recommend the CLOCK and ETIME functions for accurate timing, since they are based on the system time, which the operating system can adjust periodically. The TIC/TOC functions give a more accurate event or interval timing.

MATLAB allows the creation of a **TIMER** object that can be used to automatically time the execution of a special function called the timer callback function. The TIMER object is created using the command:

```
T = timer('PropertyName1', PropertyValue1, 'PropertyName2', PropertValue2, ...)
```

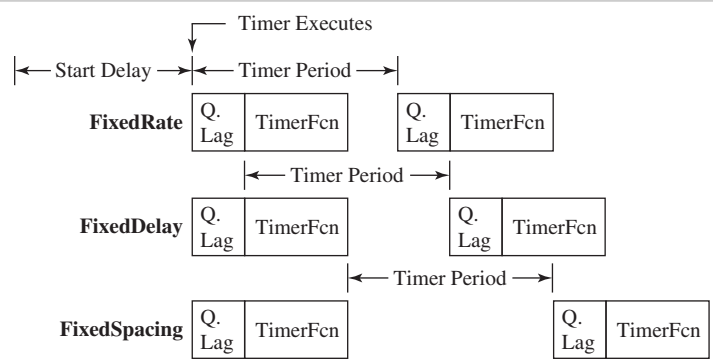
Table 6.1
TIMER object properties

Property Name	Property Description
AveragePeriod	Average time between *TimerFcn* executions since the timer started.
ExecutionMode	Determines how the timer object schedules execution of the *TimerFcn*.
Period	The delay in seconds between execution of the *TimerFcn*.
TasksToExecute	The number of times the timer should execute the *TimerFcn* if the *ExecutionMode* is not *singleShot*.
TasksExecuted	The number of times the timer has called the *TimerFcn* since the timer was started.
TimerFcn	Timer callback function.

where *T* (or any other name) is the TIMER object, and the property name/value pairs are used to specify the operating characteristics of the TIMER object. Some of the available properties and their description are listed in Table 6.1.

The timer *Period* property cannot be less than 1 ms. If unspecified, the default setting is then 1.0 s. The *ExecutionMode* property of the timer defines how the timer is run. It can run once (if the property is *singleShot*, which is the default setting) or it can do multiple executions (if the property is set to *fixedDelay*, *fixedRate*, or *fixedSpacing*). Figure 6.3 shows the different execution modes. Note that in this figure the timer period is the same in all cases, but the point of time at which the execution begins is different. In the *fixedRate* mode, the timer period starts at the point where the timer callback function is added to the queue. In the *fixedDelay* mode, the timer period starts at the beginning of the execution of the timer callback function, while in the *fixedSpacing* mode, the timer period begins at the point where the timer callback function finishes executing. Note how the absolute timing of the execution of the *TimerFcn* is different in each mode.

Figure 6.3
Illustration of the different execution modes of the TIMER object



Note: Q. Lag means Queue Lag

Once the TIMER object has been created, it needs to be started. It can be started immediately by calling the function *start(T)*, or it can be started to run at a specified time using the function *startat(T, start_time)*. The timer can be stopped by calling the *stop(T)* function.

As an example, Figure 6.4 shows the code listing to create and start a TIMER object with period of 0.5 s. The timer has a callback function that needs to be executed 20 times in *fixedRate* mode. Included in this code is a listing of the *timer_callback_fcn*.

```
% Demo of Timer Object
% File: DemoTimer.m

function DemoTimer
% Create the timer object T
T =  timer('TimerFcn', @timer_callback_fcn, 'period', 0.5,'TasksToExecute', 20,'ExecutionMode',
'fixedRate');
% Start the timer
start(T);
% Wait for the timer to complete the tasks
while(get(T,'TasksExecuted') < 20)

% do nothing
end
disp('Tasks Execution is done');
% Remove the timer from memory
delete(T);

% Listing of the timer callback function
function timer_callback_fcn(obj,event)

disp('In timer call back function \n')
```

Figure 6.4

MATLAB code listing demonstrating the TIMER object

Once the timer has completed its execution, the timer should be removed from memory using the *delete* command. The TIMER object will be utilized later in this chapter for the implementation of state-transition diagrams in MATLAB.

6.3.2 Timer Implementation in VBE

VBE has several functions that access the counters that are kept by the operating system. In this section, we will discuss two of these timing functions:

- Timer property
- Timer component (Windows Forms): Allows Windows-based application to respond to events that are spaced regularly

The **Timer property** returns the number of seconds since midnight. It is called by using the following syntax, where *CurrrentTime* is the number of seconds since midnight at the instant at which this statement is executed:

```
CurrentTime = Microsoft.VisualBasic.DateAndTime.Timer
```

The Timer property is convenient to use for implementing an interval-type timer in a program. Figure 6.5 shows an example of code that uses the timer property to delay the execution of a program by 10 s. In this example, the code continuously monitors time by repeatedly calling the Timer property and will exit the

```
Dim StartTime As Double
StartTime = Microsoft.VisualBasic.DateAndTime.Timer
While (Microsoft.VisualBasic.DateAndTime.Timer - StartTime <= 10)

End While

MsgBox("10 sec time interval has elapsed")
```

Figure 6.5

A timing delay using the Timer property

While-Loop when the 10 seconds time interval had elapsed. At that point, the code prints a message to the user using the *msgbox* function.

Notice that this timer overflows once every 24 hours, and hence, it is not suitable to keep time for applications where the program needs to run for extended periods, unless means were made to address the overflow issue.

Figure 6.6
Timer component

Unlike the Timer property, which can be used in any type of VBE applications (Windows or Console), the **Timer component** only can be used in Windows applications. The Timer component is used when a certain code needs to be run periodically (such as to perform periodic monitoring of a sensor output or to implement a feedback controller that needs to run at a certain rate). To incorporate a Timer component into an application, it has to be placed on a VBE form (see the clock symbol in Figure 6.6).

When this timer is activated (by setting the enable property to true), this timer will cause a particular function to be called automatically at intervals that are specified by the *Interval* property of this component (value is in milliseconds). This automatic function is called *Timer**_Tick*, where * is the timer number (1, 2, etc.), and will continue to execute indefinitely as long as the enabled property is true. You can think of the *Timer**_Tick* function as an event-enabled function, where the event that triggers this function to execute is the elapse of the Timer component interval. For example, if the Timer component interval is set to 100 ms, then the *Timer**_Tick* function will execute approximately once every 100 ms, or ten times per second. Note that, because of the multitasking nature of the Windows operating system, the *Timer1_Tick* function does not execute precisely every 100 ms, but it is close to that. Furthermore, while the *Interval* property can be set as low as 1 ms, the effective resolution of the Timer component is about 15 ms, as tested by the author. Note that the *Timer**_Tick* routine offers a very convenient method for implementing a periodic activity (such as the monitoring of sensors or switch inputs) when timing precision is not very important.

6.3.3 Performance Counter

While the VBE timing functions discussed previously are easy to use, they do not have a high resolution. Fortunately, many processors have a high-resolution counter called the **Performance Counter**. This counter runs from a clock operating at a frequency of about 1.2 MHz or higher, so it has a sub-microsecond resolution. The counter is typically used by Windows applications to time the execution of sections of code. However, VBE does not provide a built-in function to access that counter. Using function calls to the application programming interface (API), one can access that counter (see code listing on text website). The class called *PerformanceTimer* provides functions to access that counter. The class makes use of two Windows operating system–provided functions: *QueryPerformanceCounter* and *QueryPerformanceFrequency*. The *QueryPerfomanceCounter* function returns the current value of the Performance Counter in counts. The *QueryPerformanceFrequency* function returns the frequency of the Performance Counter. If the installed hardware supports a Performance Counter, this function returns a non-zero value, otherwise it returns a zero. Note that the frequency of the Performance Counter is processor dependent. On some systems, the frequency is the cycle rate of the processor clock. With access to the Performance Counter, we can determine time intervals using Equation (6.3).

In order to use a timer that is based on the Performance Counter, a variable (such as *tmr* of type *PerformanceTimer*) has to be declared and an object created using a statement such as

```
Dim tmr As NEW PerformanceTimer
```

The timer is then initialized by calling the function *tmr.StartTimer()*, which stores the initial reading of the counter. The time is read using the *ReadTime()* function, which returns the time since the timer was started. The *ReadTime()* function makes use of the *tmr.TimeElapsed()* function, which divides the difference between the current counter reading and the initial counter reading by the counter frequency to return the time.

Notice that the Performance Counter has a field width of 64 bits. This means at an operating frequency of ~1.2 MHz, this counter will overflow once every 487,000 years, which is more than needed for all engineering applications!

6.3.4 Timing in PIC Microcontroller

Similar to PCs, microcontrollers also have timers that can be used for timing purposes. We will discuss the timing features on the PIC16F690, a popular 8-bit microcontroller (details of this microcontroller were covered in Chapter 4). The PIC16F690 has three timers called *Timer0*, *Timer1*, and *Timer2*. Table 6.2 lists information about these timers.

Table 6.2

Timers in PIC16F690 microcontroller

	Timer0	Timer1	Timer2
Bit Size	8-bit	16-bit	8-bit
Operate as a Counter?	Yes	Yes	No
Programmable Prescaler?	Yes	Yes	Yes
Prescaler Values	1:1 to 1:256	1:1, 1:2, 1:4, 1:8	1:1, 1:4, 1:16
Postscaler?	No	No	Yes
Postscaler Values	–	–	1:1 to 1:16
Maximum Timer Interval at 8 MHz Clock (with maximum prescale)	$0.128 \times 256 =$ 32.768 ms	$32.768 \times 8 = 262.1$ ms	$0.128 \times 16 =$ 2.048 ms
Timer Overflow Interrupts	Yes	Yes	Yes

Timers 0 and 1 can operate as either timers or counters. In the timing mode, the count value is incremented every instruction cycle (or at a multiple of it if a prescaler is used). In the counter mode, the Timer0 or Timer1 module will increment on every rising or falling edge of the external signal connected to a specific pin on the microcontroller. Note that all PIC microcontrollers execute instructions at a rate that is one-quarter of the clock rate. For example, if this microcontroller was running on an 8 MHz clock, then the instruction cycle rate is 2 MHz. The use of a programmable prescaler reduces the instruction cycle rate, as seen by the timer allowing for longer timing intervals to be counted by the timer before overflow. For example, if the 16-bit Timer1 was used as a timer with no prescaler (or a prescale value of 1:1), then at a clock rate of 8 MHz, this timer will overflow every 32.768 ms. This is obtained from Equation (6.2) or

$$\text{Max interval} = \frac{2^{16}}{2 \times 10^6} = 32.786 \text{ ms}$$

With a 1:8 prescaler, the overflow period will be eight times longer or 262.1 ms.

Any of the three timers can be used to keep time as long as they are read often to prevent timer overflow from affecting the accuracy of the reading. When any of

the three timers overflow, a flag is set up. The program can poll this flag to check for this overflow condition. Alternatively, the overflow of the timer can be set to generate an interrupt (see Chapter 4). A postscaler factor, which is only defined for Timer2, affects the interval at which the overflow flag is set. At a postscaler factor of 1:8 for example, it takes eight overflow intervals of Timer2 to generate a single setting of the overflow flag.

Figure 6.7 shows a code listing for implementing the *ReadTimeNow()* function using the PIC-C compiler. This function returns the time since the timer was started. Such a timing function will enable implementation of feedback control tasks in a PIC MCU using the cooperative control strategy that will be discussed later. Figure 6.7 also has code for setting up this timer (*SetupTimer()* function). Note that the *ReadTimeNow()* function returns the time in integer units, which are multiples of the timer resolution, since the *SetupTimer()* function was called. The *ReadTimeNow()* function has code to handle the case that the Timer1 counter has overflowed since the last read. Note that, because the *Time* variable is a 32-bit unsigned integer, the *ReadTimeNow()* function can read time correctly up to 2147 s ($2^{32} \times 0.5$ μsec) unless we use a larger prescale factor. Note that the variables *Time*, *TimerRes*, and *LastCount* have to be defined somewhere else in the code before they are used in the timing functions.

Figure 6.7

PIC-C code listing for implementing the *ReadTimeNow()* function

```
////////////////////////////////////////////////////////////////////////////
///   Code for setting up a timer in software
///
///   Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////
void SetupTimer(void)
{
setup_timer_1(T1_INTERNAL);              //At 8 Mhz internal clock and 1:1 prescale, the timer
                                         // has a resolution of 0.5 usec
  TimerRes = 0.0000005;
  LastCount = 0;
 Time = 0;
  set_timer1(0);
}
int32 ReadTimeNow(void)                  //Returns time in units that are multiple of the timer resolution
{
int16 ReadTime;
ReadTime = get_timer1();
if (ReadTime > LastCount)
  Time = (ReadTime - LastCount) + Time;
else
  Time = Time + ((65536 - LastCount) + ReadTime);

LastCount = ReadTime;
return(Time);
}
```

6.4 Control Tasks

The first step in setting up a control system in software is to organize the control actions that need to be performed into separate groups. Each group will be called a **task**. Depending on the nature of the control job, the grouping will be done differently. For example, if the control job is to control the operation of an assembly

machine that consists of a number of modules that work together, such as that shown later in Figure 6.12, then we need to set up a separate task to control each module of this assembly system. Similarly, if we need to control a multiple-axis positioning system, then we can set up a separate task for each axis of the system. Thus, the grouping of control actions in both examples is set based on the physical structure of the machine to be controlled. Depending on the complexity of the control actions that needed to be performed for each module or axis, more than one task can be designed to control that unit too. The grouping also can be done based on the software activities to be performed (such as handling of Internet communication, see Section 5.11).

Within a control task, the code should be organized into separate states. Each **state** signifies a distinct condition of the machine. A machine or a process can be in only one state at a time. A crude example of the state structure is the states of an aircraft flying from location *A* to *B*. Regarding the motion of the plane, we can separate it into five states. These are taxiing from the terminal to the runway at location *A*, takeoff at location *A*, flying from location *A* to *B*, landing at location *B*, and taxiing from the runway to the terminal at location *B*. Obviously, we can break up each of the above five states into further states, but the point here is that the plane, at any point of time, can be in only one of these five states. There are also distinct conditions that cause the transition from one state to the other. The states of a task and their transitions are shown graphically using a **state-transition diagram**.

As an illustration of task/state organization, let us create a state-transition diagram for a program that needs to generate the following periodic signal (see Figure 6.8). The program should start sending this signal pattern in response to a *Start* command and should shut OFF the output in response to a *Stop* command.

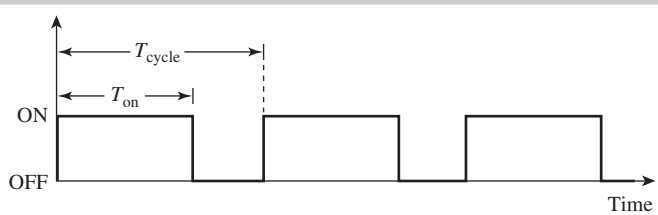


Figure 6.8

Periodic ON/OFF signal

The state transition diagram for this task is shown in Figure 6.9. We have broken the operation of this simple program into three states shown as rectangular blocks in the figure. The conditions that cause transitions between these states are shown in italics along the arrows that connect the blocks. The program starts in the

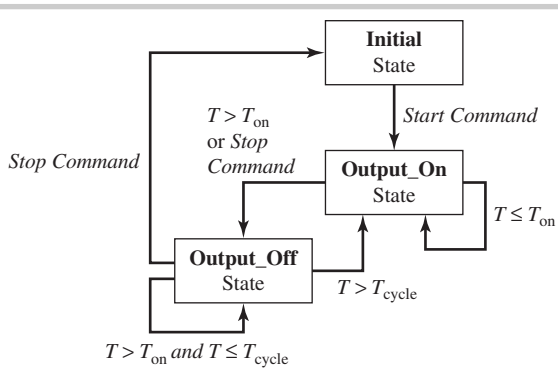


Figure 6.9

State-transition diagram for generating periodic signal

Initial state waiting for the *Start* command to be issued by the user. Once the *Start* command is received, the program switches states to the *Output_On* state where the output is turned ON. The program remains in this state while the elapsed time (T), from the beginning of each cycle, is less than or equal to T_{on}, which is the on-time interval. When the elapsed time exceeds T_{on}, the program switches to the *Output-Off* state, where the signal is switched OFF and remains in this state while the elapsed time is less than or equal to T_{cycle}, which is the cycle time. The program cycles between the *Output_On* and *Output_Off* states based on timing signals, unless the *Stop* command was issued, which causes the program to go back to the *Initial* State and wait for a new *Start* command. This example clearly shows that the program only can be in one state at a time and the transitions between the states are caused by either timing signals or user commands.

In this section, we will apply the state-transition diagram concepts to the control of discrete-event tasks and feedback control tasks. State-transition diagrams are widely used in the design of digital logic circuits. The material presented in this section uses concepts similar to those presented in [2] and [21].

6.4.1 Discrete-Event Control Tasks

Discrete-event control refers to the control of a sequence of actions. These actions include, among other things, opening or closing of a valve; turning on or off a heater, fan, or pump; monitoring of a sensor; or waiting for a time interval to elapse. Examples of discrete-event systems include operations of automated cutting, assembly, item dispensing machines, entry/exit systems, and process control systems. Some actions are time based (the valve should be on for 3 s) while others are sensor-based (the robot arm is commanded to go up when a part is available as detected by the part sensor).

The task/state control software structure is ideally suited for the control of discrete-event systems. The first step in controlling a discrete-event system is to model the system operation using a state-transition diagram. Example 6.1 illustrates this for **the operation of an automated entry door** that is commonly available in public places. Five states are used in the example with the transitions between the states being either sensor-based or time-based. Each of the five states represents a unique state of operation of the automated entry door. The control system for this door can be in only one of these states at any point of time. For this automated entry door, a single task is sufficient to represent the operation of the system.

Example 6.1 Automated Entry Door State-Transition Diagram

Develop a state-transition diagram for the operation of an automated entry door that is commonly available in public places. Assume that the control system will open the door when signaled from the sensor attached to the door. Assume also that the door will stay open while the detection sensor is ON and for a specified interval after the detection sensor switches to OFF.

Solution:

The state-transition diagram for the operation of the automated entry door is shown in Figure 6.10. The control system starts in the *Start* state, at which point the door is closed and the detection sensor is OFF. When the detection sensor turns on, the control system switches to the *OpenDoor* state. When this state first runs, control signals are sent to the actuators to open the door. The control system will stay in this state until the proximity sensors (or limit switches) that indicate that

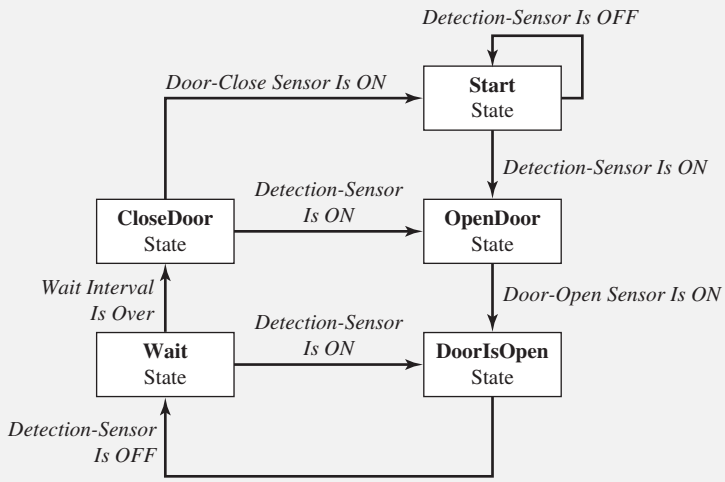


Figure 6.10

the door is fully open are trigged on. The control system then switches to the *DoorIsOpen* state and stays in that state until the detection sensor is no longer ON. For safety reasons, rather than closing the door right away, the control system switches to a *Wait* state to wait for the elapse of an interval timer before issuing the command to close the door. Since it is possible that while the control system is waiting for the timer interval to elapse the detection sensor is triggered again, the control system switches back to the *DoorIsOpen* state if this occurs. When the wait interval is over, the control system switches to the *CloseDoor* state. Similar to the *OpenDoor* state, the control system waits for the door to fully close before switching from the *CloseDoor* state to the *Start* state. Similarly, the control system can go back from the *CloseDoor* state to the *OpenDoor* state if the detection sensor was triggered while the door was closing.

To further illustrate this topic, let us consider the state-transition diagram modeling of the operation of a **digital heating thermostat**. The thermostat has a switch that turns it ON and OFF. The thermostat should operate according to the following rules:

a) The thermostat operates when the switch is in the ON position. The thermostat does not do any control if the switch is in the OFF position.

b) The thermostat turns ON the heating equipment when the room temperature is 1° below the desired temperature.

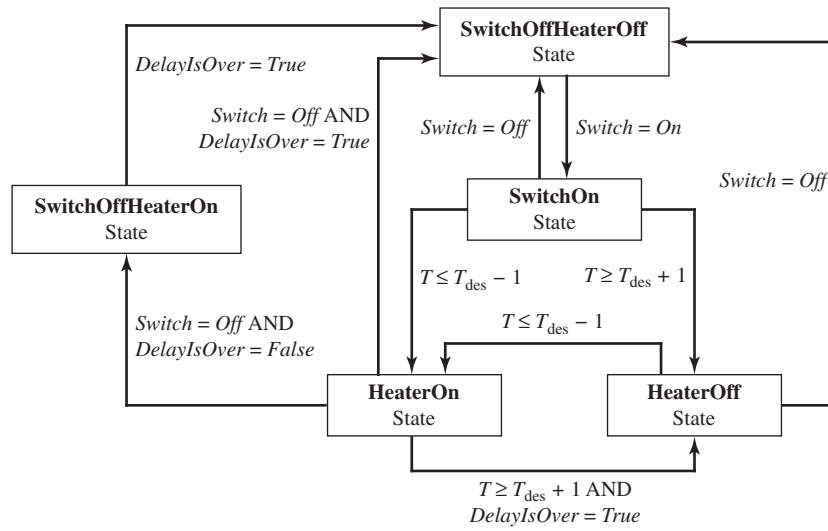
c) The thermostat turns OFF the heating equipment when the room temperature is 1° higher than the desired temperature.

d) When the heating equipment is turned ON, it cannot be turned off unless a user specified delay interval (such as 2 minutes) has elapsed since it was turned ON. This is done as to protect the equipment from rapid turn ON/turn OFF.

The state-transition diagram for the operation of a thermostat that follows the above rules is shown in Figure 6.11. It has five states. The state diagram starts in the *SwitchOffHeaterOff* state where the ON/OFF switch is OFF and the heater is OFF. When the user moves the switch to the *ON* position, the state changes to the *SwitchOn* state. There are three possible transitions from this state. If the current temperature is 1° or more below the desired temperature, then the state switches

Figure 6.11

State-transition diagram for the operation of a heating thermostat



to the *HeaterOn* state. If the current temperature is 1° or more above the desired temperature, then the state switches to the *HeaterOff* state. Moving the ON/OFF switch to the *OFF* position will cause the state to switch back to the *SwitchOffHeaterOff* state.

In the *HeaterOn* state, the heater is turned ON and the current time is recorded. Both of these actions should be done in a section of the code that is only executed once (more on this in the next section). From the *HeaterOn* state, the state switches to the *HeaterOff* state (where the heater is turned OFF) only if the temperature exceeds the desired temperature by at least 1° and the waiting interval since the heater was started is over. The other possible transitions from the *HeaterOn* State are activated by the user turning the switch to the *OFF* position. Since the switch can be turned OFF before the time delay interval is over, the state switches to the *SwitchOffHeaterOn* state (where the heater is still on) to wait for the interval to be completed before switching to the *SwitchOffHeaterOff* state where the heater is OFF. Note that at a steady state, the state diagram switches between the *HeaterOn* and *HeaterOff* states based on the relationship between the desired and current temperature. Similar to the automated entry door example, a single task is used to model the operation of the thermostat.

In **more complicated systems**, more than one task is needed to organize the operation of the system. This is the case for the operation of **automated assembly systems** such as the one shown in Figure 6.12, which is a simplified version of automated multiple-part industrial assembly systems. This assembly system assembles two components. A circular disk is placed on a rectangular base via the use of a four-station rotary indexing table. The rectangular and circular parts are fed using a conveyer belt. The assembly operation proceeds as follows. First, the rectangular part is transferred to station 1 on the indexing table, using pick and place robot #1. After the indexing table rotates 90° clockwise, the circular disk is placed on the rectangular base at station 2, using pick-and-place robot #2. After another 90° rotation of the indexing table, the two-part assembly of the circular and rectangular parts is inspected at station 3. After the third motion of the indexing table, the completed assembly (if passed inspection) is transferred to conveyer belt *C*, using pick-and-place robot #3. If the assembled part failed inspection, then either the machine operator is alerted to address this situation, or the failed assembly is picked and dropped onto

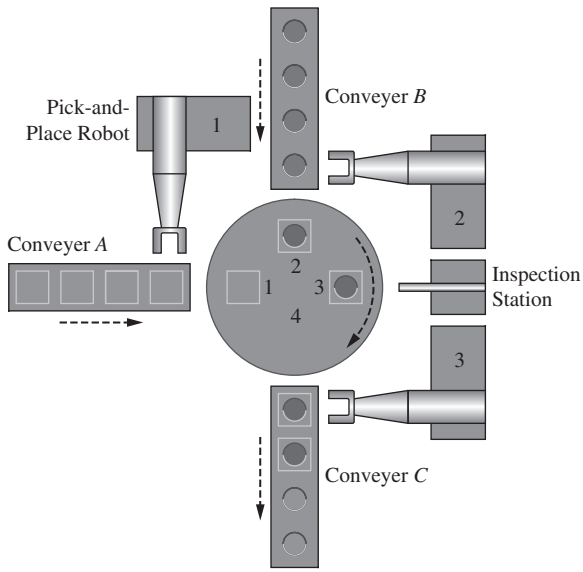


Figure 6.12

A simplified automated assembly system

a rejection bin (not shown here). At steady state, each station of the indexing table will be performing its part of the assembly process while the indexing table is stationary. When all of these actions are completed, the indexing table will rotate 90°, and the process repeats again. As a first step in designing a control system for this machine, the operation of each module in this assembly machine should be structured as a separate task. Each task will have a number of states depending on the operations done on that task. Example 6.2, which follows, shows the state transition diagram for the operation of pick-and-place robot #1 of this system.

Example 6.2 Assembly Robot #1 State-Transition Diagram

Draw a state-transition diagram for the operation of pick-and-place robot #1 in the assembly system shown in Figure 6.12. Assume that the robot used in this setup is a pneumatically driven one. Such robots are assembled from linear or rotary motion axes with two possible locations for each axis. Thus, moving the robot to the *Pick-Up Location* entails sending a digital signal to the solenoid valve(s) that controls the air supply to the particular robot axis (axes) that causes the desired motion. To determine if the robot has completed its travel, assume that end-of-travel proximity sensors (see Section 7.4), which are mounted at either end of the motion, are used. Picking or dropping a part corresponds to closing or opening the gripper that is attached to the end of the robot. Assume that a pneumatically actuated gripper is used here, and the opening and closing actions of the gripper are similarly accomplished by actuating the solenoid valve that controls the air supply to the gripper.

Solution:

The state-transition diagram for pick-and-place robot #1 is shown in Figure 6.13. The diagram consists of eight states, and the robot will be waiting in the *Start* state until it receives a signal from the cell controller to start its sequence and a part is ready for pick-up at the end of conveyer belt *A*. At this point, the robot will be in the *Move-to-Pickup-Location* state where it will be moving to the part pick-up location. When the robot reaches the part-pick-up location, as indicated

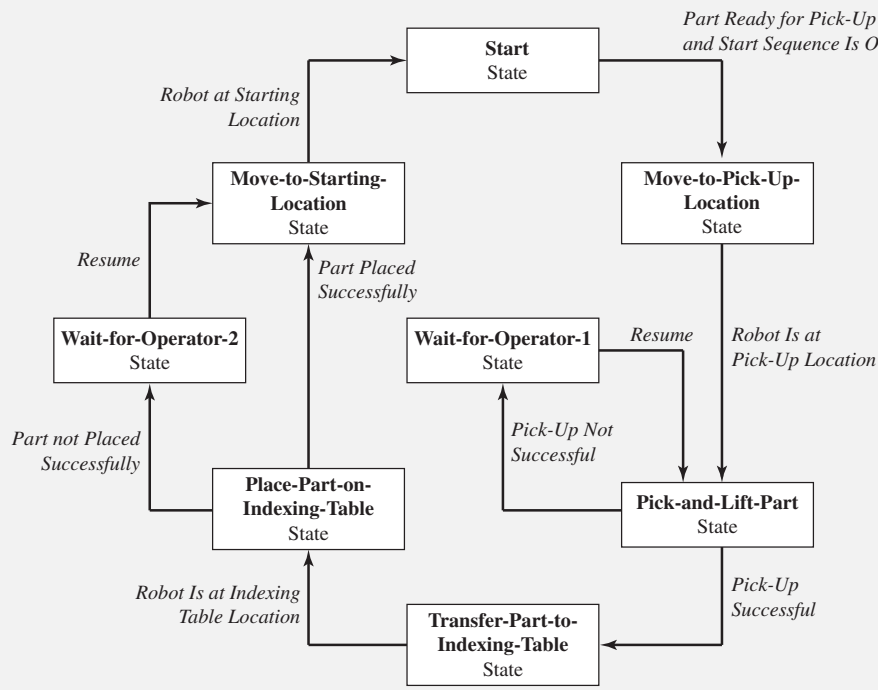


Figure 6.13

by feedback signal(s) from an end of travel sensor(s), the robot changes state to the *Pick-and-Lift-Part* state. If the pick-up was successful, the robot switches states to the *Transfer-Part-to-Indexing-Table* state, where the carried part is moved to station 1 on the indexing table. If the pick-up was not successful, the robot switches states to the *Wait-for-Operator-1* state. In this state, the operator is alerted to check the system, and on receiving a *Resume* signal from the operator, the pick-up is attempted again. Similarly, when the part has reached the drop-off location, the state changes to *Place-Part-on-Indexing-Table* state. After the part was successfully transferred to the indexing table, the robot changes state to the *Move-to-Starting-Location* state. Similarly, if the placement was not successful, the robot switches states and waits for a *Resume* signal from the operator before switching to the *Move-to-Starting-Location* state. When the robot reaches the starting location, the state switches back to the *Start* state. As seen from this example, each state of this diagram represents a distinct phase of operation for the robot. Transition between these phases is initiated by the appropriate sensor signals or operator input.

The state-transition diagrams for the other pick-and-place robots are similar in structure. The state-transition diagrams for the other components can be similarly developed, but each will have different states specific to that component. For example, part of the state-transition code for the **indexing table** should include a homing sequence to determine a starting position for the table. This homing sequence is executed once during the initialization phase. Thus, to control the operation of this assembly machine, we should have nine tasks: one for each component of the machine plus one task to schedule the operation of the machine. The function of the scheduling task is to decide which components of the machine will be active during a particular cycle of the machine. Note that in this assembly system, not all components of the machine will be active during the start and end

phases of the assembly. For example, in the first step of making the first assembly, all of the operations will take place at station #1 on the indexing table, and all of the components on the other stations will be idle. Similarly, during the last step of making the last assembly, all of the operations will take place on station #4, and all of the other components will be idle.

In this assembly example, some of the components could require feedback control action for their proper operation (for example, if the robots were servo driven). In this case, a feedback control task (to be discussed next) will be needed, but from the point of view of the task/state methodology, the structure would remain the same. We just replace the state that checks if the robot has reached its end of travel by monitoring a proximity sensor with one that waits for a flag that is set when the desired motion was completed by the feedback controller.

6.4.2 Feedback Control Tasks

Feedback control systems are used for regulation or tracking control applications. The analysis and design of feedback control systems is covered in Chapter 9. The advantage of a feedback control system is that it can follow the desired signal in the presence of disturbances. Furthermore, there is no need for calibration, since the input to the system is automatically determined by the feedback control action. Examples of feedback control tasks include speed and position control of electric motors, temperature control of ovens, and environmental control. The basic elements of a digital feedback control system are the controller, the controlled element, the feedback device(s), and interface components (such as A/D or D/A between the controller and the controlled element). Figure 6.14 shows a block diagram of a digital feedback control loop. Note that a digital control system is a sampled data system that operates at a sampling frequency that is a function of the dynamics of the system to be controlled, and hence, task timing is important. A typical **feedback control cycle** involves reading one or more of the output variables, computing a control input using a feedback control law, and transmitting the computed control output to the system under control.

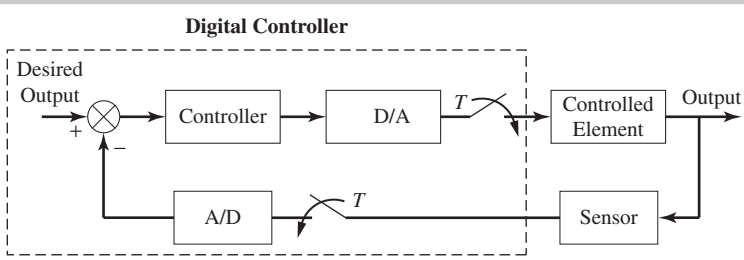


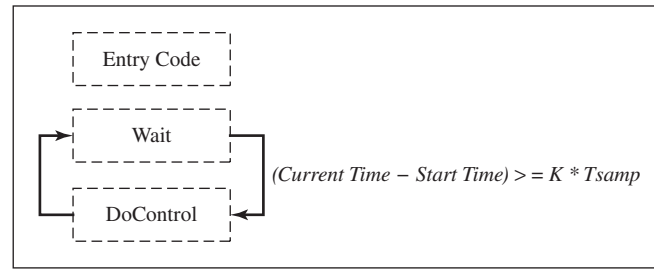
Figure 6.14

Block diagram of a digital controller feedback loop

The task/state control software structure also can be used for feedback control applications. The only thing that we need to have is access to a timer, so we can properly time the control actions. Figure 6.15 is a state-transition diagram that can be used to implement a feedback control loop. This task has only one state. We first initialize the timer, record the starting time, and set the counting variable K to 1 in a section of the code that is only executed once (entry section as discussed in Section 6.6). In the action part of the state, the current time is read in every scan of this state. If the difference between the current time and the start time is a multiple of the sampling interval, then the software calls the *DoControl* function in which the control action is performed. The *DoControl* routine will have the code for the

Figure 6.15

State-transition diagram for a feedback control task



feedback controller used such as a PID controller (see Chapter 9). After the controller completes its job, the counting index *K* is incremented, and the task goes back to wait for elapse of the next sampling interval to perform the control action again. As seen here, the control of a feedback task has an identical structure to a discrete-event control task.

6.5 Task Scanning

The breakup of a task into a number of states gives us a mechanism for easily implementing these state-transition diagrams in software. The mechanism is a **scanning mechanism** which upon calling the task (through a function call for example) goes through the states in that task (i.e., scans the states), identifying and executing the current state. Once the code for current state is executed, the software exits the task. This process is repeated again on the next scan through the task.

6.5.1 Requirements

Figure 6.16

Example of blocking code

```
'Example of blocking code

While (Input1 <= 5)
   Input1 = Read_AD(1)
End While
```

To implement this scanning mechanism in a single-processor system, two important assumptions should be satisfied. The first is that the execution time of the active state should be short. This is to give almost near parallel execution for multiple tasks running cooperatively. With the availability of high-speed processors, this can be accomplished easily as long as the computations done in the active state are limited to a few lines of code. The second condition is that the code in any state should be non-blocking. A **non-blocking code** is one that has a predictable execution time. A **blocking code** includes waiting for user inputs or waiting for a change in signal levels. An example of blocking code in VBE programming language is shown in Figure 6.16.

In this example, the use of the *While* statement halts the execution of the program until the variable *Input1* has a value greater than 5. Since it is not known when the value of *Input1* will be greater than 5, this code is blocking. While waiting for the input to change value, no other code in the program can be executed. This can cause serious problems if we have other tasks that need to be scanned and if these other tasks include time-sensitive actions (such as counting, motion tracking, or screen updates).

Other examples of blocking code include the use of the *scanf()* statement in C-language programs to read user input and the use of the *msgbox* function in VBE programs to display messages to the user. The *scanf()* statement is blocking because it waits for the user to hit the carriage return before the next statement is allowed to execute. The *msgbox* code is blocking because no other code can execute until the user hits the OK on the message box. These two assumptions are needed for implementing control systems based on this model.

The use of non-blocking code for each state coupled with the scanning mechanism discussed gives us a control software structure that can be easily implemented. This control structure is called the **cooperative control mode**. In this mode, all tasks have the same priority, and we cannot preempt the operation of one task and start another. In the cooperative control mode, the tasks to be controlled are placed in a list (see Figure 6.17). Each task in the list is scanned once per scanning cycle in the order of its placement on the list. In each scan, code corresponding to the active state in the scanned task is executed. The execution of the next task on the list cannot be started until the previous task has completed its operation. With the use of fast processors that provide high scanning rates (over 1,000,000 scans per second) and the requirement that no blocking code is used; the tasks operate in nearly parallel fashion in this mode.

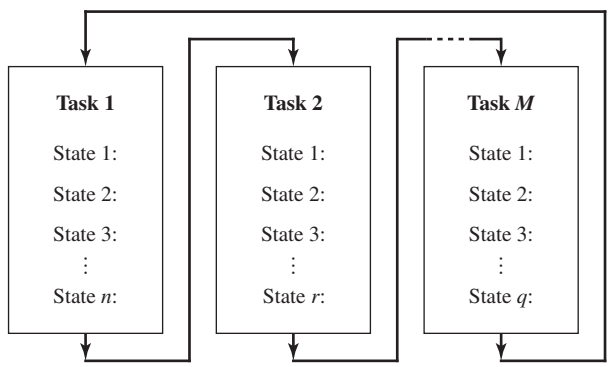


Figure 6.17

Scanning of multiple tasks in a cooperative control mode

One major **advantage of the cooperative control mode** is that data can be easily exchanged between tasks without concern about data corruption for cases where more than one task uses the same data. This is because each task completes its operation before another task starts. A **disadvantage of the cooperative control mode** is that control actions (such as activating a relay or recording the position signal from an encoder) cannot be guaranteed to occur within a particular time interval relative to the event that requires these control actions. This is because we cannot stop the execution of one task and start another. Each task has to wait for its turn to execute. This control mode or scheduling method is sometimes referred to as 'round-robin.' Nevertheless, the simplicity of this control mode make it very attractive to use in many applications.

6.5.2 Implementation

The cooperative control mode can be easily implemented in any programming language, as shown in Figure 6.18. Here we set up an infinite *Do-Loop* that runs

```
While (Stop not equal to 1)
{
    Increment Count
    Call Task1()
    Call Task2()
    Call Task3()
    If (Count mod 1000) = 0 then
        Call function to allows background processing
    Endif
}
```

Figure 6.18

Pseudocode for implementing the cooperative control mode in software

indefinitely while the *Stop* variable is not equal to 1. Within the infinite loop, we include functions to call up the tasks that we want to scan. In the example, we considered three tasks. When a task is called, the code in the active state in that task is executed. Once the code completes execution, the next task on the list is called, and the process repeats.

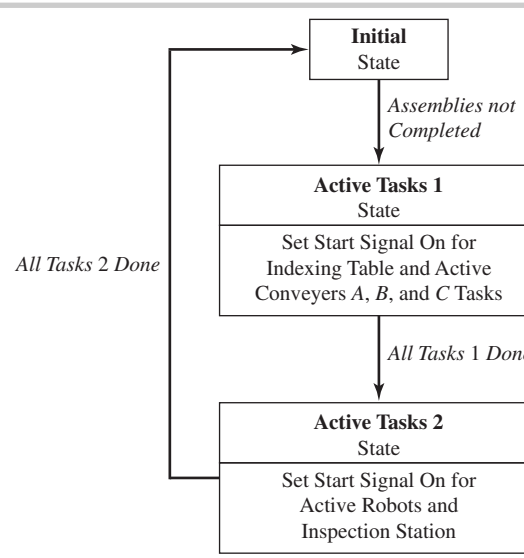
The last three lines inside the loop in Figure 6.18 consist of **code that allows background processing**. This code is necessary if the code is implemented in a Windows operating system to allow Windows to process any background events that have occurred during the execution of this code. These include command buttons that have been clicked, radio buttons that have selected, or other user actions. Without the processing of background events while the infinite loop is active, the control program appears to have hung up, since the program will not respond to user input. Note that how often you perform the check (once every 1000 scans in the example code) depends on how much delay is acceptable in responding to background events. The point here is that you do not need to perform background processing in every scan, as this will slow down the scanning rate.

The **assembly system of Figure 6.12** can be controlled easily using the cooperative control mode discussed here. In this mode, each of the nine tasks in this assembly system is scanned once per cycle (see Figure 6.17), and the active state in each task in that scan is then executed. The scheduling task should be the first task to be scanned, as it determines which task should be active, but the order of scanning of the remaining tasks is not important.

The **scheduling task state-transition diagram** is shown in Figure 6.19. Note that in this assembly system there are two basic operations that are done in each cycle. The first is the rotation of the indexing table and the advancement of the conveyer belts. Once the indexing table and the conveyers have completed their motion, then the second operation of the parts transfer to and from the indexing table and the inspection operation takes place. Similarly, the next motion of the indexing table and the conveyer belts cannot be started until all of the inspection and assembly operations have been completed. This alternating sequence of operation remains until the requested number of assemblies has been made. It should be obvious that the processing time for the longest task in each operation determines the cycle time (and hence the throughput of this system).

Figure 6.19

State-transition diagram for the scheduling task of the assembly system of Figure 6.12



6.6 State Organization

Within a particular state, the code should be organized into one of four different types depending on the actions that need to be done:

Entry Code: Executed only once on entry to the state

Action Code: Executed on every scan of the state

Test Code: Executed to check the condition for transition (to go to another state)

Exit Code: Executed if the associated transition is taken

As an example, consider the *OpenDoor* state in Example 6.1. Refer to Figure 6.20, which gives an implementation of the state code for this state in VBE. For this state, the *Entry Code* section should include the code that causes the door to open (such as actuating the mechanism that opens the door by sending a signal to a relay). Note that this code should be executed once on the first entry to this state. On subsequent entries to this state, the code should not be executed again. This easily can be accomplished in the software through the use of a **flag variable** (*EntryOpenState*) whose value indicates whether the code has been executed or not. There is no *Action Code* here for this state. The *Test Code* for this state is the code that checks if the door is fully open. This is a logical check that returns either a true or a false condition. This code should be executed in every scan of this state. The *Exit Code*, which only gets executed if the test condition is true (i.e., the door is fully open as indicated by the appropriate sensor), should include code that resets the entry flag to its initial value, so that the next time this state becomes active, the code executes in a similar fashion. The exit code should also cause the *DoorIsOpen* state to be scanned in the next scan of this task. This easily can be accomplished using a variable named, for example, *NextState*, whose value indicates which state needs to be scanned in the next scan. The value of the *NextState* variable is only updated when we need to transition from one state to another. On the next scan of the task, the program will go directly to the state that is currently indicated by the *CurrentState* variable, which is assigned the contents of the *NextState* variable. Figure 6.21 shows

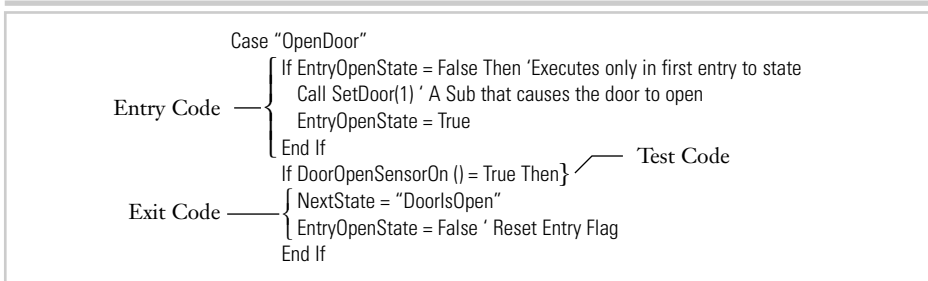


Figure 6.20

State organization for the *OpenDoor* state of Example 6.1

```
Sub DoorTask()
    . . .
    CurrentState = NextState

    Select Case (CurrentState)

        Case "Start"
            . . .
        Case "OpenDoor"
```

Figure 6.21

Code example to update and select active state in a task

how this is done in code using VBE syntax as an example. Note that the use of the *Select . . . Case* statement in VBE or the *Switch . . . Case* statement in C offers an elegant method of selecting the current state in each scan of the task.

For further illustration of state organization, let us consider the coding for the *Wait* state in Example 6.1 (see Figure 6.22). In the *Entry Code* section, the start time is read from the timing function *ReadTimer* and stored in a variable called *StartTime*. Since there are two possible transitions from this state, two test codes are included here. The first *Test Code* checks if the detection sensor returns a true value. In this case, the next state should be the *DoorIsOpen* state. The other *Test Code* checks if the delay interval (5 s here) has elapsed. If that is the case, then the code should go to the *CloseDoor* State in the next scan. Note how the entry flag (*EntryWaitState*) is reset to its initial value in the exit code section.

Figure 6.22

State organization for the *Wait* state of Example 6.1

```
Case "Wait"
Entry Code ──┐ If EntryWaitState =  False Then
             │    StartTime = ReadTimer () ' Record the start time
             │    EntryWaitState = True
             │ End If
Text and     │ If DetectionSensor () = True Then
Exit Code ───┤    NextState = "DoorIsOpen"
             │    EntryWaitState = False ' Reset Entry Flag
             │ End If
             │ If ((ReadTimer () - StartTime) >= 5) Then 'Check if 5s have elapsed
             │    NextState = "CloseDoor"
             │    EntryWaitState = False ' Reset Entry Flag
             └ End If
```

6.7 Control Task Implementation in Software

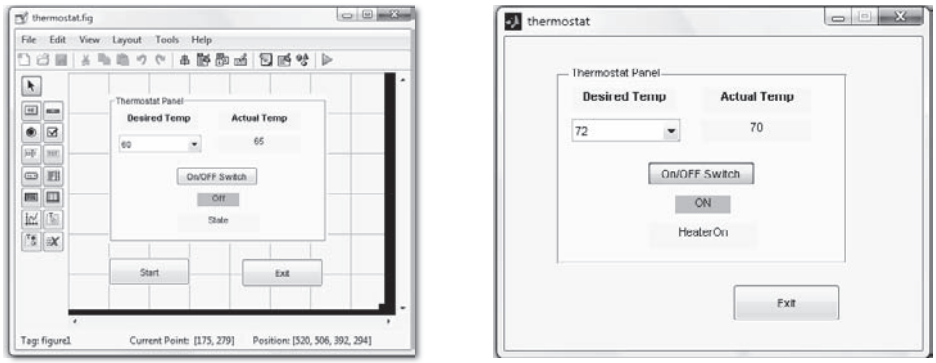
In this section, we will illustrate the software implementation of the heating thermostat state-transition diagram using three computing platforms: MATLAB, VBE, and PIC microcontrollers. The state-transition diagram will be implemented as a single task using the cooperative control mode. One feature of the heating thermostat is that it easily can be simulated in the software without the need for any actual hardware. With proper software structuring, code for simulation easily can be transferred into actual implementation by replacing few functions in the code.

6.7.1 Implementation in MATLAB

MATLAB offers two ways to implement a state-transition diagram. One way is to implement each task as a function that is called by the timer object callback function. The timer callback function offers a built-in mechanism for repeatedly calling one or more software tasks. Coupled with the use of the graphical user interface (GUI) in MATLAB, one can build an application with an interface comparable to that created by VBE or other languages.

The other way to implement a state-transition diagram is to use the **Stateflow** toolbox in conjunction with Simulink. This approach is not covered in this text.

The user interface for the thermostat problem is shown in Figure 6.23(a), and a snapshot of the code while operating is shown in Figure 6.23(b). The user interface uses two *pushbuttons*, one for the *START* command and the other for the *EXIT* command. It also uses a *group panel* on which several controls were placed. These include a *toggle button* for the thermostat on/off switch, a *pop-up menu* for a list of



(a) (b)

Figure 6.23

(a) User interface created using MATLAB GUIDE and (b) snapshot of the interface while code is running

desired temperatures, and *static textboxes* to display the actual temperature and the current state. More detail about creating graphical user interfaces (GUIs) is covered in Section 6.12.

The code works by pressing the *START* push button (which disappears after this action). The callback function (see code listing in Figure 6.24) associated with the *START* button performs initialization as well as setting two timer objects, *TIMER*1 and *TIMER*2. The callback function associated with *TIMER*1 runs every 0.5 s and handles the *Thermostat task* state transition diagram. The callback function associated with *TIMER*2 runs every 5 s and is used for code that simulates heating/cooling action. The *START* callback function also calls the function *TIC*, which will be used later with the function *TOC* to implement the heater delay.

The code for the Thermostat task, which implements the state-transition diagram of Figure 6.11, is shown in Figure 6.25. The code has five states with names

```
% --- Executes on button press in Startbutton.
function Startbutton_Callback(hObject, eventdata, handles)
% hObject    handle to Startbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
tic; % reads the hardware timer
global TIMER1 TIMER2 Temp SwitchOn HeaterOn HeaterDelay
% Perform Initialization
Temp = 65;
SwitchOn = 0;
HeaterOn = 0;
HeaterDelay = 0;
% Read the timer
tic;
%Read the desired temperature
valuepop = get(handles.Desiredpopupmenu,'Value');
stringpop = (get(handles.Desiredpopupmenu,'String'));
DesTemp = str2num(stringpop{valuepop});
%Setup the two timer objects
TIMER1 = timer('TimerFcn',  @timer_callback_fcn1,  'period',  0.5,'ExecutionMode',
'fixedRate');
start(TIMER1);
TIMER2 = timer('TimerFcn',  @timer_callback_fcn2,  'period',  4.0,'ExecutionMode',
'fixedRate');
start(TIMER2);
set(handles.Startbutton,'Visible','Off');
```

Figure 6.24

MATLAB code listing for the *START* button callback function

Figure 6.25

Thermostat task implemented inside the TIMER1 callback function in MATLAB

```
function timer_callback_fcn1(obj, event)
global HANDLESVAR UPDOWN Temp SwitchOn HeaterOn DesTemp HeaterDelay
persistent StartTime DelayIsOver NextState EntryHeaterOnState
if isempty(NextState)
      NextState = 'SwitchOffHeaterOff';
end
if isempty(EntryHeaterOnState)
      EntryHeaterOnState = 0;
end
if (toc - StartTime) >= HeaterDelay
      DelayIsOver = 1;
else
      DelayIsOver = 0;
end
State = NextState;
switch State

      case 'SwitchOffHeaterOff'
            if SwitchOn == 1
                  NextState = 'SwitchOn';
            end

      case 'SwitchOn'
            if Temp <= (DesTemp - 1)
                  NextState = 'HeaterOn';
            elseif Temp > = (DesTemp + 1)
                  NextState = 'HeaterOff';
            elseif SwitchOn == 0
                  NextState = 'SwitchOffHeaterOff';
            end

      case 'HeaterOn'
            if EntryHeaterOnState == 0
                  StartTime = toc;  % Record the start time
                  HeaterOn = 1;   %  Turn On Heater
                  HeaterDelay = 10;
                  DelayIsOver = 0;
                  EntryHeaterOnState = 1;
            end
            if (Temp >= (DesTemp+1)) && DelayIsOver
                  NextState = 'HeaterOff';
                  HeaterOn = 0;
                  EntryHeaterOnState = 0;
            elseif SwitchOn == 0
                  if DelayIsOver
                        HeaterOn = 0;
                        NextState = 'SwitchOffHeaterOff';
                  else
                        NextState = 'SwitchOffHeaterOn';
                  end
                  EntryHeaterOnState = 0;
              end

      case 'HeaterOff'
            if (Temp <= (DesTemp - 1))
                  NextState = 'HeaterOn';
            elseif SwitchOn == 0
                  NextState = 'SwitchOffHeaterOff';
            end

      case 'SwitchOffHeaterOn'
            if DelayIsOver
                  HeaterOn = 0;
                  NextState = 'SwitchOffHeaterOff';
            end
end
set(HANDLESVAR.StateText,'string',State);
set(HANDLESVAR.ActualTempText,'string',int2str(Temp));
```

similar to those used in the state-transition diagram and is implemented using a *switch-case* structure. Note the use of the *NextState* variable for transition between states. Note also the use of the *global* variables which are shared among the functions that make use of these variables. Also note the use of *persistent* declaration for variables that are local to this function but need to keep their values in each call of the function.

The elapse of the timing delay for the heater is implemented by setting the variable *DelayIsOver* to 1 whenever the current time (obtained by reading the *TOC* function) minus the *StartTime* is larger than the *HeaterDelay*. Note that *HeaterDelay* is set to 0 during initialization (see Figure 6.24) and is set to the desired amount in the entry code section of the *HeaterOn* state. Simulation of the heater operation is done by setting the global variable *HeaterOn* to either 1 or 0. This variable is used in the callback function associated with TIMER2 (see Figure 6.26) to update the temperature.

The code for simulating the heating/cooling action is shown in Figure 6.26. A simple scheme is used here in which the temperature is increased by one unit if *HeaterOn* is set to 1 and is decreased by one unit every four calls of the *TIMER2* callback function (controlled by the *count* variable) if *HeaterOn* is set to 0. The decrease in temperature when the heater is OFF is to simulate the heat loss that occurs when the current temperature is higher than the ambient temperature.

```
function timer_callback_fcn2(obj, event)
global HeaterOn Temp
persistent count
if isempty(count)
    count = 0;
end
if HeaterOn
   Temp = Temp + 1;
else
   count = count + 1;
   if count == 4
      Temp = Temp - 1;
      count = 0;
   end
end
```

Figure 6.26

MATLAB code listing for simulating heating/cooling action, which is implemented inside the TIMER2 callback function

It is recommended to dispose the timer objects once the program completes its execution. This is done inside the code of the callback function associated with the *Exit* pushbutton. The code listing is shown in Figure 6.27, and it stops the two timer objects and then deletes them. If the timer objects were not deleted, then on subsequent call of the program, the timing could be OFF.

```
% --- Executes on button press in ExitButton.
function ExitButton_Callback(hObject, eventdata, handles)
% hObject    handle to ExitButton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global TIMER1  TIMER2
stop(TIMER1);
delete(TIMER1);
stop(TIMER2);
delete(TIMER2);
close;
```

Figure 6.27

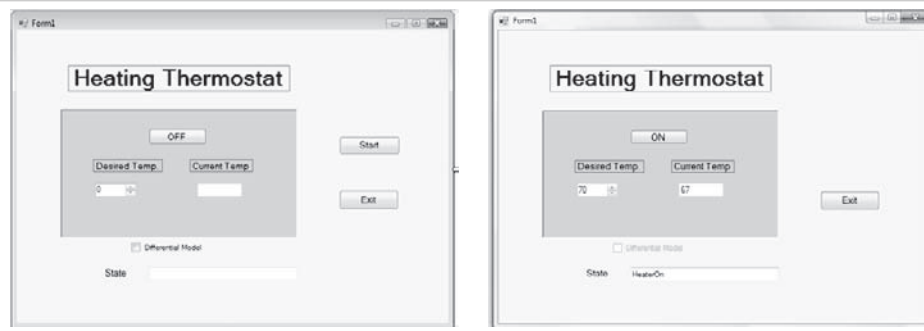
MATLAB code listing for the *Exit* push-button callback function

6.7.2 Implementation in VBE

The **user interface** form for the thermostat task implemented in VBE is shown in Figure 6.28(a). Section 6.12 gives an overview of developing a GUI in VBE, and Appendix A gives an overview of VBE. The interface design is similar to that created in MATLAB. Command buttons are used for the *Start* and *Exit* commands. The remaining controls are placed on a *panel* window. A numeric *updown* control is used to display the list of desired temperatures. The ON/OFF switch is implemented using a *button* that is set to behave like a toggle switch by making the text displayed on the button switches each time the button is pressed. The actual temperature is displayed in a *textbox*. A *checkbox* placed below the panel is used to enable the use of a differential model to simulate the heating/cooling action instead of a simple scheme.

Figure 6.28

(a) User interface created using VBE and (b) snapshot of the interface while code is running



(a) (b)

When the user presses the *Start* button, the *ThermostatTask* function (see Figure 6.29) is called repeatedly inside an infinite loop. The *ThermostatTask* is structured as five states implemented using the *Select Case* structure. At startup, the *NextState* variable is initialized in the *formload* function to *SwitchOffHeaterOff*.

The heater operations are controlled by the functions *TurnHeaterOn()* and *TurnHeaterOff()*, which are used to set the global variable *HeaterOn* to true or false. The timing is obtained using the *ReadTimer()* function, which in this case is used to read the *Timer property* in VBE. Since timer resolution is not an issue in this, the *Timer property is* sufficient to meet the timing needs. If smaller timer resolution is needed, then we just simply replace the code inside the *ReadTimer()* function to make it use a finer resolution timer such as the *Performance Counter* that was discussed in Section 6.3, but none of the other code in this task has to change. Note how the heater shut-off delay is implemented by determining the value of (*ReadTimer() – StartTime*) in every scan of the thermostat task. If the elapsed interval exceeds the *HeaterDelay*, then the *DelayIsOver* variable is set to true. The *HeaterDelay* variable is initially set to zero.

The **heater operation is simulated** by the tick routine of a Timer component. The code listing is shown in Figure 6.30. The execution rate is set by the *Tsamp* property of the component. Two schemes are shown for simulating the heating cooling/operation. In the simple scheme (*DynamicModel* is false), when *HeaterOn* is true, the temperature is incremented by one unit in every call of the tick routine. If *HeaterOn* is false, the temperature is decremented by one unit in every four calls of the tick routine. In another scheme (*DynamicModel* is true and is set by checking the checkbox in the GUI), the temperature is obtained by numerically integrating the energy equation for a thermal system. In using the dynamic model, one needs to

```
Sub ThermostatTask()
  Static Dim EntryHeaterOnState As Boolean
  Static Dim StartTime As Double

  If (ReadTimer() - StartTime) >= HeaterDelay Then
     DelayIsOver = True
  Else
     DelayIsOver = False
  End If

  State = NextState

  Select Case (State)

     Case "SwitchOffHeaterOff"
        If SwitchOn = True Then
           NextState = "SwitchOn"
        End If

     Case "SwitchOn"
        If Temp <= (DesTemp - 1) Then 'And DelayIsOver Then
           NextState = "HeaterOn"
        ElseIf Temp >= (DesTemp + 1) Then ' And DelayIsOver Then
           NextState = "HeaterOff"
        ElseIf SwitchOn = False Then
           NextState = "SwitchOffHeaterOff"
        End If

     Case "HeaterOn"
        If EntryHeaterOnState = False Then
           StartTime = ReadTimer()  '  Record the start time
           Call TurnHeaterOn()          '  Turn On Heater
           HeaterDelay = 20
           DelayIsOver = False
           EntryHeaterOnState = True
        End If
        If (Temp >= (DesTemp + 1)) And DelayIsOver Then
           NextState = "HeaterOff"
           Call TurnHeaterOff()
           EntryHeaterOnState = False
        ElseIf SwitchOn = False Then
           If DelayIsOver Then
              Call TurnHeaterOff()
              NextState = "SwitchOffHeaterOff"
           Else
              NextState = "SwitchOffHeaterOn"
           End If
           EntryHeaterOnState = False
        End If
     Case "HeaterOff"
        If (Temp <= (DesTemp - 1)) Then 'And DelayIsOver Then
           NextState = "HeaterOn"
        ElseIf SwitchOn = False Then
           NextState = "SwitchOffHeaterOff"
        End If

     Case "SwitchOffHeaterOn"
        If DelayIsOver Then
           Call TurnHeaterOff()
           NextState = "SwitchOffHeaterOff"

        End If
  End Select
End Sub
```

Figure 6.29

VBE code listing for the thermostat task

Figure 6.30

VBE code listing for the timer tick routine, which simulates the heater operation

```
Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
Timer1.Tick
    Static count As Short
    Dim heat As Single
    Static TempC As Single
    If DynamicModel = False Then
      If HeaterOn Then
        Temp = Temp + 1
      Else
        count = count + 1
        If count = 4 Then
          Temp = Temp – 1
          count = 0
        End If
      End If
    Else
      If HeaterOn Then
        heat = qheat
      Else
        heat = 0
      End If
      TempC = (Temp – 32) * 5 / 9
      TempC = (1 / RC) * (R * heat – TempC + Ta) + TempC
      Temp = TempC * 9 / 5 + 32
    End If
```

select values for the thermal resistance *R*, the thermal capacitance *C*, and the heater output *qheat*. In the code listing in Figure 6.30, the temperature is converted from Fahrenheit to Celsius (centigrade) before the integration is performed and then converted back to Fahrenheit, since we used SI units for the *R*, *C*, and *qheat* values.

6.7.3 Implementation in a PIC Microcontroller

In the previous two sections, we have illustrated the implementation of the thermostat control problem using a PC computing platform. In this section, we illustrate it using a PIC16F690 microcontroller. While an MCU is normally not used to just implement simulation code, it was done here to illustrate coding using an MCU. One major difference between a PC and an MCU is the lack of a built-in display device on the microcontroller. While we can use a serial interface to display the microcontroller variables on a PC, we will opt here for a solution with no graphical user interface. We will just make use of the interface features available on the low pin-count development board to solve this problem (If the low pin-count development board is not available, then one can use another development board or construct an equivalent setup using a breadboard and equivalent components). The low pin-count board (see Figure 4.11) has a rotary pot that we will use to specify the desired temperature. For the purpose of simplifying the interface, we will make the desired temperature to range from 6 to 10 degrees. Thus, we will map the 10-bit A/D range of the pot into five numbers (i.e., 0 to 204 → 6, 205 to 409 → 7, etc.). For displaying the current temperature, we will use the four LEDS on the board to display the temperature in binary form. With the four LEDS, this will be enough to display a temperature ranging from 0 to 15 degrees. At startup, the current temperature will be set to 2. The built-in NO push-button switch (SW1) on the board will be used as the thermostat switch ON/OFF button with the OFF action indicated with the switch pressed all the time.

```
////////////////////////////////////////////////////////////////////////
///                        Thermostat.c
///
///  This program implements the thermostat state diagram on
///  PIC16F690 using the Low pin count board
///  Compiler: PCWH complier from CCS, Inc.
////////////////////////////////////////////////////////////////////////
#include <16F690.h>            //Include file for the particular chip used
#DEVICE ADC = 10               //10-bit A/D return value
#use delay(internal = 8M)      //Use Internal 8 MHz- clock

#fuses INTRC_IO, NOMCLR, NOWDT, NOPROTECT, NOBROWNOUT

#define SwitchOffHeaterOff 1  //Define the states
#define SwitchOn 2
#define HeaterOn 3
#define HeaterOff 4
#define SwitchOffHeaterOn 5

unsigned int32 Time;           //Variable to record time using Timer1
unsigned int16 LastCount;      //Internal variable used by GetTimeNow()
unsigned int32 Tupdate;        //Update interval for heater simulation
float TimerRes;                //Resolution of Timer1

int8 count = 0;                //Variable used in heater simulation
int8 EntryHeaterOnState = 0;   //HeaterOnState Entry variable
int8 DelayIsOver;              //Variable to indicate heater delay is over
int32 StartTime1, StartTime2;  //Variables used for interval timing
int8 State, NextState;         //State and NextState of transition diagram
int HeaterDelay = 0;           //Heater delay variable with initialization
int8 Temp = 3;                 //Actual temperature
int8 DesTemp;                  //Desired temperature
int8 HeatOn;                   //Variable to indicate heater status

                               //Declaration of functions used in program
void Thermostat_Task(void);    //Thermostat state transition diagram function
void heater(void);             //Function to simulate heater
int32 GetTimeNow(void);        //Returns time in multiple of timer resolution
void SetUpTimer(void);         //Function to setup timer 1
```

Figure 6.31

Variable declaration for the thermostat implementation in C-code on the PIC16F690 microcontroller

Figure 6.31 shows the variable declaration section for the C program written for the thermostat control problem. The code is compiled using the PIC-C compiler. The *#define* statement is used to define each of the five states in the state-transition diagram.

The *main()* routine as well as the routine to simulate the heater operation and the timing functions are shown in Figure 6.32. In the *main()* routine, the function *SetupTimer()* is called to setup Timer1, which is then read using the *GetTimeNow()* function. A/D channel 0, which is connected to the rotary pot, is also setup here. The *main()* routine then enters an infinite loop in which the *Thermostat_Task()* and the *heater()* routines are repeatedly called. The *heater()* routine updates the current temperature every *Tupdate* interval using a simple scheme. If the variable *HeatOn* is set to 1, then the temperature is increased by one unit every *Tupdate* interval. If *HeatOn* is set to zero, then the temperature is decreased by one unit every four *Tupdate* intervals. The *heater()* routine also updates the current temperature output on the microcontroller.

The implementation of the state-transition diagram for the thermostat task is shown in Figure 6.33. The coding is very similar to that used in the MATLAB and VBE versions. The *switch-case* statement is used to implement the state-transition

Figure 6.32

Main, heater, and timing routines for the thermostat implementation on the PIC16F690

```
void main(void)                          //main function
{
  SetUpTimer();                          //Setup timer 1
  setup_adc(ADC_CLOCK_DIV_16);           //Setup A/D
  setup_adc_ports(sAN0);                 //Select channel RA0 for A/D
  set_adc_channel(0);
  Tupdate = 5 * 1000000;                 //Update interval in units of usec
  StartTime1 = GetTimeNow();             //Record initial time
  StartTime2 = GetTimeNow();
  NextState = SwitchOffHeaterOff;
  output_c(temp);                        //Display initial temperature

  while ( 2 > 1) // Start infinite loop
  {
  Thermostat_Task();
  heater();
  }
}
void heater(void)
{
    if ((GetTimeNow() – StartTime2) > =  Tupdate)
    {
      If (HeatOn == 1)
         {
         Temp = Temp + 1;                //Heating action
         }
         else
         {
         count = count + 1;
         if (count  ==  4)
           {
           count = 0;
           Temp = Temp -1;               //Cooling action
           }
         }
      StartTime2 = GetTimeNow();
      output_c(Temp);                    //Display current temperature
    }
}

void SetupTimer(void)
{
setup_timer_1(T1_INTERNAL| T1_DIV_BY_2); //At 8 Mhz internal clock and
                //2 prescale, the timer has a resolution of 1 usec
TimerRes = 0.000001;

LastCount = 0;
Time = 0;
set_timer1(0);                           //Initialize the timer
}

int32 GetTimeNow(void) //Returns time in units of usec
{
unsigned int16 ReadTime;

ReadTime = get_timer1();
if (ReadTime > LastCount)
  Time = (ReadTime – LastCount) + Time;
else
  Time = Time + ((65536 – LastCount) + ReadTime);

LastCount = ReadTime;
return(Time);
}
```

```
void Thermostat_Task(void)
{
  if ((GetTimeNow() – StartTime1) >= HeaterDelay )
    {
     DelayIsOver = 1;
    }
  else
    {
   DelayIsOver = 0;
    }

   DesTemp = 6 + (int8)(read_adc()/204.0);//Read desired temp from pot
                        // DesTemp will range from 6 to 10
  State = NextState;
  switch (State)
    {

   case SwitchOffHeaterOff:

        if (input(PIN_A3) == 1)// Is OnOffSwitch on?
          {
          NextState = SwitchOn;

          }
          break;

    case SwitchOn:
      if (Temp <= (DesTemp – 1))
          NextState = HeaterOn;
      else if (Temp >= (DesTemp + 1))
          NextState = HeaterOff;
      else if (input(PIN_A3) == 0 )
          NextState = SwitchOffHeaterOff;
       break;

   case HeaterOn:
      if (EntryHeaterOnState == 0)
       {
          StartTime1 = GetTimeNow();  // Record the start time
          HeatOn = 1;   //  Turn On Heater
          HeaterDelay = 10 * 1000000;
          DelayIsOver = 0;
          EntryHeaterOnState = 1;
       }

      if ((Temp >= (DesTemp + 1)) && (DelayIsOver == 1))
        {
          NextState = HeaterOff;
          HeatOn = 0;
          EntryHeaterOnState = 0;
        }
      else if (input(PIN_A3) == 0 )
        {
         if (DelayIsOver == 1)
           {
            HeatOn = 0;
            NextState = SwitchOffHeaterOff;
           }
         else
           {
            NextState = SwitchOffHeaterOn;
           }
         EntryHeaterOnState = 0;
```

Figure 6.33

The state-transition diagram for the thermostat task implemented on the PIC16F690

Figure 6.33

The state-transition diagram for the thermostat task implemented on the PIC16F690 (*continued*)

```
            }
            break;

    case HeaterOff:
        if (Temp <= (DesTemp – 1))
            NextState = HeaterOn;
        else if (input(PIN_A3) == 0 )
            NextState = SwitchOffHeaterOff;
        break;

     case SwitchOffHeaterOn:
        if (DelayIsOver == 1)
         {
            HeatOn = 0;
            NextState = SwitchOffHeaterOff;
         }
        break;

      }
}
```

diagram. The delay before turning off the heating action is controlled by the variable *DelayIsOver* which is set to 1 if the time interval since the heater was turned on has elapsed.

Note that the code shown in Figures 6.31 to 6.33 can be changed easily if another microcontroller is used instead of the PIC16F690. These changes include changing the header file (<16F690.h>) to that corresponding to the other chip used. It also involves changing some of the constants that are used to identify parameters of functions that set the timer and the A/D (such as the parameter that selects pin RA0 for A/D).

6.8 MULTITASKING

Real systems have many tasks that need to be controlled at the same time. This brings the issue of how to manage the control of these tasks. The answer to this question depends on the level of control we want over these tasks. Issues that need to be considered include: Do all the control tasks have the same level of priority or do some have higher priority than the rest? Will we have a situation in which we need to preempt the execution of one task and start another task instead? And is it acceptable if there was a delay in starting the execution of a task?

We have already discussed the **cooperative control mode** in the previous section. In the cooperative control mode, all tasks have the same priority, and we cannot preempt the operation of one task and start another. We will discuss the preemptive control mode in this section.

The **preemptive control mode** allows the stopping of one task and the start of execution of another task. This is needed in situations like alarm processing where a certain task needs to run immediately in response to an alarm signal. For this scheme to work, each task needs to have a priority level assigned to it. A task that has a higher priority than a currently executing task can stop the execution of the lower priority task and run instead of it. When the higher priority task completes its execution, the lower priority task will resume its operation. Similarly, if two tasks are ready to run, but one task has a higher priority than the other, then

the preemptive scheduler will allow the higher priority task to run first. The lower priority task will run only after the higher priority task has completed its job.

Preemptive schedulers are much more difficult to implement in software. For example, VBE does not offer built-in code to implement such a scheduler. The closest thing to preemptive scheduling in VBE is the thread class (to be discussed later), but since Windows is a general operating system, the preemption is different from that implemented in a preemptive real-time operating system (discussed in Section 6.11).

It is informative to discuss how an operating system (such as the Windows operating system) handles the operation of a program that it runs. When a program executes, it runs as one or more processes. A **process** is the term that is used by the operating system to designate the address (or memory) space needed to run the program as well as the control information that is used to control the execution of the program. Windows also gives process designation to Windows services (such as Windows time keeping and Windows events logging). Processes do not share memory space. When several programs are run on the same computer, each one is run as a separate process. In the case of a single processor, the processor switches among the processes, giving each one a share of the **Central Processing Unit (CPU)** time. If we have multiple processors, each processor could be assigned to run one process. Each time a processor switches from one process to another, which is a procedure called **context switching**, it needs to save and restore process information as well as to point the processor to a new address location. Context switching can consume a considerable amount of CPU time if there are a large number of processes that are running. Since processes do not share memory space, communication between processes also is more difficult and has to be done through operating system resources (such as pipes and sockets).

To improve on the overhead associated with context switching between processes, a process can be split into several entities of code called **threads**, where each thread has its own control information but shares memory space with the other threads running in the same process (see Figure 6.34 for illustration). A thread can therefore be defined as the entity within the process that is scheduled for execution. A process starts as one thread, but it can add or create more threads as needed. Because of address sharing between threads, it requires less overhead to communicate between threads as well as to switch between one thread and another. Similar to a process, threads share CPU time, and only one thread can be active at a time. However, it is more difficult to write threaded code. In addition, there is the possibility that threads can have a conflict with accessing shared resources (such as memory and open files).

Threads are a key to implement **multitasking programs**. The Windows operating system is an example of a multitasking system, where several threads from

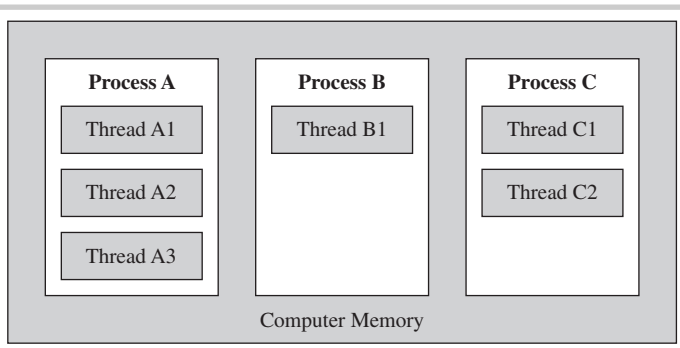


Figure 6.34

Graphical illustration of process and threads

many processes appear to run concurrently on a single processor. The operating system can start, terminate, or suspend the execution of a particular thread. Due to the fast operating speed of modern PCs, these threads appear to execute simultaneously. A typical process (such as Microsoft® Word) is split into over 10 threads, and in a typical PC, hundreds of threads are scheduled for execution. The reader is encouraged to call the Windows Task Manager on a PC and examine the processes and threads that are running.

6.9 Threading in VBE

6.9.1 BackgroundWorker

The most reliable method of implementing multi-threaded applications in VBE is to use the ***BackgroundWorker*** control. This control is included in the *Components* folder in the *Toolbox*. The *System.ComponentModel* namespace should be added to the application when using this control. The *BackgroundWorker* creates a thread that runs in the background in parallel with the application code that runs in the foreground or main thread. Note that in VBE, the foreground and the background threads' use of the computing resources is similar. The difference between the two concerns the existence of the background thread. A foreground thread runs indefinitely, while a background thread terminates once the last foreground thread has stopped. The *BackgroundWorker* thread can be used to run lengthy operations (such as handling file input/output, complex computations, or Internet communication). By placing computationally intensive operations in a separate thread that runs in the background, the main thread can remain responsive to user input and the application interface does not freeze. There is no need to use the **VBE *DoEvents*() method**, which can lead to slow performance if called repeatedly, to handle any pending user interface commands. In addition, **blocking code** in one thread does not stop the execution of code in another thread which has no blocking code. Note that more than one *BackgroundWorker* thread can be set up to run in one application.

To illustrate how to use the *BackgroundWorker* control, let us assume that the lengthy computations that need to be done are placed in a function called *BackgroundFunction*(). This *BackgroundFunction*() should be called from the *BackgroundWorker***_DoWork*() function, which is the VBE specified method of performing the work done by the *BackgroundWorker* thread. The * in the name of the function is an integer number that defines the particular *BackgroundWorker*. An example code is shown in Figure 6.35. Note that the *BackgroundWorker***_DoWork*() function is not explicitly called by another procedure in the application. This function is automatically called as the event handler when the *BackgroundWorker* is started by executing

```
BackgroundWorker*.RunWorkerAsync()
```

Figure 6.35

BackgroundWorker DoWork code listing in VBE

```
Private Sub BackgroundWorker1_DoWork(ByVal sender As System.Object, ByVal e As
System.ComponentModel.DoWorkEventArgs) Handles BackgroundWorker1.DoWork

        BackgroundFunction()

End Sub
```

The foreground thread can cancel the operation of the background worker at any time by executing the following statement, provided that the *WorkerSupportsCancelation* property is set to true:

BackgroundWorker.CancelAsync()*

The *BackgroundWorker*_DoWork()* function can monitor for cancellation by checking the value of the *CancellationPending* proporty. If the foreground thread has requested cancellation of the background thread, then the *BackgroundWorker*.CancellationPending* property will be set to true.

In addition to the *DoWork* event handler function, the application should also include a function to handle the event when the *BackgroundWorker* completes its operation. This event handler is called *BackgroundWorker*_RunWorkerCompleted()* and should include code to handle cases where the background function completes its operation normally, was canceled by the user, or generates an exception during its execution. While it may not needed in every application, the *BackgroundWorker* also supports an event that can inform the foreground thread of the progress of the background operation. This progress reporting is handled by the *BackgroundWorker*_ProgressChanged()* function provided that the *WorkerReportsProgress* property is set to true.

Note that VBE does not allow the *BackgroundWorker* thread (or any created thread in general) to communicate directly with any of the controls on the application form. If the *BackgroundFunction()* happens to update one of the controls, then an *InvalidOperationException* is generated, since **cross-thread calls** are generally not allowed. This exception can be overridden by setting the property *FormName.CheckForIllegalCrossThreadCalls* to false where *FormName* is the name of the form file in the application, but it is not the preferred method. A preferred way of cross-thread communication is to use the ***Invoke* method** which compares the thread ID of the calling thread to the ID of the thread on which the controls were created. If the threads are different, a function should be created to call itself asynchronously using the *Invoke* method to update the control. If the threads are the same, then the control can be updated directly. Figure 6.36 shows the VBE code listing for a function *SetText2* that sends output to a textbox named *TextBox2* using the invoke method.

```
Private Sub SetText2(ByVal [text] As String)

    ' InvokeRequired compares the thread ID of the
    ' calling thread to the thread ID of the creating thread.
    ' If these threads are different, it returns true.
    If Me.TextBox2.InvokeRequired Then
        Dim d As New SetTextCallback(AddressOf SetText2)
        Me.Invoke(d, New Object() {[text]})
    Else
        Me.TextBox2.Text = [text]
    End If
End Sub
```

Figure 6.36

Cross-thread communication using Invoke method in VBE

The *SetTextCallback()* function has to be declared as a *delegate* function as shown below in the file in which the function *SetText2* is defined. The delegate designation enables asynchronous calls for setting the text property on the TextBox control:

```
Delegate Sub SetTextCallback(ByVal [text] As String)
```

6.9.2 Thread Class

In addition to the *BackgroundWorker* control, VBE has a class called **thread** that can be used to implement multithreaded programs. Each task in a program can be assigned to a separate thread, and using the methods provided for the thread class one can start, abort, or sleep (put on hold) a particular thread. **Starting a thread** means that the code associated with the thread will start executing. **Aborting a thread** means that the thread is removed from the list of executing threads. Note that aborting a thread is not guaranteed to abort the thread immediately or even at all. This depends on what the thread is doing when the call is issued. **Sleeping a thread** blocks the execution of the thread for certain amount of time as specified by the integer timeout argument, whose value is in milliseconds (ms), that is passed to it. Note that the *Suspend* method, which stops the execution of a thread, is obsolete and not supported in VBE. In addition, different priorities can be assigned for different threads to manage the execution of the threads.

To use the thread class, the *System.Threading* namespace should be added to the form in which the thread code is written. To assign a task to a thread, the following statement is used:

```
Thread1 = New System.Threading.Thread(AddressOf Task1)
```

where *Task1* is the code associated with *Thread1*. To start the execution of a thread, the following statement is used:

```
Thread1.Start()
```

The user can monitor the execution state of a thread by looking at the ***ThreadState* property**. When a task is assigned to a thread variable, the thread is in the *Unstarted* state. The state changes to *Running* when the thread has started. During the process of aborting a thread, the thread is in the *AbortRequested* State. If the abortion is successful, the state changes to the *Stopped* state. The thread is also in the *Stopped* state if it has completed its work. If the thread is blocked as a result of a sleep request, the state is in the *WaitSleepJoin*. Note that you cannot start a thread that has been aborted or has completed its work. The thread has to be created again (with the *New* keyword) before it can be started.

The thread class allows **five levels of priority for threads**. These priorities from lowest to highest are: *Lowest*, *BelowNormal*, *Normal*, *AboveNormal*, and *Highest*, with *Normal* being the default priority. The priority of a thread is set by changing the *Priority* property of the thread object. Note that the priority property of a stopped thread cannot be changed. Only threads in the *Unstarted* or *Running* state can have their priority changed.

6.10 Resource Sharing

Sharing resources (such as variables and data) is very tricky in multithreaded applications. If not done properly, it can lead to data corruption. To illustrate this, consider the statement listed as

$$x = x + 1$$

This single-line, high-level programming statement increments the value of the variable x by 1. When this statement is compiled, it is translated into several

machine instructions:

- Move x from memory into addition register
- Add 1 to x
- Move the result back to the original memory location of x

In a threaded application on a PC, the runtime language manager can stop the execution of one thread at the end of any machine instruction (and not the end of a VBE statement) and jump to another thread. Now assume we have two threads, the variable x has originally a value of 1, and the first thread just completed the execution of the first instruction when the computational resources were shifted to the second thread. Now assume further that the second thread works with the same shared variable x, and it assigns a value of 10 to x when it runs. Now, when thread 1 resumes its operation, the value of x would be 2 and not 11 after the last two machine instructions were executed. This is because thread 1 executes the last two machine instructions with the original value of x, which was already moved into the addition register. Thus, we have a situation here where the computation done in the first thread nullifies the works done in the second thread. This type of error is called a **race** condition and can be prevented if we design the code to prevent thread switching in the middle of performing a data update operation.

Programming languages such as VBE support several mechanisms for **thread synchronization** and resource sharing. These mechanisms include the *Interlocked* keyword, the *Synclock* keyword, and the *Mutex* and the *Semaphore* classes.

The ***Interlocked*** keyword offers an easy way to prevent thread switching in the middle of certain commonly performed operations (such as incrementing a variable, decrementing a variable, and addition of two variables). The statement $x = x + 1$ can be written as

```
Interlocked.Increment(x)
```

where the increment method of the *Interlocked* keyword is used here. This *Interlocked* statement makes the machine instructions that increments the variable x by 1 to act as one big machine instruction (i.e., interlocked) or as one atomic operation, and thus it prevents thread switching in the middle of execution of this statement.

In many situations, we have computations other than incrementing or decrementing a variable, and thus, we need another mechanism to perform thread synchronization. The ***Synclock*** keyword offers such a mechanism. *Synclock* is a mechanism that can create a critical section of code where only one thread can access that code. It is done through the concept of a lock. If a thread owns a lock on an object, all other threads are prevented from acquiring that lock. The format of the *Synclock* statement is

```
Synclock ObjectVariable
    Critical code here
End Synclock
```

where the critical code is placed between the *Synclock ObjectVariable* and *End Synclock* statements. Now if we have two threads and the first thread got a hold on the *Synclock*, the code in the *Synclock* statement in the second thread is prevented from execution until the first thread releases the lock by executing the *End Synclock* statement. Thus, the second thread is blocked while waiting for the

lock to be available. Obviously, the same *ObjectVariable* name should be used in both *Synclock* statements to achieve the synchronization. There are two limitations about the *Synclock* statement that we should mention. The first is that the lock name (or *ObjectVariable*) has to be an object type and not a simple variable such as integer. Second, if the first thread does not release the lock, then the second thread will wait indefinitely for that lock to be available with no time out possibility.

A more sophisticated method of creating a critical section can be done using a ***Mutex***. A *Mutex* is an abbreviation for mutual exclusion. It is used to give one thread an exclusive use of a shared resource for the case where two or more threads need to share the same resource. When a thread requests access to a *Mutex* that is acquired by another thread, the requesting thread normally will be suspended until the other thread has released the *Mutex* (default method) or until the end of a timeout period if the thread is not available and a timeout period was specified. Furthermore, by passing a name to the constructor when creating a *Mutex* object, the *Mutex* can be used to perform synchronization across different processes.

To use a *Mutex*, first the *Mutex* has to be declared. An example of such a statement is

Dim mutex1 As New Mutex()

To request access to the *Mutex*, the following statement is used:

mutex1.WaitOne()

And to release the *Mutex*, the following method is used:

mutex1.ReleaseMutex()

Note that the *WaitOne()* method is overloaded. When called without a parameter, the *WaitOne()* waits indefinitely until the resource become available. To wait only a limited amount of time, the following form is used:

mutex1.WaitOne(TimeoutPeriod, ExitContext)

where *TimoutPeriod* is the waiting period in ms and *ExitContext* is a Boolean variable that allows the *Mutex* to exit before the wait is over if the resource becomes available (if set to true). With this calling format, the calling thread gives up on waiting on the resource if did not become available by the end of the timeout period.

To illustrate the use of the *Mutex*, consider two threads each performing the same *For-Loop* operation 100 times using the same global variable *count*. In each run through the *For-Loop*, the corresponding thread is called to sleep for 20 msec. The example code is listed here. The * symbol is replaced by 1 or 2 in actual coding.

```
Dim i As Integer
    For i = 1 To 100
                count = count + 1
                thread*.Sleep(20)
    Next
```

If we run the two threads without using any synchronizing mechanism and we displayed the counted variable from each thread when the threads complete their

work, in some cases, the count variable will be less than 200 when both threads complete their work. Due to thread switching, the computation done in one thread nullifies the work done in the other thread if the switching happens in the middle of data increment operation. As explained previously, if the switching happens after the *count* variable was moved into the addition register, then the updated value of *count* in the second thread is lost when the first thread resumes its operation. Now if we use a *Mutex* as shown next, the final tally for the *count* variable is always 200.

```
Dim i As Integer
   For i = 1 To 100
            mutex1.WaitOne()
            count = count + 1
            thread*.Sleep(20)
            mutex1.ReleaseMutex()
   Next
```

Here each thread does not execute the data update operation while the other thread is incrementing the *count* variable, and thus, the work done in one thread is not nullified by the other thread. Since the count variable is incremented 100 times in each thread, the final tally will always be 200.

The last thread synchronization mechanism that we will consider is the ***Semaphore***. While the *Semaphore* can perform the same functions as a *Mutex*, it is fundamentally different. A *Semaphore i*s primarily used to control the number of entries to a shared resource, while a *Mutex* allows only one entry. The *Semaphore* is initially set with the allowed number of entries or count. Each time a thread enters the *Semaphore*, the count on the *Semaphore* is decremented. The count is incremented when the thread releases the *Semaphore*. When the count is zero, all calls to the *Semaphore* are blocked until the other threads release the *Semaphore*. Similar to a *Mutex*, we can declare *Semaphores* that can be used across processes as well with a timeout capability. As an example, let us consider a *Semaphore* initialized with a count value of 1 and with two resources that can use it. The declaration for such a *Semaphore* is given in the following statement.

```
Dim sem1 As New Semaphore(1, 2)
```

Entering the *Semaphore* is done with the following statement:

```
sem1.WaitOne()
```

And exiting the *Semaphore* is done with the following statement:

```
sem1.Release()
```

The above *Semaphore* will do exactly the same function as a *Mutex*. If on the other hand we initialized the *Semaphore* with 2 as the initial count, then the *Semaphore* does not effectively block one thread from executing while the other thread is inside the semaphore, since two threads can enter the *Semaphore* at the same time. However, if we declare the semaphore with 0 as the initial code, then both threads will not execute, since either thread cannot enter the *Semaphore*.

Threaded code is more complex than non-threaded code and is subject to errors that do not occur in non-threaded applications. These errors include races

and deadlocks. **Race** is the situation that occurs when the computation done by one thread nullifies the work done by another thread (discussed before).

Deadlocking is the situation that arises when one is not very careful in accessing shared resources. As an example, consider two threads that share two resources (*A* and *B*). Assume that thread 1 locks resource *A*, while at the same time, thread 2 locks resource *B*. Now if thread 1 requests resource *B*, while thread 2 requests resource *A*, both threads would be blocked from execution. This is because each thread is waiting for the other thread to release the resource it needs before it can continue. Thus, the two threads are in deadlock situation. The only way to prevent such a situation to occur is to request resources in both threads in the same order. This means that thread 1 requests resource *A* and then *B*. Similarly, thread 2 should requests resource *A* and then *B*. Now if resource *A* was not available when it was requested by thread 2, thread 2 waits for this resource to be available. This resource will be made available to thread 2 after thread 1 is done with it, and thus, deadlocking is avoided.

6.11 Real-Time Operating Systems

For many real-world applications, the cooperative control mode is not sufficient to meet the strict timing requirements for many of these applications. For some applications, it is not acceptable to just have the correct computational value produced by a control program, but it has to be produced at a particular time or within a specified time interval or the value is worthless. Examples of these applications include weapon delivery systems, space navigation, automotive safety systems, and high-speed motion control applications. These applications require the resources available from a priority-based preemptive **real-time operating system (RTOS)**. Note that not all RTOSs use a preemptive scheduling method.

The heart of an RTOS is the **kernel** which is the component of the operating system that provides most of the basic services for the application programs that run on the system. These basic services include:

- Task management
- Timing and timers
- Intertask communication
- Dynamic memory allocation
- I/O device support

Task management service is concerned with starting and stopping the different control tasks as well as assigning priorities to tasks, while timing and timers service is concerned with providing means to time events and to include time delays. The intertask communication service provides means for tasks to pass information from one task to another without worrying about data corruption or interference that can result from two separate tasks trying to access the same data. Dynamic memory allocation service is concerned with the ability of a task to create and use a certain amount of memory while the task is executing and releasing that memory when it is no longer needed. Finally, the I/O device support service provides a uniform method to access the many hardware devices that the application needs to communicate with.

Real-Time operating systems can be classified into soft and hard real-time systems [22]. In a **soft real-time system**, there is no guarantee that the system can handle the response to an event within a specified interval, while a **hard real-time system** can do that. A hard real-time operating system should support the following features.

1. Be multithreaded and preemptible
2. Supports different priority levels
3. Have predictable thread synchronization mechanisms such as mutexes and semaphores
4. Have known and deterministic performance and timing parameters
5. Supports priority inheritance

While the Windows operating system was not designed for time-critical control applications, it still satisfies the first three features listed. Windows is different from a dedicated RTOS in the fourth and fifth feature. Unlike a Windows operating system or any other general computing operating system, the services performed by a hard RTOS have a predictable and a fixed execution time and are not dependent on how many applications are running on the processor. There are no random delays that could affect the responsiveness of a hard RTOS. As such, a hard RTOS has a deterministic and very fast response to external events. The timing response of a hard RTOS is the same whether two or ten tasks are running in the system, which is not the case of a soft RTOS.

The **priority inheritance problem** is illustrated in Figure 6.37. Here we have three threads T1, T2, and T3, where T1 has the highest priority and T3 has the lowest priority. Assume that after T3 has locked a shared resource, the Thread T1 starts executing. Because the shared resource was already locked by thread T3, thread T1 will be suspended, waiting for that resource to be available. If only the T1 and T3 threads are running, this would not be a problem. When T3 completes its work, the locked resource would be released, at which point thread T1 would continue. This delay in execution of the high-priority thread is deterministic. The problem happens if we have a thread T2 with a priority in between T1 and T3. If T2 happens to run after T1 was suspended, then T3 is prevented from running until T2 has completed its operation. Since the running time of T2 is not known, we cannot determine when T1 would be able to run again. This situation can be prevented by having an RTOS that supports priority inheritance. In this case, the RTOS will boost the priority of the lowest-priority thread above the middle one

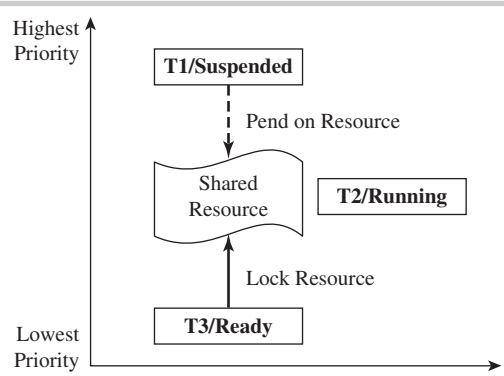


Figure 6.37

Illustration of priority inversion

and up to the suspended thread, causing T3 to run, which releases the locked resource so T1 can run.

From this, we conclude that a general computing operating system like Windows is sufficient for a soft real-time system application, but a preemptive hard RTOS is needed for a time-critical application. If one is interested in using a **PC platform in a time-critical application**, then one needs to use a hard RTOS that can operate on a PC platform. One such operating system is VxWorks® from Wind River Systems of Alameda, CA. This operating system has an extension that can work on PC platforms.

There are many commercial RTOSs available for embedded processors. Examples include VxWorks form Wind River Systems and ThreadX® from Express Logic. We will discuss two RTOS operating systems for microcontrollers in this section. The first is the PIC-C RTOS for PIC microcontrollers. The other one is the ThreadX RTOS.

6.11.1 PIC-C RTOS System

The C-compiler for PIC microcontrollers from CCS, Inc. supports a cooperative scheduling RTOS. The RTOS can be used to schedule tasks to run at specified time intervals, but tasks cannot have a priority level assigned to them nor preempt each other. To start the operation of the RTOS, the *rtos_run()* function needs to be called. The operation of the RTOS is terminated by calling the *rtos_terminate()* function. The code structure for using the RTOS is shown in Figure 6.38.

Since the tasks scheduled by the CSS RTOS are time based, the *#use rtos* directive needs to be called to specify the timer used for time keeping as well as the timer

Figure 6.38

Code structure for implementing a RTOS in PIC-C

```
////////////////////////////////////////////////////////////////////
///    Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////
...
//Define timer to use for RTOS
#use rtos(timer = 0,minor_cycle = 100ms)

//Define first task
#task(rate = 1000ms,max = 100ms)
void first_task ( )
{
  printf("In task 1\n\r");
  ..
}
//Define second task
#task(rate = 500ms,max = 100ms,queue = 2)
void second_task ( )
{
  printf("In task 2\n\r");
  ..
}
//Define third task
...
void main()
{
//Call RTOS to run
rtos_run();
...
}
```

minor cycle, which is the interval at which the RTOS will be called to run the pending tasks. In the code structure shown in Figure 6.38, Timer 0 is used for time keeping, and the minor cycle interval is set to 100 ms. Each of the tasks that need to be run by the RTOS need to have the *#task* directive located above it to inform the compiler that the following function is an RTOS function. There is no restriction on the code listing for a task. In the *#task* directive, the timing rate at which the task should run is specified as well as the maximum budgeted time for the task when it is run. The task budgeted time should be less than or equal to the RTOS timer minor cycle.

We can also specify in the *#task* directive a **queue size** in bytes to be used for messaging between RTOS tasks, as shown in the declaration for the second task. A task can send a **message** to another task using the *rtos_msg_send()* function, which takes two arguments: the task name specified in the first argument and the value of the variable specified in the second argument. Similarly, a message is read using the *rtos_msg_read()* function, which returns the next byte of data available in the task's message queue. The *rtos_msg_poll()* function, which returns true if there is data in the task's message queue, should be called before calling the *rtos_msg_read()* function to ensure that there is data to read, since the *rtos_msgs_read()* function is blocking if there is no data to read. An executing task can yield to another task by calling the function *rtos_yield()*. This causes the task to break out at a given point and return to the same point the next time the task is executed.

To prevent conflicts arising from tasks trying to access a shared resource, tasks can use the *rtos_wait()* and *rtos_signal()* functions to implement a **semaphore mechanism**. Calling the *rtos_wait(sem)* function will decrement the semaphore variable *sem* by 1. Similarly, calling *rtos_signal(sem)* will cause the semaphore variable *sem* to be incremented by 1. When *sem* is equal to 0, another task cannot execute the code included between *rtos_wait(sem)* and *rtos_signal(sem)* until the other task has released the resource by executing *rtos_signal (sem)* function. Figure 6.39 shows a code listing that illustrates the use of the semaphore mechanism to control access to an LED. A different LED is used in each task so the program operation can be easily observed. The code is run on Microchip PIC18 Explorer Board (see Fig. 10.29), which uses the PIC18F8722 MCU. The RTOS code was not able to be compiled for the PIC16F690 MCU.

In the code listing shown in Figure 6.39, the first task flashes LED1 four times every two seconds, while the second task flashes LED2 one time every one second. Both tasks use the same semaphore when accessing the LEDs. When the program runs with the *rtos_yield()* statement in the first task commented out, LED2 will flash once, then after a delay of one second, LED1 will flash four times immediately followed by LED2 flashing once, and the process repeats. The flashing of the LEDs in this case is set by the execution rate for each task. Now if we run the program with the *rtos_yield()* statement in place, then LED2 will flash one time, then LED1 will flash a total of four times (once every two seconds) followed by LED2 flashing once, and the process repeats. Note here that while the second task is designed to run every second, it is prevented from flashing LED2 while the first task is running. This is because the first task calls the *rtos_yield()* function in each run through the *for loop* in that task, so the release of the semaphore (by calling *rtos_signal(sem)*) is done only after the first task is called four times (or the *for loop* in that task is completed).

6.11.2 ThreadX

ThreadX (developed by Express Logic, San Diego, CA) is a hard RTOS designed for embedded applications running on microcontrollers including PIC microcontrollers and others. The software supports preemptive scheduling. We have selected

Figure 6.39

C-Program to illustrate semaphore mechanism using the CCS compiler

```
////////////////////////////////////////////////////////////////////////////////////////////
//////    Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////////////////
#include <18F8722.h>
#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=10M)
#use rtos(timer=0, minor_cycle= 50ms)
#define LED1 PIN_D0
#define LED2 PIN_D1
int8 sem;     //Semaphore variable to control access
//Define first task
#task(rate=2000ms,max=50ms)
void first_task ( )
{  int8 i;
   rtos_wait(sem);
   for (i=0; i < 4; i++)   {
      output_high(LED1); delay_ms(5); output_low(LED1);
      rtos_yield();
      }
   rtos_signal(sem);}

//Define second task
#task(rate=1000ms,max=20ms)
void second_task ( )
{
   rtos_wait(sem);
   output_high(LED2); delay_ms(5); output_low(LED2);
   rtos_signal(sem);
}

void main()
{
      sem=1;
      rtos_run();
}
```

ThreadX because a demo version of the software can be downloaded freely, and it can be used to illustrate the detailed operation of an RTOS. The reader can get further information about ThreadX from the company website (www.rtos.com) or from [23], so we want to focus here on just two aspects of ThreadX. The first is the execution flow of threads in ThreadX, and the second is the priority mechanisms that are implemented.

Figure 6.40 shows the **state-transition diagram** for thread operation in ThreadX, which shares some similarities with thread operation in MS Windows. After a thread is created, it can be either in the *Ready* state (auto start) or in the *Suspended* state (manual start). ThreadX selects the highest-priority thread from all of the ready threads and changes its state to the *Executing* state. Obviously, because of the use of a single processor, only one thread can be in the *Executing* state. While executing, several things can happen. If the thread completes its work, its state changes to the *Completed* state. If the thread was suspended (for example, due to an unavailable resource that it needs to use or because a higher-priority thread needs to run), then it moves to the *Suspended* state. If the cause of suspension was no longer present, then the thread moves to the *Ready* state. A thread ends up in the *Terminated* state if the thread was terminated in either the *Ready*, *Suspended*, or *Executing* state.

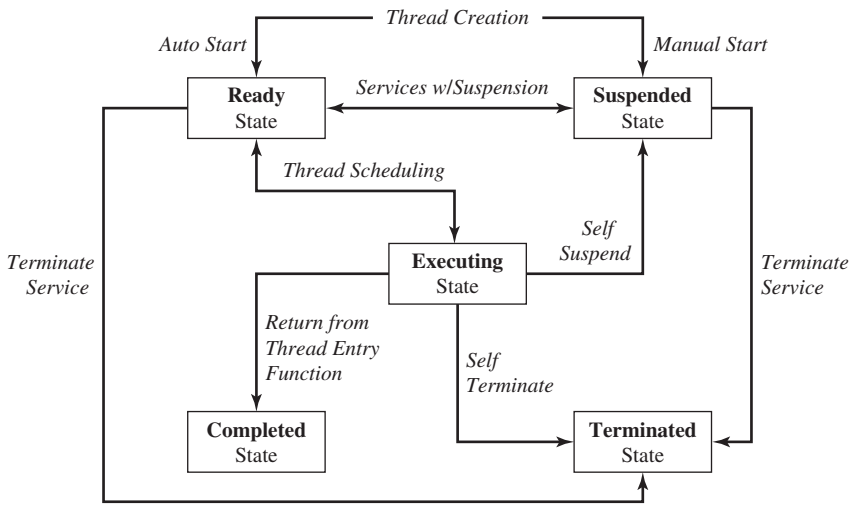


Figure 6.40

State-transition diagram for thread operation in *ThreadX*

ThreadX has as a default 32 levels (can be expanded up to 1024 levels) of priority for each thread, ranging from 0 (highest) to 31 (lowest). If multiple threads of the same priority are ready for execution, they are executed in a first input–first output (FIFO) fashion. ThreadX uses two methods, **time slicing** and **voluntary relinquishing**, to schedule threads that have the same priority. In time slicing, each thread is given a certain number of ticks (i.e., a time slice). When the thread's time slice expires, all other ready threads of the same priority are given a chance to execute before the time-sliced thread is executed again. In voluntary relinquishing, the thread issues a call to voluntary relinquish resources, which causes the thread to stop executing (similar to calling *rtos_yield()* in PIC-C compiler).

As mentioned before, ThreadX has a preemptive scheduler. Preemption is the process of stopping the execution of a thread of a lower priority and the executing of a thread that has a higher priority. ThreadX implements preemption threshold which allows the selection of the priority of threads (ceiling) that can preempt other threads (with priority higher than ceiling).

6.12 Graphical User Interface

Control or measurement programs should have means to interact with the user. This should serve two purposes. The first is to provide feedback information for monitoring the operation of the program. This information typically includes the current actual value of the controlled or measured variable(s) (such as temperature or speed), the desired value of the controlled variable(s), and/or the value of the control input. The second purpose is to let the user have the capability to start/stop the control or measurement action and to modify the desired value of the controlled variable(s), the control gains, or the measurement setting parameters. An **operator interface** is the mechanism that can perform this function. While an operator interface can be built using hardware elements (such as switches, control knobs, and display devices), advances in PC technology and software allow one to create a sophisticated user interface in software. This section focuses on developing graphical user interfaces using MATLAB and VBE.

A graphical user interface (GUI) can be built for a control or measurement program running on a PC or on a microcontroller. Due to the lack of built-in display devices in microcontroller-based systems, a graphical user interface for a microcontroller-based system needs to communicate with the microcontroller through either a serial or USB interface. Thus, designing a user interface for such a configuration requires attention to the data communication and transmission aspects.

Operator interface operations are quite important in many industries such as oil and gas processing, waste water treatment, and electric power distribution, and many vendors provide custom-made programs to handle these operations. These programs are commonly known as **HMI**, which stands for Human Machine Interface. Many of these HMI programs come with a set of graphical tools to represent control devices (such as knobs and switches) and control system components (such as tanks, pumps, and motors).

6.12.1 MATLAB Graphical User Interface

MATLAB offers a tool to build a graphical user interface or GUI. The graphical user interface tool in MATLAB is called *GUIDE* or GUI development environment. This section gives only a brief overview of *GUIDE*. For more detailed information, see MATLAB documentation or one of the many texts available on this topic, such as [24].

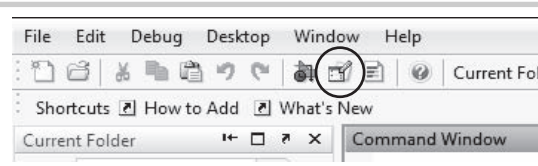
The process of building a GUI in MATLAB is as follows.

- The user selects controls from a palette to design the layout of the user interface in a figure file. The user customizes these controls to make them display the needed information by changing some of their properties.
- The user saves the figure file. This causes MATLAB to process the layout information and to generate an m-file with the same name as the figure file that has a skeleton code for the controls that the designer has selected.
- The designer modifies the skeleton code in the m-file so the GUI performs the intended action.
- Once the user completes the editing of the m-file, the interface operates as intended.

To start building a user interface, the designer needs to type the word *GUIDE* in the command window or to click on the *GUIDE* icon circled in Figure 6.41.

Figure 6.41

GUIDE icon



This will bring the *GUIDE Quick Start* form shown in Figure 6.42 where the user can either open an existing graphical user interface (GUI) or create a new one.

Selecting a new blank GUI displays the form shown in Figure 6.43 which is called the *Layout Editor*. This is the design form where the user can select controls or objects from the left menu and drop them into the design area (gray area with gridlines) on the form. The top of the form has a toolbar that contains a number of

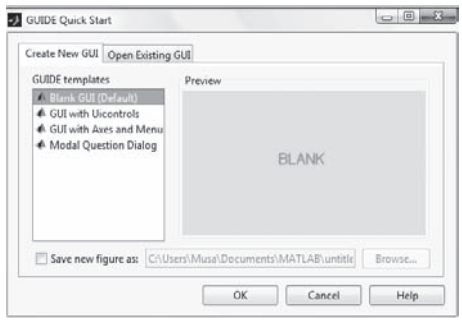


Figure 6.42

GUIDE Quick Start form

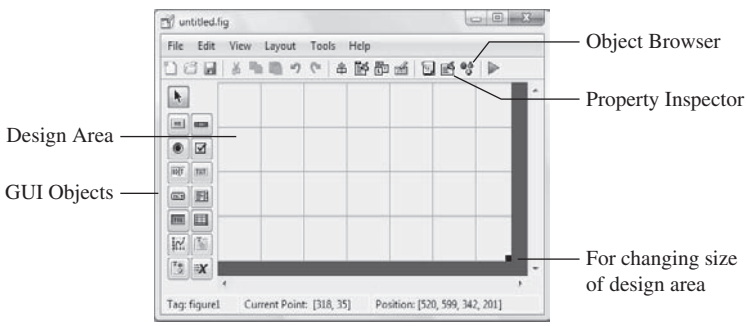


Figure 6.43

Layout Editor form

tools including an object browser, and a property inspector. The user can change the size of the form by dragging the lower-right corner of the design area.

As an illustration, we will create a simple GUI with two objects: a push button and a static text. Selecting these objects and dropping them onto the form and enlarging them, we obtain the form shown in Figure 6.44.

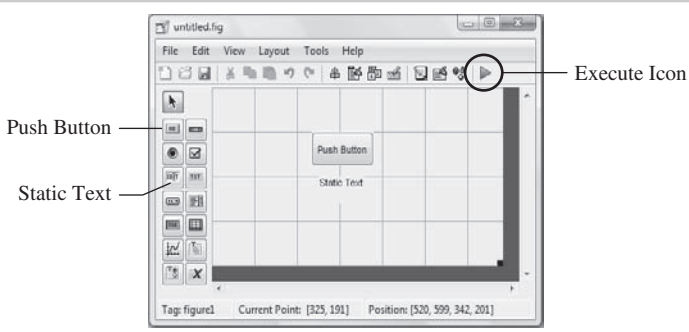


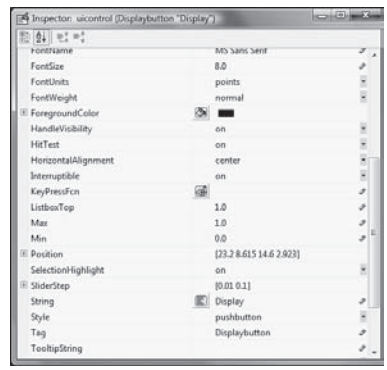
Figure 6.44

GUI with two objects

Any object placed on the form has **properties** such as background color, font size, text displayed, and tag information that can be customized. For example, we can change the text displayed on the push button by changing the *string* property from the *property inspector* menu (see Figure 6.45) that is displayed when we right click on the object in question. The *property inspector* menu also can be directly accessed from the toolbar of the *Layout Editor* form (see Figure 6.43). For this example, we will change the text displayed from *Push Button* to *Display*. The ***tag*** property is the name that MATLAB uses to refer to the object in the m-file associated with the GUI. The tag should be changed from the default label selected by MATLAB to make it easy to distinguish the object in the case where several objects

Figure 6.45

Portion of the property inspector menu for the push button



of the same type are used. For this example, we will change the tag for the push button from *pushbutton1* to *DisplayButton*.

We will save the created GUI with the name *SimpleGUI*. When we save the figure file, MATLAB will save it with the *.fig* extension and will create an m-file with the same name. The generated m-file has a number of functions in skeleton form to process user interface actions. To see a list of these functions, click on the icon circled in Figure 6.46 in the generated m-file.

Figure 6.46

A list of the functions created in the m-file associated with the GUI

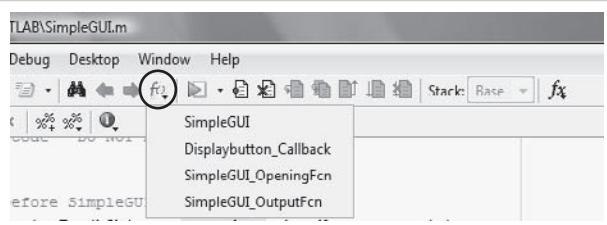


Figure 6.46 shows that four functions were created. The *SimpleGUI* function, where *SimpleGUI* is the name we used when the GUI was saved, is the main function in the GUI. This function sets up a data structure that contains control information plus function handles for some of the subfunctions. Normally, the user does not need to edit this function. The *Displaybutton_Callback* function is the function that processes the event of clicking on the push button. We need to add code to this function to get any action from clicking on the button. The *SimpleGUI_OpeningFcn* is the function that is automatically executed when the GUI is called. This function executes before the GUI is made visible, and we can use this function to add any initialization code that is needed. The *SimpleGUI_OutputFcn* sends outputs from this function to the command line. If our GUI had more objects, then the m-file will have additional callback functions to handle the events associated with these objects. Note that the static textbox object that we used in *SimpleGUI* example does not respond to a mouse click, and thus, MATLAB did not provide a callback function for it. The static text box has other events (such as *ButtonDown* event created by pressing down on key while the control is in focus), but unless we explicitly called for them, MATLAB will not provide code to process them.

In the *SimpleGUI* example, we would like to display the text 'Simple GUI' in the static textbox when the user clicks on the push button. To do this, we need to modify the callback function associated with the push button. Figure 6.47 shows the skeleton code created by MATLAB for the *Displaybutton_Callback* function. The

```
% --- Executes on button press in Displaybutton.
function Displaybutton_Callback(hObject, eventdata, handles)
% hObject    handle to Displaybutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Figure 6.47

Callback function for the *DisplayButton*

variable '*handles*' in the argument list is a data structure that contains a reference to all the objects in the GUI. Any object in the GUI can be referred to using syntax *handles.tag_name*, where *tag_name* is the tag property of the object in question. For example, to refer to the static textbox we created, we simply use *handles.text1*.

Thus to display the message, we need to add the following code to the above callback function:

```
set(handles.text1,'string','Simple GUI');
```

The *set* function is used here to modify the string property of the static textbox so it can display the 'Simple GUI' text. A static text box also can be used to display the value of integer or real variables, but the value needs to be converted to a string first using *int2str()* or *num2str()* functions.

Now, the *SimpleGUI* interface is ready to run. We can type *SimpleGUI* in the command window, or we can click on the *Execute* arrow button (see Figure 6.44) to run the GUI. Doing this, we get the interface shown in Figure 6.48(a). If we click on the *Display* button, the message 'Simple GUI' is displayed in the static textbox, and the interface looks like that shown in Figure 6.48(b).

(a) (b)

Figure 6.48

(a) Interface in operation and (b) after pushbutton was pressed

Realistic GUIs are more complicated than the simple interface illustrated, but the process of creating one is the same. Depending on the type of the object we have in the GUI, we need to add code in the callback function(s) to handle the event(s) associated with that object. We have shown in Section 6.7.1 an interface for simulating a heating thermostat. The thermostat GUI used a popup menu object, two push buttons, four static text boxes, and a panel control. For example, for the popup menu, we used the code shown in Figure 6.49 to get the desired temperature.

One important issue to consider in building a GUI is the **scope and sharing of variables in the GUI file**. In the thermostat example, we had used the *global* declaration to share user defined variables among several callback functions. User-defined variables also can be made accessible among the different functions in the file by adding them to the *handles* data structure (see documentation on *GUIDE* in MATLAB). We also used the *persistent* declaration for local variables that only are used in

Figure 6.49

MATLAB code for handling the popup menu in the thermostat example

```
% --- Executes on selection change in Desiredpopupmenu.
function Desiredpopupmenu_Callback(hObject, eventdata, handles)
% hObject    handle to Desiredpopupmenu (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns Desiredpopupmenu contents as cell array
%        contents{get(hObject,'Value')} returns selected item from Desiredpopupmenu
global DesTemp
valuepop = get(handles.Desiredpopupmenu,'Value');
stringpop = (get(handles.Desiredpopupmenu,'String'));
DesTemp = str2num(stringpop{valuepop});
```

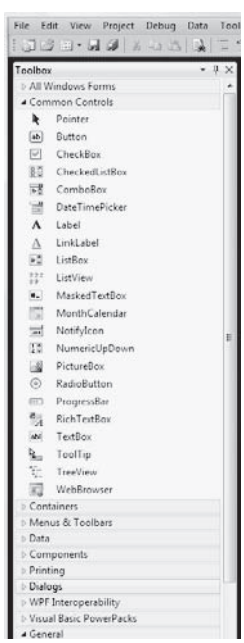
one function, but they need to keep up their values in repeated calls of that function. If we need to access any of the objects in the GUI in a user-defined function (i.e., not in one of the GUI objects' callback functions) such as the *TIMER1* object callback function in the thermostat example, then we need to save the *handles* variable in a global variable (such as the *HANDLESVAR* variable in the listing shown in Figure 6.25) that is shared by the user function that needs to use it.

6.12.2 VBE Graphical User Interface

VBE is a high-level programming language that is primarily used to produce Windows-type applications although console-type applications (with MS-DOS type interface) also can be produced. Since Windows applications are event driven and have inherently a graphical user interface, VBE was thus designed to offer the user an easy way of creating friendly and powerful graphical user interfaces, which are needed in software-based instrumentation, measurement, or control applications. Note that VBE is used by a large number of programmers worldwide to produce professional code for a multitude of applications. VBE is also available to be downloaded free of charge, so users will not incur additional costs or restrictions in using the language. Moreover, the executable version of the code can run on its own (no need to have VBE installed) which is not the case with MATLAB. Appendix A gives a detailed overview of VBE, including variables, operators, functions, sub-procedures, looping and conditional statements, classes, and error handling. For further reading, see [25-26].

Figure 6.50

Controls available in VBE 2010



The process of creating a GUI in VBE is somewhat similar to that to in MATLAB, although the terminology used is different. To develop code in VBE, first you need to create a project that consists of a set of files grouped together. In VBE, you can create different types of projects, with the most common being the *Windows Forms Application* and *Console Application*. To develop a *Windows Forms Application*, the user places controls on a Windows form and then writes code to manage events from these controls in another file. VBE offers a diverse set of controls that are grouped in different categories. The controls are placed in the *ToolBox* window in VBE. Figure 6.50 shows the controls available in the *Common Controls* category. These include *Button*, *CheckBox*, *PictureBox*, *ProgressBar*, and *TextBox*.

Similar to what we did in the previous section on the MATLAB GUI, we will illustrate in this section the development of a simple GUI in VBE. Our simple GUI will utilize a button and a textbox. Selecting these controls and dropping them on the design form that is created when we select a *Windows Form Application*, we get the form shown in Figure 6.51.

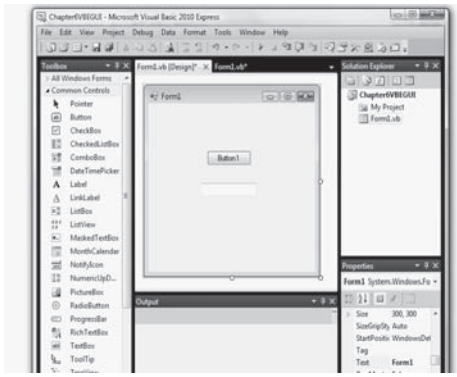


Figure 6.51

VBE form with button and textbox controls

We will modify the *Text* property of the button control to display 'Run' instead of the default value 'Button1'. We will also change the *Name* property (similar to the *tag* property in MATLAB) of the button from 'Button1' to 'cmdRun.' The properties window is normally displayed in the lower-right corner of the VBE integrated development environment (see Figure 6.51) but also can be accessed by clicking the right mouse button.

Now if we double-click on the button control, the design form will disappear, and we will get a window named *Form1.vb*, as shown in Figure 6.52 where the code is added. The design window has a function called *cmdRun_Click* to handle the event of clicking on the button. Similar to a MATLAB callback function, this event-handling function is in skeleton form and does not do any useful action unless we add code to it. In addition to mouse clicking, the button control has many events associated with it (such as mouse hovering over the control, mouse entering the control, pressing and releasing a key while the control is in focus, etc.), and a similar event-handling function can be created for any of these events. For example, to create a handling function for mouse hovering, we locate this event in the property/events window for the control and then we double click on it. VBE will automatically create the event-handling function for that particular event.

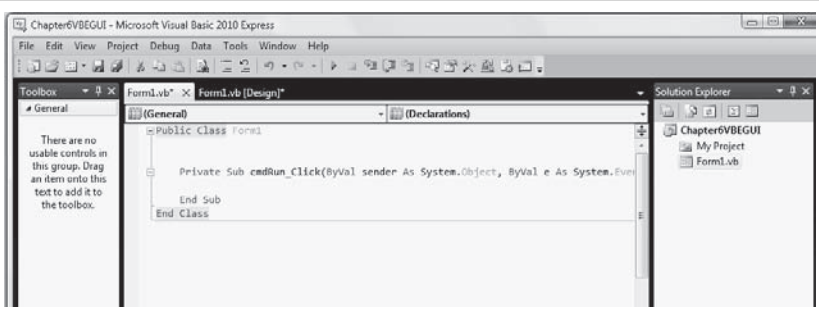


Figure 6.52

Form1.vb code listing

We would like our simple GUI to display in the textbox the number of times we have clicked on the *Run* button. We thus need to create a variable called *count* to keep track of the number of clicks. This variable should not lose the count value on repeated clicks of the *Run* button that would occur if it was declared as a local variable inside the event-handling function. Thus, the variable has to be either declared outside of the event-handling function or declared as a *static* variable

Figure 6.53

Code added to *cmdRun_Click* function

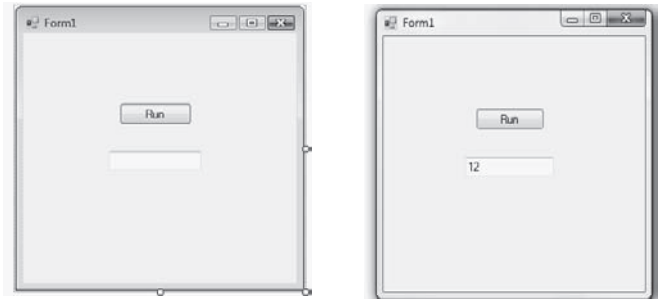
```
Static count As Integer = 0
count = count + 1
TextBox1.Text = count
```

inside the event-handling function to retain its value (see material on variable scope in Appendix A). Choosing the *static* declaration method, we will add the code shown in Figure 6.53. Note how the value of the count variable is displayed by setting the text property of the textbox (accessed by using the dot operator).

Now our simple VBE application is ready to run. We will run the program from the IDE by selecting *Start Debugging* from the *Debug* menu. We will get the form shown in Figure 6.54(a). After we have clicked on the *Run* button several times, the form will look as shown in Figure 6.54(b), where the number 12 in the textbox is the number of times we had clicked on the *Run* button.

Figure 6.54

(a) Program at start and (b) after the *Run* button was clicked several times



(a) (b)

This example shows that the process of creating a GUI in MATLAB or VBE is similar, but there are some differences:

- Since VBE was specifically designed for ease of use graphical interface input, each control in VBE has many more events associated with it than in MATLAB.
- Any user-defined function in the design form in VBE can access any of the controls on the form. This is not the case in MATLAB unless the *handles* variable is made accessible to the user-defined function.

In Section 6.7.2, we developed a VBE GUI and program for the thermostat control problem that looks and behaves similar to the MATLAB created one. The interface used a panel window with a number of controls placed on it. To simulate the operation of the thermostat, we implemented code beyond just handling the events from clicking on the controls. As discussed in Section 6.7.2, we implemented the state-transition diagram as a software task that is run inside an infinite loop that is called when the user presses on the *Start* button. Thus, the thermostat example shows the integration between the user interface and the software structure for a discrete event system.

As an added example, we will discuss the design of an interface in VBE for controlling the speed of a DC motor. The purpose is to discuss some of the issues that arise in using an operator interface. The DC-motor control operator interface will consist of two control buttons, one to start and one to stop the control action. It also has three text boxes to display information about the control system: one box to display the current speed, the second box to display the current control input, and the third box to display the elapsed time since the control action has started. The user input consists of the desired speed and the control gains of a PI-controller (see Chapter 9).

Figure 6.55 shows the form layout for this operator interface. For readability, the form is divided into three groups using the *GroupBox* control in VBE, which displays a frame around a set of controls. These are the *User Input*, the *Monitored Signals*, and the *Control Actions*. For this program to do any useful action, we need

Figure 6.55

Form layout for operator interface

to add additional code including code to either simulate the motor or interface to a real one. The motor dynamics can be simulated by a simple first-order dynamic model that is numerically integrated using the Euler method (see Section 10.3.8), or an actual motor can be used by adding code to interface with the motor/tachometer system through a D/A and an A/D convertors.

One issue in Windows interfaces is that the user can click on any of the buttons or access any of the controls at any point in time. When textboxes are used for user input, as shown in Figure 6.55, problems could happen when the program attempts to read the values in the textboxes while the user is changing these values. These problems occur due to having incorrect or incomplete values in the textboxes while the user is changing the values. To **avoid passing incomplete or wrong data**, one of the several methods can be used:

- The user input is blocked after the control has started. This can be done by changing the *Enabled* property of the user input textboxes (or the entire group) to false. The user, however, cannot change the input while the control program is running.
- Adding an *Update* command button to the *User Input* group. The user input values are updated inside the code only after the user presses on this button. This assumes that the user has entered correct values in the textboxes before the *Update* button was pressed.
- Adding an event of clicking on the textbox. The event handler for clicking the textbox will cause the input values to be updated.

For outputting values to the interface, the data does not need to be updated in every scan through the code. First, this wastes computational resources. Second, the human eye will not be able to respond to such a fast update rate. An update rate of about 5 Hz is sufficient. Furthermore, with the use of real variables, one should format the output so only a few digits are displayed. This can be accomplished using the *Format* function in VBE.

6.13 Chapter Summary

This chapter addressed timing and control software structures. It focused on software issues that arise when using a microcontroller or a PC as the controller in a mechatronics system. This chapter started by discussing how timers are implemented using a combination of clock source and a counter. It then discussed timer issues such as resolution and overflow. This was followed by a discussion about the timing functions in MATLAB, VBE, and a PIC microcontroller. The chapter then focused on the task/state software structure for control of mechanical systems, the concept of task scanning, and state organization. The task/state software structure uses a state-transition diagram to represent

the task activity and can be used for structuring software for both discrete-event and feedback control applications. Discrete-event control refers to the control of a sequence of events or actions, while feedback control is used for regulation or tracking applications. States are mutually exclusive, and a task can be in only one state at a given time. Coding examples for state-transition diagrams in MATLAB, VBE, and in a PIC microcontroller were presented. This was followed by discussing the cooperative and the preemptive control modes, the two basic control software structures for handling multitasking control programs. The concept of a thread and a process was also discussed. Threading brings the issue of resource sharing among the different tasks and the means available to handle shared resources such as mutual exclusion and semaphores are discussed in this chapter. The limitations of common operating systems was also discussed, and the need for real-time operating systems (RTOSs) in certain applications was addressed. RTOS implementation in PIC microcontrollers using the CCS compiler was discussed. Also, a commercial preemptive RTOS system (ThreadX) that is widely used in embedded control applications was discussed in this chapter. Developing a graphical user interface (GUI) for a control program was the last topic covered in this chapter. MATLAB and Visual Basic Express approaches for building a GUI were presented.

QUESTIONS

6.1 How is a timer implemented in a processor?

6.2 Explain what is meant by timer overflow.

6.3 How can timer overflow be detected?

6.4 Name two timing functions in VBE.

6.5 For what cases the is Performance Counter used?

6.6 List the available timing functions in MATLAB.

6.7 Name the types of control tasks.

6.8 What is a state-transition diagram?

6.9 Why are state-transition diagrams important?

6.10 List conditions that cause transitions between states.

6.11 Name the code sections in a particular state.

6.12 Is performing a '*For-Loop*' inside a state considered a blocking code?

6.13 List several examples of blocking code.

6.14 What is the difference between cooperative and preemptive control modes?

6.15 Can data-corruption occur when running tasks in cooperative control mode?

6.16 What is the difference between a thread and a process?

6.17 For what purpose is the *BackgroundWorker* component used in VBE?

6.18 Name several resource-sharing mechanisms.

6.19 What is a 'race' condition in multithreaded applications?

6.20 For what purpose are RTOS systems used?

6.21 How is a GUI created in MATLAB?

Problems

P6.1 Determine the resolution and the maximum counting interval of a timer that uses a 1 MHz clock that feeds into a 16-bit counter.

P6.2 Write pseudocode (syntax is not important) that allows one to determine the execution time of a certain function or code segment. Assume you have access to the function that returns the current time information.

P6.3 Describe the type of timer (interval or absolute) needed and a reason for the following operations.

a. Alarm monitoring system

b. Climate control system

c. Elevator door opening/closing

d. Screen saver program

P6.4 Monitor the operation of an appliance (such as a dishwasher or a washing machine) and write a state-transition diagram for the different states of operation.

P6.5 Develop a state-transition diagram for a software counter. The counter should count up when the user presses the *UP* command, and count down when the user presses the *DOWN* command. The counting should stop when the user presses the *STOP* command or when the count reaches a user-specified limit such as +100 or −100. Show the different states and the conditions that cause transitions between states.

P6.6 Develop a state-transition diagram for the operation of a linear motion positioning table system that operates as follows. The system is controlled by three commands: *Move Right*, *Move Left*, and *Stop*, which cause the system to move right, left, or stop, respectively. At each end of the travel, a limit switch is mounted that should cause the system to stop if the table touches the switch. Show the different states and the conditions that cause transitions between states.

P6.7 Develop a state-transition diagram for the operation of a garage entry system that operates as follows. The user presses a button to get a ticket or swipes a card in a card scanner. Once the ticket is picked up by the user or the card is validated, the gate arm rotates upward. The gate arm remains in a raised position until the vehicle has completely cleared the gate or a waiting interval has elapsed, at which point the gate drops down. The system has a proximity sensor to prevent the gate from striking people and vehicles, and the gate rotates upward if an object is detected while moving downward.

P6.8 Develop a state-transition diagram for the operation of the four-position rotary indexing table shown in Figure 6.12 and discussed in Example 6.2. Assume that the indexing table needs to be first homed to determine a starting position for the table.

P6.9 Develop a state-transition diagram for the operation of a vending machine. The vending machine operates as follows. The user enters the required money and then selects an item to be bought. The machine dispenses the selected item and returns change to the user if needed. The machine displays a message if the selected item is not available. The user can cancel the transaction before an item is selected.

P6.10 A resource needs to be used by three different tasks with only one task having access to the resource at a given time. Describe a resource-sharing mechanism that one can use to control access to the resource.

Laboratory/Programming Exercises

L/P6.1 Using the *Timer property* in VBE, which overflows every 24 hours, write code to display the time since the application started when the user presses a button called 'GetTimeSinceStart.'

L/P6.2 Write a program that uses the *Timer component* in VBE to trigger events at intervals ranging from one second to 10 hours. Set the Timer component interval to 100 ms. Note that such a program can be used for scheduled monitoring

of signals or to perform control actions at a defined interval.

L/P6.3 Using the *Timer object* in MATLAB, implement MATLAB code that displays a message to the command window every 10 s.

L/P6.4 Using the Performance Counter, write a VBE program that can capture the open-loop step speed response of a motor or any other dynamic system. The user specifies the sampling interval, the test duration, and the desired step magnitude. The program applies the step input to the system at time zero and then records the desired number of samples at the desired sampling rate. When the test duration is over, the program writes the data to a text file. (Note: This problem assumes the availability of a data-acquisition card with a software library for accessing the A/D and D/A.)

L/P6.5 Implement, using MATLAB or VBE, the software counter discussed in Problem 6.5. The count value should be displayed to the user in a textbox. The counting rate is controlled by a timer, and the user should be able to change the counting rate while the code is running. Ensure that the software program that is implemented follows the design of the state-transition diagram.

L/P6.6 Implement, using any PIC MCU, the software counter discussed in Problem 6.5. The count value should be displayed to the user using either LEDs or through an RS-232 connection to a PC. The *UP*, *DOWN*, and *STOP* commands should be implemented through push-button switches that are connected to the MCU. Use a rotary pot to vary the counting rate.

L/P6.7 Use MATLAB to implement the state-transition diagram for the operation of the positioning system in Problem 6.6. Develop a MATLAB GUI for the user interface.

L/P6.8 Use MATLAB to implement the state-transition diagram and a GUI for the thermostat control problem discussed in Section 6.7.1.

L/P6.9 Use VBE to implement the state-transition diagram and a GUI for the thermostat control problem discussed in Section 6.7.2.

L/P6.10 Use the PIC MCU to implement the state-transition diagram for the thermostat control problem discussed in Section 6.7.3.

L/P6.11 Using the PIC-C compiler with the RTOS-provided functions, develop an RTOS program that implements a feedback control system that runs every 2 ms.

L/P6.12 Download an evaluation copy of ThreadX RTOS and run the provided example code to familiarize yourself with the operation of a commercial RTOS system. In particular, do the following:

a. Run the project for few minutes and observe the behavior of the system. In particular, track the thread counters and explain the sequence of threads operation.

b. Change the priority and the preemption threshold of one of the threads in the thread create statement. Compile the project, and animate the system. Explain what you observe in the execution of threads that interact with the thread whose priority has been changed. Be specific in your explanation. Repeat this approach for other threads.

c. Change the initial value of one of the semaphores from 1 to 2 in *tx_semaphore_create*(..) statement in the code. Compile the project, and animate the system. Explain what you observe in the execution of threads that use that semaphore. Be specific in your explanation.

d. Change the initial value of one of the semaphores from 1 to 0 in the *tx_semaphore_create*(..) statement in the code. Compile the project, and animate the system. Explain what you observe in the execution of threads that use that semaphore. Be specific in your explanation.

CHAPTER 7

Sensors

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

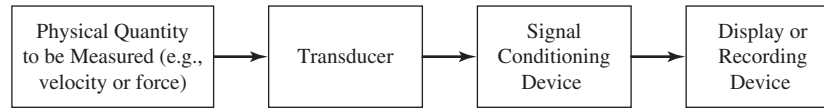
- Interpret a sensor performance specification
- Select a specific sensor for a given measurement application
- Explain the different types of displacement, proximity, speed, temperature, and vibration sensors
- Predict the output of strain gage-based sensors
- Explain the principle of operation of many of the different sensors covered in this chapter
- Explain the use of filters in signal processing
- Analyze a bridge circuit to process the output of resistance-type sensors

7.1 Introduction

Sensors are vital components of mechatronic systems, since they provide information that allows us to monitor and to control the operation of these systems. Without the availability of sensory information, automated systems cannot operate. A sensor is an element that produces an output in response to changes in physical quantity (such as temperature, force, or displacement). The active element of some sensors is called a **transducer**, which is the part of the sensor that converts the physical quantity (such as the force or displacement into an equivalent electrical signal in the form of voltage or current). The physical quantity changes a property of the transducer (such as its resistance, inductance, or magnetic coupling). Through electronic circuits, these property changes of the transducer are converted into a low-level voltage or current electrical signals. The terms sensor and transducer are sometimes treated as synonyms, but note that not all sensors produce an electrical signal as an output. Examples include the mercury bulb thermometer and the spring scale force sensor. Figure 7.1 shows a block diagram of the process of measurement using a sensor with a transducer. Normally, the output from the transducer is not in a form suitable to be read by a display device or meter, and signal conditioning operations (such as filtering or amplification) are needed to process the output.

There are a variety of sensors available that are commonly used. These include sensors that measure motion-related information (such as strain, speed, displacement, and acceleration). Also, sensors are available to measure process parameters (such as temperature, level, and pressure). This chapter will focus more on sensors that

Figure 7.1
Measurement process



measure motion-related information. The next chapter will discuss electric actuators. Both sensors and actuators are key to implementing feedback control of motion-control systems. We start this chapter by discussing some of the performance parameters of sensors. For further reading on sensors and measurements, see [27-29].

7.2 Sensor Performance Terminology

There are a number of parameters that characterize sensors' performance. The time-independent characteristics are called the static characteristics, while the time-dependent characteristics are called the dynamic characteristics. The **static characteristics** characterize the sensor output after it has settled due to changes in the physical quantity being measured. The **dynamic characteristics** describe the sensor characteristics from the time the physical quantity has changed to the time before the output has settled.

7.2.1 Static Characteristics

Range Minimum to maximum value that can be measured is the range. The range defines the allowable range of the physical quantity that can be detected by the sensor.

Accuracy The difference between true and actual measured value is the accuracy. It is commonly expressed as a percentage of full-scale value. For example, if a temperature sensor has a range of 0 to 200°C and an accuracy of $+/-0.5\%$ full-scale value, then the temperature read by the sensor is off from the true actual temperature by $+/-1°$. Note that the accuracy error can be improved by calibration.

Sensitivity The relationship between the measured input and the output of the sensor is its sensitivity. If the sensor has a linear input–output relationship, then the sensitivity is the slope of this curve. Sometimes, this parameter is used to indicate the sensitivity of the sensor to non-measured input (response due to transverse motion when the sensor is designed to measure axial motion) or the environment (temperature).

Resolution The smallest change in input value that will produce an observable change in the output is the resolution. The inherent resolution should be distinguished from the display device resolution.

Hysteresis The maximum difference in sensor output for the same input quantity is the hysteresis, with one measurement taken while the input was increasing from zero and the other by decreasing the input from the maximum input. A sensor with hysteresis will have a different output value that is a function of whether the input quantity was increasing or decreasing when the measurement was made. The hysteresis error is illustrated in Figure 7.2.

Repeatability Error in output value for repeated application of the same input value is called repeatability or precision. The smaller the repeatability error, the higher the measurement precision would be. Repeatability is affected by signal

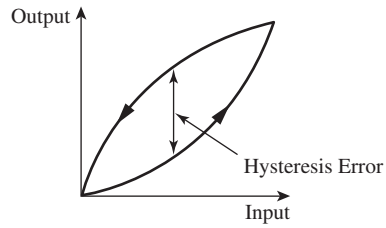
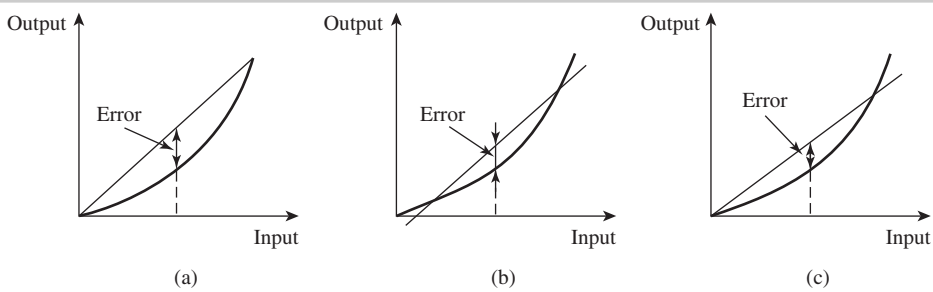


Figure 7.2
Illustration of hysteresis error

interference, vibration, and temperature fluctuation. Repeatability error cannot be reduced by calibration.

Non-Linearity Error Most sensors are designed to have a linear output, but their output is not perfectly linear. The non-linearity error is a measure of the maximum difference between the sensor actual output and a straight line fit to the sensor input–output data and is usually specified as a percentage of the full-scale output. There is no unique way to obtain the straight line fit. The straight line can connect the minimum and maximum output values that define the sensor range, or it can be obtained from a least-square fit to the entire input–output data or from a least-square fit to the input–output data with one end of the line passing through the origin. Figure 7.3 illustrates these cases and shows that the magnitude of this error is dependent on how this error is defined.



(a) Minimum to maximum fit
(b) Least-square fit through data
(c) Least-square fit with one end through the origin

Figure 7.3
Illustration of non-linearity error

Stability Stability or drift refers to the variation of the output with time when the input quantity is not changing. When no input is applied to the sensor, the output variation is called the zero drift. Stability affects the repeatability of the measurement.

7.2.2 Dynamic Characteristics

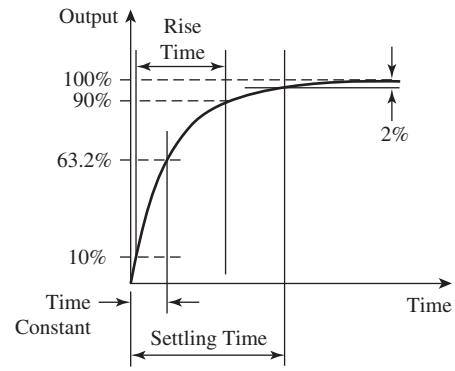
Rise Time The time it takes the output to change a certain percentage is the rise time. A common measure is the time for the output to change from 10 to 90% of the final steady–state value.

Time Constant This is defined as the time it takes the output to reach 63.2% of the final output. A large time constant implies a sluggish sensor, while one with a small value indicates a rapidly responding sensor. If the sensor has a first-order response characteristics, then it takes about four time constants to reach the final value when subjected to a step input.

Settling Time The time it takes the output to reach within certain percentage of the final steady–state value is the settling time. A common value is the 2%

Figure 7.4

Illustration of basic dynamic response characteristics



settling time. The rise time, time constant, and settling time are illustrated in Figure 7.4 for a sensor with first-order response characteristics.

Bandwidth The bandwidth defines the frequency range for which the sensor is designed to operate. At the bandwidth frequency, the sensor output will be 70.7% of the DC level. The sensor can operate at frequencies higher than the bandwidth, but the output of the sensor will be significantly diminished. When a sensor is used to provide feedback information in a closed loop control system, the sensor bandwidth should be larger than the controller bandwidth.

Values for each of these sensor performance characteristics are found in the manufacturer data sheet for the particular sensor. Normally, the specification in the data sheet is grouped into categories, including dynamic or performance, electrical, mechanical, environmental, and physical. An example of some of the characteristics for a compression load cell is shown in Table 7.1, including a description of each specification.

Table 7.1

An example of the specifications for a load cell sensor

Item	Value	Explanation
Rated Capacity	10 lbs	The maximum weight that the cell is rated to handle
Excitation	10 VDC	The cell requires a 10-V DC power supply to operate
Rated Output	2 mV/V nominal	The nominal cell output at the maximum load (10 lbs) will be 20 mV (2 mV/V × 10 V)
Linearity	+/−0.25% FS	With a 5-lb load applied, the cell output will indicate a load of 5 lbs +/−0.025 lb
Hysteresis	+/−0.15% FS	Due to hysteresis, the cell output can be vary by +/−0.015 lb
Maximum Load (Safe Overload)	150% of rated capacity	The allowable increase in the rated capacity. For this load cell, the maximum load should not exceed 15 lbs
Bridge Resistance	350 Ω	The resistance of the strain gage inside the load cell

7.3 Displacement Measurement

Displacement sensors are ones that provide information about the change in the position of a rigid body. The sensors can be classified as those that provide **analog output** (such as potentiometers and resolvers) and those that provide

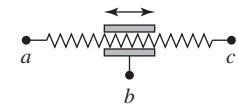
digital output (such as encoders). Displacement sensors also can be classified as contact or non-contact, depending on whether the sensor contacts the object during measurement. **Contact-type displacement** sensors include strain gages and potentiometers, while **non-contact displacement** sensors include encoders and capacitive-type displacement sensors. This section will discuss different types of displacement sensors.

7.3.1 Potentiometers

A potentiometer is a contact-type sensor that provides displacement information by measuring the voltage drop across a resistor. Potentiometers can be of the linear or rotary type. A **linear potentiometer** is designed to measure linear displacement. The sensor has a linear slider that is attached to the object whose displacement needs to be measured (see Figure 7.5). The displacement of the slider changes the electrical resistance between nodes *a* and *b*, which then can be used as a measure of displacement. In normal operation, a DC voltage is applied between nodes *a* and *c*, and the voltage output between nodes *a* and *b* is used as a measure of displacement. If node *a* is connected to the ground, then the *b* node voltage will increase as the slider travels from *a* to *c*. The *b* node is commonly called the wiper.

Figure 7.5

Model of a linear potentiometer



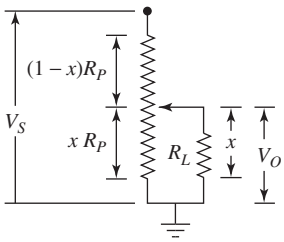
A **rotary-type potentiometer** (see Figure 7.6) is designed to measure angular displacement. The sensor has a rotary knob that is coupled to the shaft of the object whose angular displacement needs to be measured. Similar to a linear potentiometer, the rotation of the knob changes the electrical resistance between the leads of two nodes on the potentiometer. Rotary potentiometers are available as **single-turn** or **multi-turn** devices. Multi-turn devices can measure several shaft revolutions, while single-turn devices are designed to measure a rotation of up to one revolution. Note that some single-turn devices cannot measure a complete revolution due to the construction of the potentiometer with a dead zone, which prevents the wiper on the potentiometer from making a complete turn. The contact element in a potentiometer is constructed either from a wound wire or from conductive plastic. Wire-wound elements provide better stability and linearity than conductive plastic, but conductive plastic offers better resolution and longer life. The use of a wound wire results in a step change in the output voltage (and hence a coarser resolution than conductive plastic) as the slider moves from one turn in the coil to the next turn. Potentiometers have the advantage that they are easy to use, but because they are contact-type devices, they have a frictional resistance, which affects the

Figure 7.6

A commercial rotary potentiometer
(© Wayne Higgins/Alamy)

Figure 7.7

Model of a potentiometer interfaced with a measuring device with load resistance (R_L)



motion of the measured object. Because potentiometers provide an analog voltage as their output, they also need to be interfaced with an A/D converter before the signal is read by a PC or a microcontroller.

The resistance of a potentiometer is important. A high resistance results in a smaller current and hence less heat loss through the potentiometer while it is in operation, but it also worsens the loading error, since in practice, the output voltage of the potentiometer is read by a device that does not have infinite impedance (see Section 2.6). Loading introduces **nonlinearities** into the potentiometer output. To see this, refer to Figure 7.7, and assume that the potentiometer has a resistance R_P, the measuring device has a resistance R_L, and the supply voltage is V_S. Then the voltage output at any position x ($0 \leq x \leq 1$) is given by the relationship

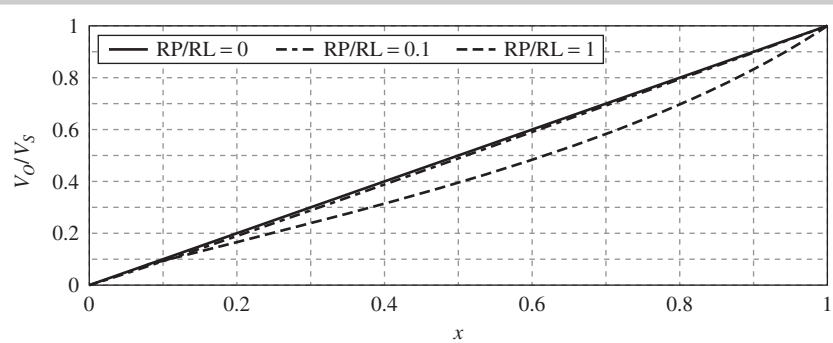
$$V_0 = \frac{xV_S}{1 + x(1 - x)\frac{R_P}{R_L}} \tag{7.1}$$

This relationship is obtained by using the voltage dividing rule to compute the output voltage (V_O) and noting that the load resistor (R_L) and the potentiometer resistor ($x\,R_P$) are two resistors in parallel. Note that if the load resistance (R_L) is infinite, then R_P/R_L is zero, and Equation (7.1) reduces to $V_O = x\,V_S$, where the output voltage is directly proportional to the slider position x. If R_L is not infinite, then the output voltage varies nonlinearly with the slider position x. The nonlinearity worsens as the ratio of R_P/R_L increases.

Figure 7.8 shows a plot of Equation (7.1) for three values of the ratio R_P/R_L: one for $R_P/R_L = 0$, another for $R_P/R_L = 0.1$, and the third for $R_P/R_L = 1.0$. Note how the output voltage varies nonlinearly with x, especially for $R_P/R_L = 1$. To eliminate this problem, one should select a potentiometer with as small a resistance as possible. Example 7.1 illustrates computations for a potentiometer.

Figure 7.8

A plot of Equation (7.1)



Example 7.1 Potentiometer

A single-turn rotary potentiometer with a 330° measurement range is used to provide angular-position feedback information for a positioning application. A 5-V DC voltage is applied across the potentiometer leads, and the potentiometer output is connected to a 12-bit A/D convertor with a +/−5 V range. The potentiometer resistance is 50 Ω. Determine:

a. The effective resolution of this sensor
b. The power loss by the potentiometer, assuming a half-motion displacement

Solution:

a. Assuming that the potentiometer uses a film as the resistance element, then the resolution is determined by the A/D resolution. The 330° motion range is mapped into the 5-V output, and the A/D total voltage range of 10 V is mapped into 2^{12} positions. Thus, the angular resolution is

$$1/4096 \times 10/5 \times 330° = 0.161°$$

b. If the load impedance is considered to be infinite, then all of the current is passed through the potentiometer. In this case, the current is

$$5\ V/50\ \Omega = 0.1\ A$$

and the power loss is i^2R or 0.5 W.

If the load impedance is not infinite, then some of the current will pass through the load, and the power dissipated by the potentiometer will be smaller than computed above.

7.3.2 LVDT

The linear variable differential transformer (LVDT) is a device for measuring mechanical displacement. The device has a simple construction and consists of a moveable iron core surrounded by three transformer coils (see Figure 7.9). An external AC voltage is applied to the center (primary) coil, and the output signal is obtained from the two end (secondary) coils which are connected in opposite phase. The excitation frequency is typically several kHz, but it could range from 100 Hz to 20 kHz. Similar to a regular transformer, the voltage in the secondary coil is proportional to the number of turns in the secondary coil that are coupled to the turns in the primary coil. The position of the core affects the coupling between the center and the two end coils, and thus, the AC output signal is proportional to the displacement of the core relative the windings. The AC output signal, which has the same frequency as the excitation frequency, is processed to produce a DC output signal proportional to the displacement that can be read by an A/D convertor or connected to a display device. Processing the AC signal includes rectification to produce a DC signal, filtering to remove high-frequency signals, and amplification to produce a suitable voltage output level. A low-pass filter with a cut-off frequency about 10% of the excitation frequency is used. The filter passes the components associated with the low-frequency mechanical motion but filters the high-frequency excitation signal.

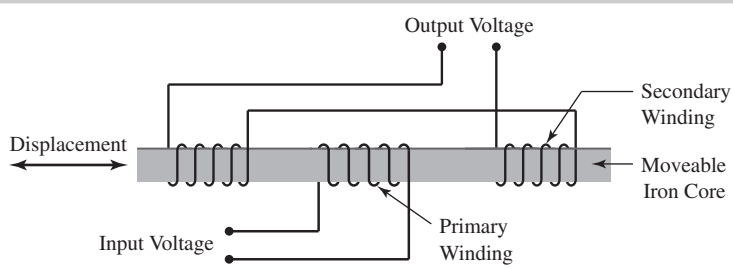


Figure 7.9
LVDT construction

To reduce noise sensitivity, the voltage-output signals from the two secondary coils (A and B) are computed using a **ratiometric formula** where the difference between the voltage signals is divided by the sum of the two signals as

$$V_0 = \frac{V_A - V_B}{V_A + V_B} \tag{7.2}$$

An LVDT has the advantage that it can be constructed to measure displacement ranging from a few centimeters to a few inches with almost infinite resolution. There is also no damage from overloading, as overloading simply separates the core from the device, and it is relatively insensitive to temperature changes. However, being a contact displacement sensor, the LVDT has a limited frequency response. Also, signal conditioning is required to process the output signal. LVDTs are available as DC or AC power operated. The DC configuration offers ease of installation and the ability to use battery power in measurement situations where AC power is not available, while the AC version results in a smaller body size and more accurate signal.

7.3.3 Incremental Encoder

Encoders are non-contact, optical-based digital devices that are used for measuring displacement. Similar to potentiometers, they can be used to measure both linear and rotary displacement. We will concentrate on rotary-type encoders, which are widely used in motion-control applications. Rotary encoders are available as incremental or absolute type. An **incremental encoder** measures changes in rotation from some datum position, while an **absolute encoder** measures the actual angular position. When an incremental encoder is used, the motion system needs to be 'homed' to establish reference information.

In its basic form, an incremental encoder is constructed from two light sources that shine light through a disk that has an alternating pattern of black and clear stripes. The light is sensed by two photodetector sensors that are located on the other side of the disk. To understand the operation of an incremental encoder, let us assume first that we have only one light source and one sensor. When the disk rotates, the light signal (as measured by the photo detector sensor) looks like that shown in Figure 7.10. The output of the sensor will be a train of pulses with each pulse corresponding to the light pattern that is captured by the optical sensor while one strip on the disk passes through the light sensor zone. To get angular displacement information, we simply count the number of pulses generated as the disk rotates from one position to another. Unfortunately, this simple scheme cannot provide us with direction information.

Figure 7.10

Output from a single light/sensor combination

To get direction information, incremental encoders have another light source and sensor (called channel *B*). The channel *B* sensor is located one-half slot-width apart from the first sensor and photodetector (channel *A*). There are two ways to implement this **offset** in practice.

1. There is only one track of lines: one light source-sensor combination is located to line up with the edge of one of the slots, while the other light source-sensor combination is located to have an offset of one half-slot with respect to the edge of one of the slots.
2. Two concentric tracks of lines are used: the slots in one track have an offset of one-half slot with respect to the slots in the other track, but the sensors have no offset between them.

The pattern of the two sensor signals generated for clockwise (CW) and counterclockwise (CCW) rotations of a disk under constant angular speed are shown in

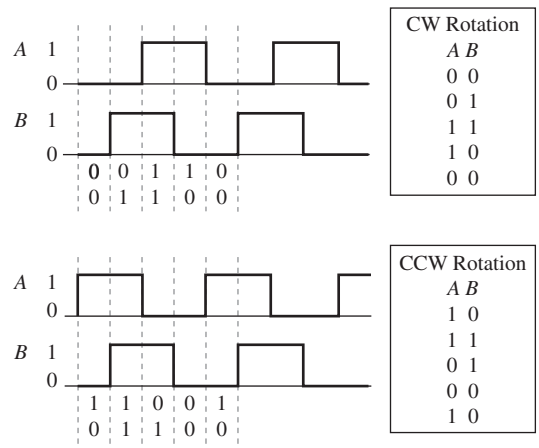


Figure 7.11

Output of an incremental encoder

Figure 7.11. Notice how the *B* signal leads the channel *A* signal by a quarter of a cycle for CW rotation, and how it lags behind the channel *A* signal by the same amount for CCW rotation. The cycle that we are referring to is the chord distance made up of one black strip and one clear strip. This lead/lag between the two channel outputs enables one to determine the direction of the rotation of the shaft that is attached to the encoder disk.

To understand this further, examine the *A* and *B* channel signal patterns for CW and CCW rotations that are shown in Figure 7.11. Notice that the output switches between one of four possible states for either rotation direction, and the order of these states is different for each direction. For example, if the photodetector sensors output is 00, then the next state will be 10 for CCW rotation and 01 for CW rotation. The different transitions for CW and CCW rotations enable one to write state-transition logic to determine the direction of rotation. Figure 7.12 shows an example of a state-transition diagram (see Section 6.4) that can do this job. Notice that, regardless of what state the output of the two sensors starts at, the diagram can determine the direction of rotation by examining the transitions from any one of the four possible states.

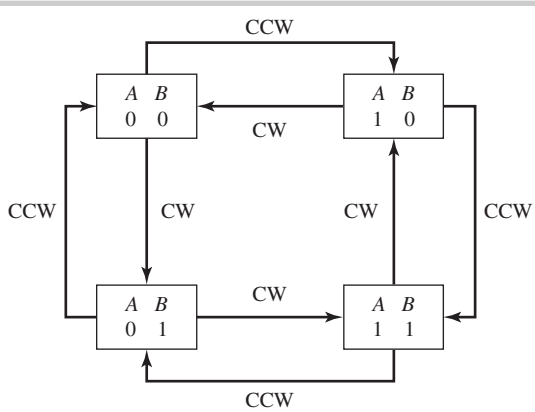


Figure 7.12

State-transition diagram for an incremental encoder

As noted before with the use of two sensors, we get four distinct states for each strip on the disk. Thus, if an encoder has 1000 strips (or lines), we will get 4000 distinct states per one revolution of the encoder disk. Thus, the use of two sensors improves the resolution of the encoder by a factor of 4, since we can count 4000 leading and trailing edges per one revolution compared to counting 1000 pulses per revolution for a single sensor. An encoder that gives four times the number of

lines is operating in **quadrature** mode. In practice, the number of counts per second can get very large. For example, if the 1000-line encoder was rotating at 1000 rpm, then we will get 66.6e3 counts per second if the encoder was operating in quadrature mode. Most PC's or microcontrollers cannot keep up with counting at this rate if the output from the *A* and *B* channels is directly connected to the digital input port of the PC/microcontroller. In practice, hardware counters are used to process the *A* and *B* signals instead of using a software counting solution. These dedicated counters implement logic very similar to the method shown in Figure 7.12. The counter value is incremented by 1 on each state transition if the motion happens to be in one direction and is decremented by 1 for a motion in the opposite direction. To get the current position information, the PC or the microcontroller simply reads the output of the hardware counter. Thus, the processor does not have to worry about accuracy problems resulting from failing to count a particular transition. Example 7.2 illustrates the application of encoders.

Example 7.2 Incremental Encoder

A DC motor equipped with an incremental optical encoder is used to drive a lead-screw positioning table, as shown in Figure 7.13. The screw has a lead of 0.1 in./rev., the encoder disk has 1000 lines, and the encoder is operated in quadrature mode. Determine the measurement resolution of this encoder for the following.

a. The setup shown Figure 7.13.
b. The motor replaced with a geared one with a 5:1 gear ratio.

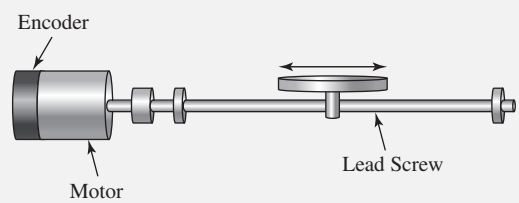


Figure 7.13

Solution:

a. The table travels a distance of 0.1 in. or 2.54 mm per one revolution of the motor. During this interval, the encoder will output 4000 counts (1000 $\times$ 4). Thus, the measurement resolution of this encoder setup is

$$2.54 \text{ mm}/4000 \text{ counts} = 0.635 \ \mu\text{m per count}$$

b. The encoder is mounted on the input side of the motor. Thus, if the motor rotates one revolution, the lead screw will rotate 0.2 revolution due to the use of a 5:1 gear on the output shaft of the motor. Hence, the encoder will generate 4000 counts for 0.508 mm (0.1 in. $\times$ 0.2) travel of the table, and the measurement resolution in this case is

$$0.508 \text{ mm}/4000 \text{ counts} = 0.127 \ \mu\text{m per count}$$

Note that while the measurement resolution is high, the part (b) configuration is not normally used for high-precision applications due to backlash in the gears and lead screw. A linear encoder mounted directly on the table is used instead.

Figure 7.14 shows an example of a **commercial counter IC**. The LS7166 IC is a 24-bit counter that can count in different modes, including up/down and quadrature. From Example 7.2 using 1000 rpm rotation speed and a 1000-line encoder operating in quadrature mode, this counter can count an interval exceeding 250 s before overflow. The incremental encoder *A* and *B* lines are connected to the *A* and

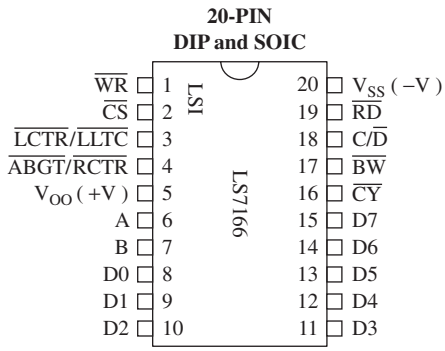


Figure 7.14

Commercial counter IC
(Courtesy of LSI Computer Systems, Melville, NY)

B pins on this IC. This counter uses an 8-bit (pins D0 through D7) three-state I/O bus to communicate with external circuits. An 8-bit bus is used instead of a 24-bit to reduce the number of wires needed but also to be compatible with most microcontrollers/external devices, which have a limited number of I/O lines. An external device can read the counter value by simply sending a preset control value over the I/O bus. This causes the 24-bit counter value to be transferred to the output port of the counter or the output latch. The three byte contents of the output latch are then transferred by performing three successive read operations of the output latch where (after each byte is read) the address pointer for the next byte is automatically incremented.

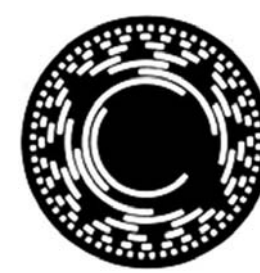
While incremental encoders do not have the limitations of potentiometers in terms of limited motion range and friction due to contact, they need to be 'homed' before they can be used in a motion-control application. In *homing*, the motor is rotated in one direction until a reference signal changes state. The output of the counter is recorded at this location, and displacements are measured with respect to this reference counter or home position. Most commercial incremental encoders have a third output called the **marker** or **z-channel** that is used in the homing sequence. A limitation of incremental encoders is that homing may not be safe to perform at all times. An example would be a robot arm that uses incremental encoders and is used for operations inside a vehicle frame or in regions with obstacles. If the robot happens to lose power while it is inside the vehicle, the robot will lose its current position information after the power is turned back on. In this case, the robot should not be homed automatically because of the possibility of the robot hitting the vehicle or one of the obstacles. It would be better in this case to use a position sensor that does not need to be homed. Such sensors are called absolute encoders and are discussed in the following section.

7.3.4 Absolute Encoder

An absolute encoder is one which has different track information for different angular positions of the encoder disk. Figure 7.15 shows a layout of a commercial absolute encoder disk. There is no need for homing with an absolute encoder, since each angular position of the disk has a unique output.

Figure 7.15

8-bit commercial absolute encoder disk
(Courtesy of BEI Sensors, Goleta, CA)



Absolute encoders are available with two different types of output: natural binary and gray code. In **natural binary**, the output of the encoder as the disk rotates changes in the normal way that binary numbers increase (i.e., 00, 01, 10, 11, . . .). In **gray code**, the output only changes one bit at time as the disk rotates (i.e., 00, 01, 11, . . .). This makes gray code useful to reduce errors when reading the encoder if all the bits have not changed at the same time. Figure 7.16 shows the disk pattern (shown as linear for ease of display) and the corresponding output of the encoder for a 3-bit natural binary and gray code absolute encoders. Table 7.2

Figure 7.16

Disk pattern and output from each track of an absolute encoder

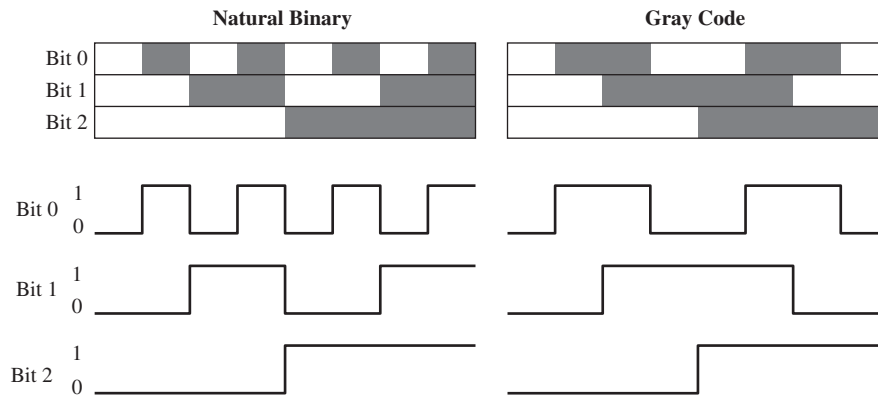


Table 7.2

Encoder output for natural binary and gray code

Position	Angular Segment (degrees)	Encoder Output	
		Natural Binary	Gray Code
1	0–45	000	000
2	45–90	001	001
3	90–135	010	011
4	135–180	011	010
5	180–225	100	110
6	225–270	101	111
7	270–315	110	101
8	315–360	111	100

gives the binary output patterns for both types of encoders for the eight different positions of the encoder disk.

Notice that this 3-bit absolute encoder (which has three tracks) can measure eight distinct absolute positions, each 45 degree in size (i.e., 0–45, 45–90, . . . etc). Commercial absolute encoders have typically 10-bit (or 10 tracks) or higher resolution, which give them an angular resolution of 360/1024 degrees or less. With the use of a geared motor, the resolution of the angular measurement of the output shaft increases by the gear ratio factor.

To use an absolute encoder for **multi-revolution measurement**, multiple disks need to be used. A high-resolution disk is used for the detailed position information and one or more disks are used for counting turns. For example, to use an absolute encoder having a measurement range of 16 revolutions, two disks are used. The second disk will have four tracks (to indicate 16 different turns) and should be coupled to the high-resolution disk through a 16:1 gear ratio. If the primary disk has a 10-bit resolution, then this two-disk encoder will measure 16×1024 or 16384 discrete positions.

Commercial absolute encoders are available with different types of output. These include parallel digital output, which uses a single line for each bit. For a multi-turn encoder with a 14-bit disk, this results in an interface cable that has over 30 wires, which increases the cost of the encoder. A smaller sized cable (and hence lower cost) is obtained if an encoder with SPI output is used. The SPI interface (see Chapter 5) uses only three wires for transmitting the data. Other output formats include DeviceNet™, Profibus, and Interbus.

7.3.5 Resolver

A resolver is an absolute angular-displacement measuring device, similar to an absolute encoder, but giving analog voltages as an output. Resolvers were originally developed for military applications, and they normally are used in rugged, harsh environments where encoders may not be suitable.

There are different types of resolvers. The most common is the **rotary brushless resolver control transmitter**. A schematic of the construction of such a resolver is shown in Figure 7.17. It has two parts: a rotor and a stator. The rotor has a winding, called the reference winding, which gets energized by an AC voltage signal in a non-contact fashion using a rotary transformer. The stator has two windings, called the SIN and COS winding, which are offset from each other by 90°. The rotation of the rotor induces voltages in the SIN and COS windings. These voltages are a function of the angular position of the rotary shaft. The revolver gives two analog output signals: one from the SIN winding and the other from the COS winding. The ratio of the SIN and the COS outputs is the tangent of the shaft angle.

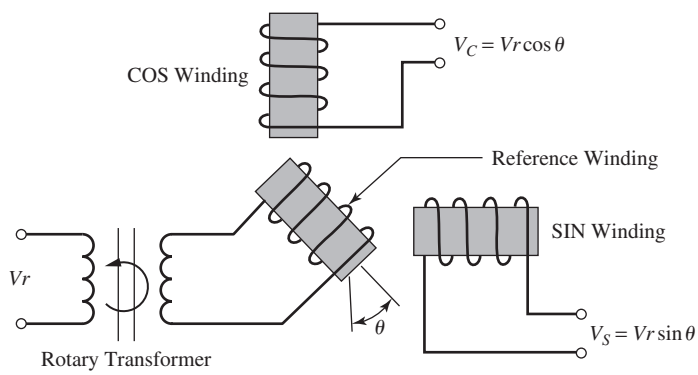


Figure 7.17

Schematic of rotary brushless resolver control transmitter

Similar to absolute encoders, resolvers are available in single- or multi-turn configuration. The multi-turn configuration actually uses two resolvers that operate similar to a vernier.

7.4 Proximity Measurement

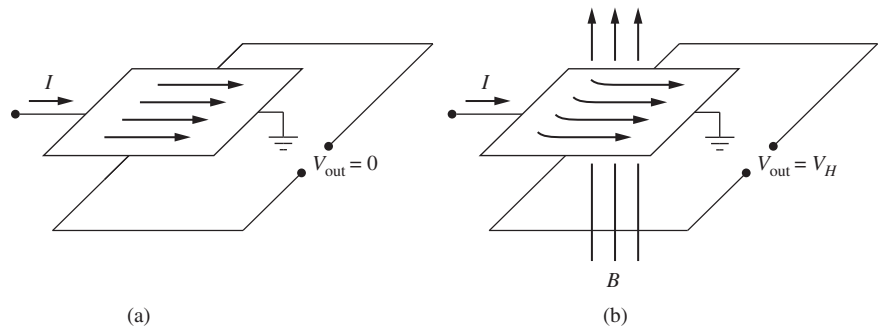
A proximity sensor measures the presence or absence of an object. Proximity sensors are widely used in products in various industries, including automotive, appliance, and manufacturing. Examples include sensors to detect seat-belt on/off status in vehicles, door and lid open/close detection in appliances, obstacle presence in closing powered doors, rotor angle position in brushless DC-motors, and end-of-travel detection in pneumatic actuators. There are several types of proximity sensors, including Hall-effect, inductive, capacitive, photoelectric, ultrasonic, and switch-type contact. Some of these sensor types will be discussed here.

7.4.1 Hall-Effect Sensors

A Hall-effect sensor is a non-contact type sensor that is based on the Hall effect, which was discovered by Hall in 1877. The **Hall effect** states that a voltage difference is developed in a current-carrying conductor when subjected to a magnetic field. This voltage is perpendicular to both the current and the magnetic field.

Figure 7.18

Illustration of the Hall effect (a) Current in a conductor with no magnetic field applied and (b) current in a conductor with a magnetic field perpendicular to the current flow



The Hall effect is illustrated in Figure 7.18. Figure 7.18(a) shows a thin conducting plate in which a current is flowing. The voltage difference across the sides of the plate will be zero in this case. If, however, we apply a magnetic field to the plate perpendicular to the direction of current flow, as shown in Figure 7.18(b), then the current distribution will be affected, and a voltage difference will be developed at the plate sides. The voltage is given by **Lorentz's law** and is equal to

$$\vec{V}_H = \vec{I} \times \vec{B} \tag{7.3}$$

where ***I*** is the current vector and ***B*** is the magnetic flux vector.

Note that the voltage difference is perpendicular to both the current flow and the magnetic flux direction. The amount of voltage that is generated is typically small (microvolt, μV) and a differential amplifier is used to amplify this voltage signal.

Hall-effect sensors are solid-state sensors that are constructed using semiconductor processing techniques. A Hall-effect proximity sensor consists of two pieces: a stationary sensor package and a magnet that is attached to the object whose presence needs to be detected, as seen in Figure 7.19. The magnet and the sensor package are separated by an air gap. There are two variations of Hall-effect sensors: unipolar and bipolar. In the **unipolar** design, when a south pole magnet approaches the designated package surface within a specified distance, the sensor turns ON. When the magnet is removed, the sensor turns OFF. In the **bipolar** design, removal of the south pole does not cause the sensor to turn OFF; a north pole needs to approach the sensor to cause the sensor to switch OFF. A typical circuit for a Hall-effect digital proximity switch is shown in Figure 7.20.

Figure 7.19

Hall-effect proximity sensor

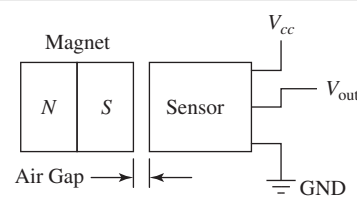
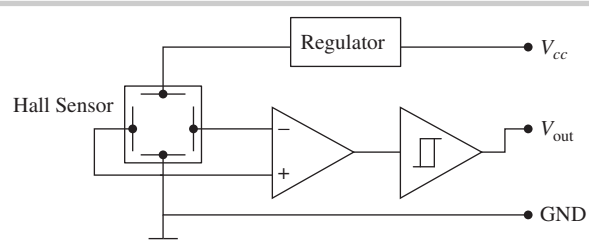


Figure 7.20

Hall-effect proximity switch wiring



In this circuit, the supply voltage is connected to the Hall sensor through a voltage regulator. The Hall-effect voltage is processed through a differential op-amp (see Section 2.9.4) to amplify the voltage generated by the Hall-effect sensor. The output of the differential op-amp is connected to a **Schmitt trigger**. The Schmitt trigger (see Section 2.9.3) compares the output voltage form the differential op-amp to a preset voltage level. If the output voltage exceeds the preset voltage, the switch output will be set high. When the differential op-amp output falls below a threshold level, the switch output will be set low. The hysteresis of the Schmitt trigger is used to reduce the sensitivity of the sensor to noise and false triggering. The Schmitt trigger output can also be connected to a switching transistor.

An example of a commercially available Hall-effect sensor is shown in Figure 7.21.

Figure 7.21

Commercially available Hall Effect sensor

(Courtesy of OPTEK Technology, Carrollton, TX)

7.4.2 Inductive Proximity Sensors

Inductive proximity sensors utilize the eddy current generated when a metallic element is placed within the proximity of an electromagnetic coil. The principle of operation of the sensor is shown in Figure 7.22. The sensor has an oscillator circuit that creates a magnetic field in front of the sensor through the coil inductance. When a metal target enters this magnetic field, it changes the magnetic field in the oscillator. This causes a swirling current, called **eddy current**, to be generated in the coil. The change in current in the coil is detected by a circuit that is connected to a switching amplifier. The oscillator, the current-detection, and the switching-amplifier circuits are all normally housed within the resin of the sensor.

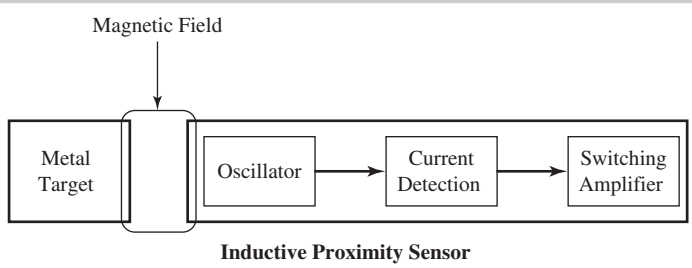


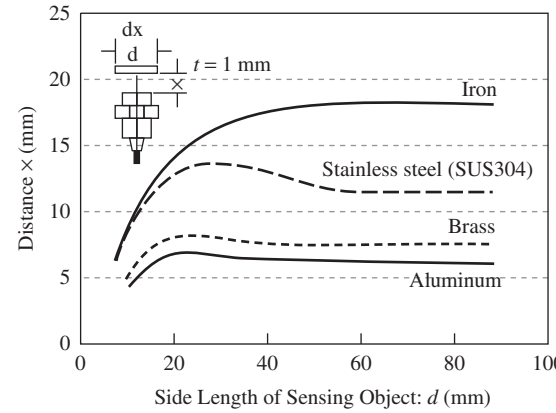
Figure 7.22

Inductive proximity sensor

Figure 7.23(a) shows commercially available inductive proximity sensors. While most inductive sensors are cylindrical in shape, rectangular-shaped sensors are also available. Cylindrical-shaped sensors are available with threaded or flat surfaces. Some units have an LED built into the sensor head to provide indication of object detection. The sensor electronics can be built into the sensor head or located separately from the head. Unlike Hall-effect sensors in which the target material is magnetic, inductive proximity sensors detect all metal objects at distances ranging from 1 to 30 mm. The larger the size of the sensor, the longer the detection range is. Standard inductive proximity sensors have a reduced detection range for nonferrous metals (such as copper, aluminum, and brass) than for ferrous metals (such as steel and iron, see Figure 7.23(b)). For non-metal objects, capacitive-type sensors can be used instead.

Inductive proximity switches, as all switches, are available in either NO or NC switch configuration. Furthermore, wiring to these sensors is available in either two- or three-wire configuration. In the three-wire configuration, the output is available

(a)



(b)

Figure 7.23

(a) Commercially available inductive proximity sensors and (b) detection range for a typical sensor for different metals
(Courtesy of Omron Corporation)

with either NPN or PNP transistor configuration. Example 7.3 illustrates the wiring circuit for a two-wire NO inductive proximity sensor used as a switch in relay circuit.

Inductive proximity sensors are also used to detect vehicle presence at **intelligent traffic lights**. These traffic lights are commonly used in rural or country roads where the traffic volume is variable. The sensor takes the form of a wire loop that is placed in groove that is cut in the asphalt surface. When a vehicle passes over the loop, the inductance of the loop is affected by the presence of the metallic body of the car. The electronics sense the vehicle presence and use this information to adjust the traffic light timing.

Example 7.3 Two-Wire Inductive Proximity Sensor

Draw a wiring circuit for a two-wire NO inductive proximity sensor used as a switch in a relay circuit. The output circuit of the sensor is shown on the left side of Figure 7.24.

Solution:

The circuit is shown below. The supply voltage (typically 24 VDC) is connected to one end of the relay coil, with the relay coil acting as the load resistor on the sensor output circuit. The other end of the relay coil is connected to the load input on the sensor circuit. The other wire of the sensor circuit is connected to ground. Since this is a NO sensor switch, the relay coil will not energize unless an object came within the detection range of the sensor. The detection of an object will thus cause the relay switch to close and to transmit power to the load connected to the relay.

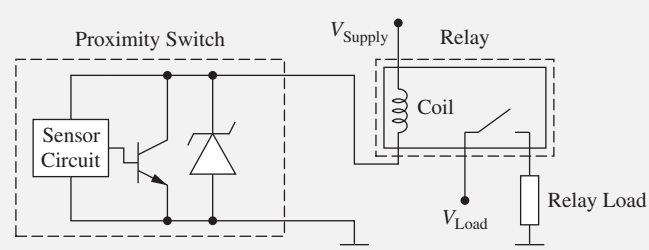


Figure 7.24

7.4.3 Ultrasonic Sensors

Ultrasonic proximity sensors detect the presence of objects by measuring the travel time of a high-frequency sound wave that is reflected off an object in the path of the transmitted signal. The sensor has a transducer that periodically emits a burst of sound at high frequency (200 kHz or higher) for a short time interval. After the transmission of the burst signal, the sensor switches to receiving mode and records the time when the echo signal was received by the sensor. This process is repeated continuously. Since the speed of sound is known in the transmission medium such as air, the time between the transmission of the source signal and the arrival of the reflected signal is then used to infer the position of the object.

Ultrasonic sensors have a much larger detection distance than inductive proximity sensors (m versus mm), but they cannot be used to measure the presence of very close-by objects (few centimeters away). This is because the echo of the leading burst signal for such nearby objects could be received while the trailing edge of the burst signal has not left the sensor, since the sensor is not set to monitor for echoed signals while it is still transmitting. The size of the object detected by these sensors is dependent on the frequency of the sound signal, with small objects needing a lower maximum frequency than larger objects. Ultrasonic proximity sensors are available with an analog output voltage that is a function of the distance of the object away from the sensor or with two states of digital output that indicate object presence/absence within a defined zone. They are typically used for liquid level measurement and for object detection on production lines. One feature of ultrasonic sensors is that they are not affected by the color, transparency, or lighting conditions of the object being detected. However, they are not very suitable to use for detecting material that absorb high-frequency sound, such as cotton or sponge.

7.4.4 Contact-Type Proximity Sensors

Contact mechanical switches known as '**limit switches**' are used in robotic and machine tool applications to detect the end of travel for a moving axis. They are also used in conveyer systems and transfer machines to detect objects and packages as well as in elevators, scissor lifts, and safety guarding applications. These sensors are available with different 'operator' types that provide the interface between the contact object and the switch mechanism. These types include a roller plunger, a dome plunger, a roller lever, a telescoping arm, and a short lever. Figure 7.25 illustrates a few of these operator types. These sensors are rugged and can be used in harsh situations.

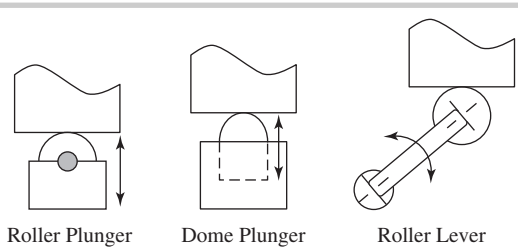


Figure 7.25

Operator types for limit switches

7.5 Speed Measurement

7.5.1 Tachometer

Figure 7.26

A tachometer
(Jouaneh, University of Rhode Island)

While speed information can be obtained by differentiating the position data, this approach is not very desirable as it amplifies the noise if the position signal is noisy. A better method is to use a sensor that can directly provide the speed information. A **tachometer** (see Figure 7.26) is a speed-measuring device that provides an analog output voltage that is proportional to the speed. A tachometer is constructed similar to a brush DC motor (see next chapter), but it is designed to operate in reverse. When the tachometer shaft rotates, the tachometer gives a DC output voltage. A characteristic of a tachometer is its sensitivity, which refers to the output voltage of the tachometer for a given speed. It is normally reported as a number of volts per 1000 rpm but other speed units can be used. Another characteristic of a tachometer is its ripple. **Ripple** refers to the AC component of the output signal. Due to the use of a commutator in the construction of the tachometer, the output signal of the tachometer exhibits fluctuation which can be as high as 3 to 4% of the nominal output voltage.

The ripple affects the operation of a closed loop speed control system since the control system responds to variation in the tachometer output voltage regardless of whether the variation is caused by a speed change in the motor or is due to ripple effect. One way to eliminate ripple is to place a low-pass filter constructed using an RC circuit on the output leads of the tachometer (see Figure 7.27). The R and C values should be chosen such that the cut-off frequency of the RC filter is below the ripple frequency. However, the use of an RC filter changes the dynamics of the feedback system.

Figure 7.27

RC filter on tachometer leads

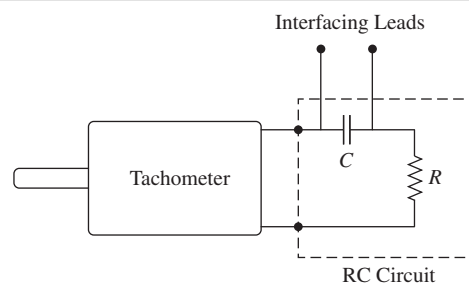
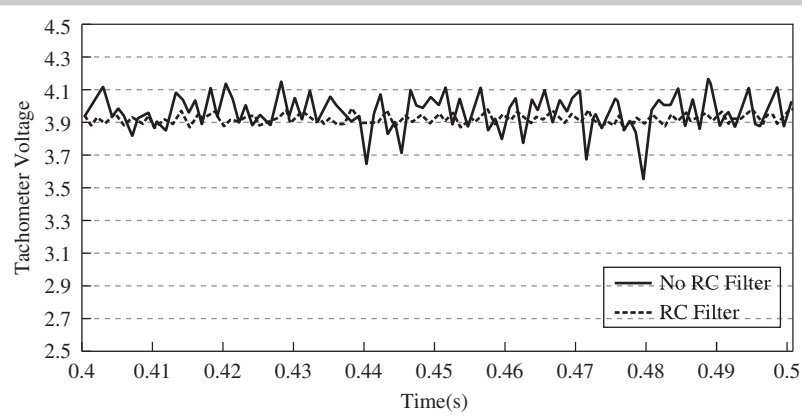


Figure 7.28 shows the output speed of a DC motor tachometer with and without using an RC filter to eliminate ripple.

Figure 7.28

Output speed of DC motor tachometer with and without an RC filter



7.5.2 Encoder

Incremental encoders can also be used to measure speed. There are two techniques that are normally used. These are the pulse-counting method and the pulse-timing method. In the **pulse-counting** method, the encoder count values are read at a certain fixed sampling frequency. The speed is obtained by dividing the difference between two successive encoder counter readings by the sampling time interval. If the encoder disk has l lines, the sampling interval is T in seconds, and the count difference between two successive readings is N, then the angular speed ω in rad/s is given by

$$\omega = \frac{2\pi N}{lT} \tag{7.4}$$

If quadrature is used, the above expression needs to be multiplied by ¼. The resolution of this technique increases with an increase in the speed, because more counts are generated in the given sampling interval as the speed increases.

In the **pulse-timing** method, a high-frequency clock is used to record the time interval for the motion travel between two adjacent lines on the encoder disk. Assuming a clock frequency of f cycles/s, an encoder disk with l lines, and m clock cycles recorded, then the angular speed ω is given by

$$\omega = \frac{2\pi/l}{m/f} = \frac{2\pi f}{ml} \tag{7.5}$$

This technique is particularly suitable for low-speed measurement. Note that as the speed increases, the resolution of this pulse-timing method decreases, since fewer clock cycles are used to record the motion travel between two adjacent lines on the encoder disk.

7.6 Strain Measurement

Strain is a basic quantity in solid mechanics. When a force (torque) acts on a member, it leads to a deformation of the member. The deformation is expressed in terms of strain. For elastic loading, the resulting stress (σ) and strain (ϵ) are linearly related through the modulus of elasticity of the material, E or

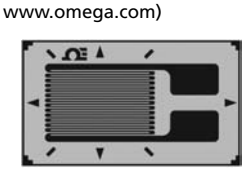
$$\sigma = \epsilon E \tag{7.6}$$

Strain is measured using a **strain gage** which is a resistor whose change of resistance is used as a measure of strain. The most commonly used strain gage is the metal-foil strain gage shown in Figure 7.29, which replaces the wire-resistance strain gage that was developed over 70 years ago. Other types of strain gages include a semi-conductor strain gage, which has a sensitivity of over 100 times that of the metallic gage.

Figure 7.29

Metal-foil strain gage

(Reproduced with permission of Micro-Flexitronics Ltd. (MFL), courtesy of Omega Engineering, Inc., Stamford, CT 06907 USA www.omega.com)



The metal-foil strain gage consists of a metal alloy foil, typically constantan, in the form of a grid placed on a flexible polyimide backing. The backing material serves as an electrical insulator from the metal part to which the gage is attached. The gage is bonded using adhesive to the surface of the part whose strain needs to be measured. Some gages are made with the lead wires already attached to the gage, but other gages provide an area where one can solder the lead wires. Standard rectangular gages are made with a grid gage lengths varying from 1.5 to 25 mm and grid gage widths varying from 1.2 to 8 mm.

Strain is defined as

$$\epsilon = \frac{\Delta l}{l} \tag{7.7}$$

where Δl is the change in length of a part of length l. Using a strain gage, the measured strain is obtained using the relationship:

$$\epsilon = \frac{1}{F}\frac{\Delta R}{R} \tag{7.8}$$

where F is called the **gage factor** and its value is provided by the strain gage manufacturer. We will show next how Equation (7.8) was obtained. For this, consider a bar with a cross-sectional area A, and length L. The resistance of the bar is given by

$$R = \frac{\rho L}{A} \tag{7.9}$$

where ρ is the resistivity of the material. The area can be expressed as $A = CD^2$, where C is a constant and D is the section dimension. For a square section, $C = 1$; for a circular section, $C = \pi/4$; etc. Equation (7.9) can be written as

$$R = \frac{\rho L}{CD^2} = f(\rho, L, D) \tag{7.10}$$

Differentiating Equation (7.10), we get

$$dR = \frac{\partial f}{\partial \rho}d\rho + \frac{\partial f}{\partial L}dL + \frac{\partial f}{\partial D}dD = \frac{L}{CD^2}d\rho + \frac{\rho}{CD^2}dL - 2\frac{\rho L}{CD^3}dD \tag{7.11}$$

Dividing Equation (7.11) by Equation (7.10), we get

$$\frac{dR}{R} = \frac{d\rho}{\rho} + \frac{dL}{L} - 2\frac{dD}{D} \tag{7.12}$$

but

$$\epsilon_a = \frac{dL}{L}, \epsilon_l = \frac{dD}{D}, \text{ and } \upsilon = -\frac{\epsilon_l}{\epsilon_a} \tag{7.13}$$

where ϵ_a is the axial strain, ϵ_l is the longitudinal strain, and υ is the Poisson's ratio. Replacing the terms in Equation (7.12) by the equivalent terms in Equation (7.13), we get

$$\frac{dR}{R} = \frac{d\rho}{\rho} + \epsilon_a + 2\upsilon\epsilon_a \tag{7.14}$$

Dividing Equation (7.14) by ϵ_a, we get

$$\frac{dR/R}{\epsilon_a} = \frac{d\rho/\rho}{dL/L} + 1 + 2\upsilon \tag{7.15}$$

The right-hand side of Equation (7.15) is termed the gage factor F. The value of F depends on the Poisson's ratio ν of the strain gage material as well as on how the resistivity changes with strain. For Constantan, F is 2.0. From Equation (7.15), if we replace dR by ΔR we thus get

$$\epsilon = \frac{1}{F}\frac{\Delta R}{R} \tag{7.16}$$

In most cases, the strain is a very small quantity, and the term **microstrain** is used where the strain is multiplied by one million. Strain gages are made with a standard resistance of 120 to 1000 Ω with 120 Ω being very common. When they are used, the change in resistance is small and is typically less than a fraction of one percent. Example 7.4 illustrates this point. To improve sensitivity, a bridge circuit

is typically used with a strain gage (see Section 7.10.3), since it can measure the change in resistance more precisely than a normal ohmmeter.

Strain gages are used in a variety of applications. In addition to their use in directly measuring the strain and the resulting stresses on members subjected to loading, they are also used in the construction of force and torque sensors (see next section), some types of pressure sensors, and in temperature sensors, since they can measure the elongation due to a temperature change. Due to its finite size, a strain gage measures only the average strain over an area and not the exact strain. This approximation is acceptable in cases where the strain is uniform, but it can lead to errors in cases where the strain changes considerably, such as in stress concentration areas.

Example 7.4 Strain Under Axial Loading

A 2-cm diameter steel rod is subjected to a tensile axial force of 2500 N. Assume a strain gage with a resistance of 120 Ω, and a gage factor F of 2 is used to measure the strain due to this loading. Determine the change in resistance of the gage under this loading.

Solution:

The stress due to this loading is given by

$$\sigma = \frac{F}{A} = \frac{2500}{\pi(0.01^2)} = 7.96 \text{ MPa}$$

The strain is obtained from equation (7.6), which gives

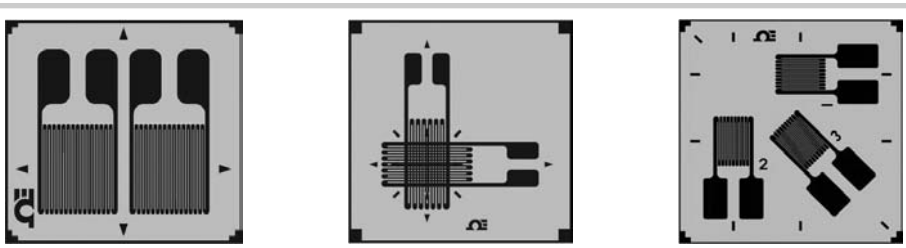
$$\epsilon = \frac{\sigma}{E} = \frac{7.96\,10^6}{200\,10^9} = 39.8 \text{ microstrain}$$

The change in resistance of the strain gage is then given by

$$\Delta R = \epsilon F R = 39.8\,10^{-6} \times 2 \times 120 = 0.00955\ \Omega$$

Note that the change in resistance is very small (~0.008%) and cannot be precisely read from an ordinary ohmmeter, which does not have such a sensitivity.

While the single, linear-pattern strain gage (Figure 7.29) is very common, strain gages are also made with many other configurations. These include the dual-grid gage (Figure 7.30(a)) that is typically used to measure bending strain, the biaxial strain gage (Figure 7.30(b)) to measure axial strain where the principal strain directions are generally known such as in pressure vessels, and the three-element rosette (Figure 7.30(c)) to measure strain in cases where the principal strain directions are not known in advance. The biaxial and the three-element rosette gages are



(a) (b) (c)

Figure 7.30

Other configurations of strain gages: (a) dual-grid gage, (b) biaxial, and (c) three-element rosette.

(Reproduced with permission of Micro-Flexitronics Ltd. (MFL), courtesy of Omega Engineering, Inc., Stamford, CT 06907 USA www.omega.com)

available with the grids stacked as in Figure 7.30(b) or in planar form (Figure 7.30(c)). The stacked configuration is more compact, but it is stiffer and less conformable than its planar counterpart.

7.7 Force and Torque Measurement

There are two methods to measure forces and torques. One is the direct comparison method, which is based on the use of some form of beam balance with known weights. The other is the indirect comparison method, which is based on the use of calibrated transducers. This textbook will focus on the second method, since the output of the transducers can be easily interfaced to a PC or a microcontroller.

7.7.1 Force Sensors

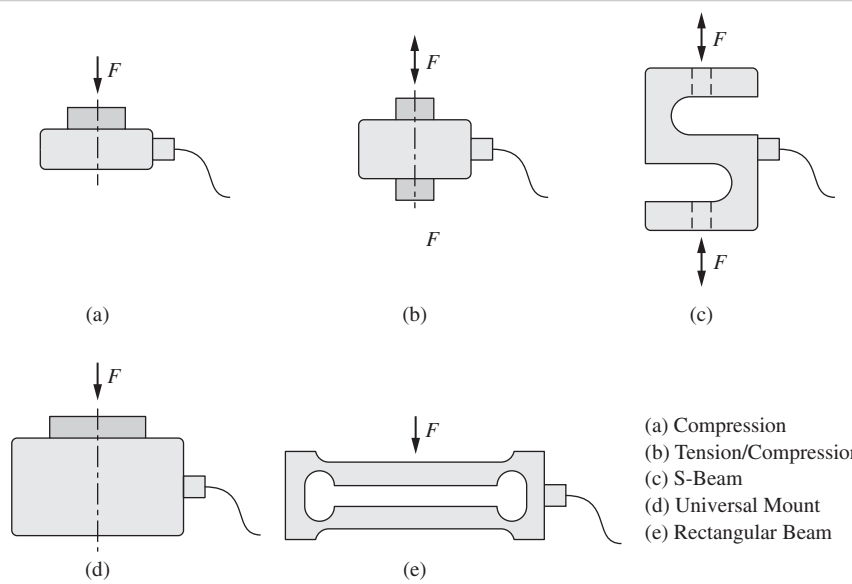
Transducer-type force sensors or load cells can be hydraulic, pneumatic, or strain-gage based. Hydraulic load cells measure the weight by sensing the pressure change in the fluid system, while pneumatic load cells measure changes in air pressure. Strain-gage types are one of the most common types. They are based on the use of an elastic element combined with one or more strain gages. The resistances of the gages are processed by a Wheatstone bridge circuit (See Section 7.10.3). **Strain gage load cells** are available in different configurations:

- Compression type
- Tension/compression type
- S-beam load cells
- Universal mounts
- Rectangular beam cells

A schematic of these configurations is shown in Figure 7.31.

Figure 7.31

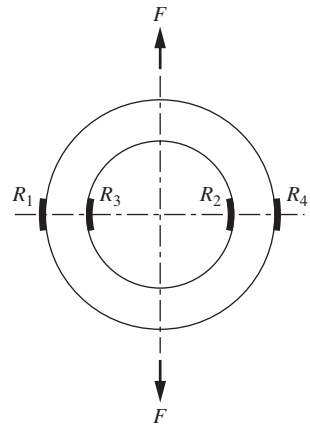
Different configurations of load cells



The compression-type cell is designed to handle compressive loads. It has a low profile, a small size, and can be made to handle high loads, but the load has to be centered. The tension/compression type can handle both compressive and tensile center loads. The S-beam load cell can also handle both compression and tensile loads, but it is better suited for harsh environments and offers good resistance to side loads. The universal mount is similar to the compression type, but it can handle off-center loads. The rectangular beam is a low-cost sensor that can handle compressive eccentric loading. This design is also known as the single-point load cell. Load cells typically use more than one strain gage to increase the sensitivity of the sensor. In many cases, four strain gages are used as seen in Figure 7.32, and they are laid out so that the change in resistance of the strain gages under the applied loading adds to improve the output of the Wheatstone bridge (see Section 7.10.3). Strain gages with a 350 Ω resistance are commonly used in load cells.

Figure 7.32

Four strain gages used in a load sensor



Manufacturers of load cells list the output of the load cells in mV/V (such as 2mV/V). Due to the use of a bridge circuit, the output is directly related to the excitation input. For example, if the supply voltage is 10 V, then the full-scale output of the load cell is 20 mV for a load cell with a 2 mV/V output rating. An external amplifier can be used to amplify the output signal of the load cell before it is read by a display device or a microcontroller. Load cells are calibrated so the output corresponds to the units of measurement of the load cell such as pounds (lbs) or Newtons (N). As with any sensor, a load cell is not 100% accurate; load cells are sensitive to thermal errors resulting primarily from the thermal expansion/contraction of the elastic element employed in the load cell.

7.7.2 Force-Sensing Resistor

A force-sensing resistor (FSR) is a sensor that uses electrical resistance to measure the force applied to the sensor. It is made using polymer film technology Figure 7.33 shows a photo of an FSR with a round active area.

Figure 7.33

Force-sensing resistor

(Courtesy of Interlink Electronics, Inc., Camarillo, CA)

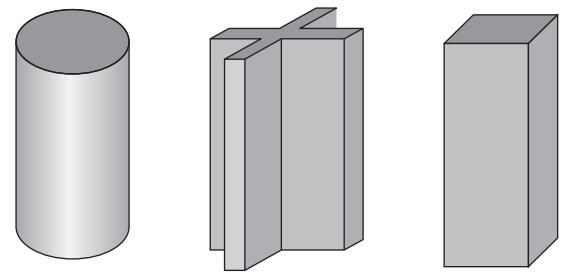
When no pressure is applied to it, the sensor has an infinite resistance. As pressure is applied to the sensor, the resistance decreases. Note that the resistance decreases nonlinearly with an increase in pressure. At maximum pressure the resistance approaches several hundred ohms. One advantage of FSRs is their low cost and simplicity. The sensor has just two leads. It is normally wired with a fixed resistor in a voltage-dividing circuit form. A disadvantage of FSRs is their low accuracy.

7.7.3 Torque Sensors

Measurement of torque is done using two different configurations of sensors. These are the reaction torque sensors and the rotating torque sensors. Both configurations are based on the use of strain gages that are mounted on elastic members. The elastic element in both configurations of torque sensors could be a

Figure 7.34

Schematic of different elastic elements used in torque sensors



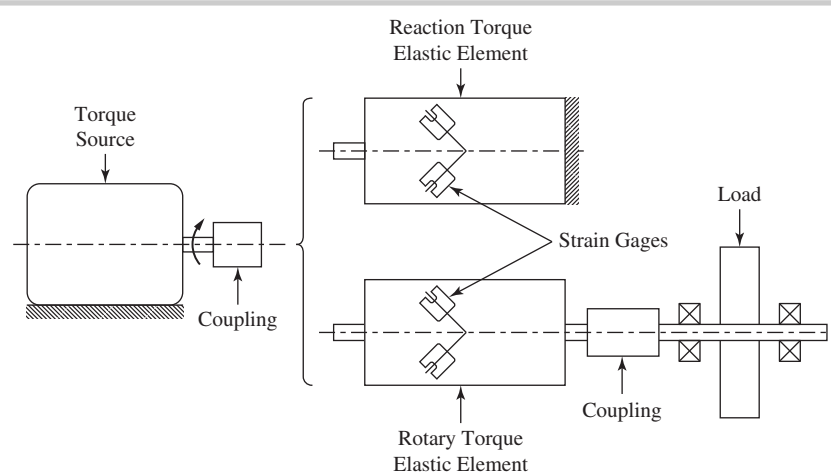
(a) Solid Shaft (b) Solid Cruciform (c) Solid Square Shaft

solid or hollow circular shaft, a solid or hollow cruciform, or a solid square shaft (see Figure 7.34). Hollow cruciform is typically used for low-torque measurement applications, while the solid circular and square shafts are used for high-torque applications.

The **reaction torque sensor** is used to measure torque in non-rotating applications. In this configuration, the sensor is stationary, and the shaft of the part of which the torque needs to be measured is connected through a coupling to the sensor. Reaction torque sensors are used, for example, to measure the motor torque output at zero speed or the starting torque. Other applications include bearing friction measurement and automotive brakes torque sensing. The **rotary torque sensor** on the other hand is used to measure torque between rotating devices. The sensor is typically mounted inline between the torque source and the load. Typical applications for rotary torque sensors include engine dynamometer testing, fan and blower testing, and clutch testing. Figure 7.35 illustrates the use of these two sensors. Similar to load cells, torque sensors give an output voltage that is proportional to the applied torque.

Figure 7.35

Illustration of reaction and rotary torque sensors



Because the sensing element is rotating in a rotary torque sensor, inertia effects are important. This is especially important during the power up and power down phases of the rotating member when the rotational speed is not constant. Thus, torque sensors with low inertia are desirable. Also, means must be provided to transmit the sensor signals from the rotating strain gage transducer to the stationary electronics. Common methods for transmitting the signals include the use of slip rings and rotary transformers. Slip rings are similar to a commutator in a brush DC motor (Chapter 8) and are suitable for low–rotation speed applications. At speeds above 5000 rpm, the noise induced from brush friction make them not very

suitable. A rotary transformer is a non-contact device and is similar to a regular transformer but the secondary coil is rotating relative the primary coil. Two rotary transformers are used: one for transmitting the supply voltage to the Wheatstone bridge circuit and the other for transmitting the output from the bridge circuit (see Figure 7.36). Some rotary torque sensors also output the rotation angle of the sensor, which is obtained from an encoder that is built into the sensor. Reaction torque sensors have a higher torque-measurement capability than rotary torque sensors, and some units are made to measure torque values up to a few million pound inches. End connections to both configurations include the use of a keyed shaft, a flange, or a spline.

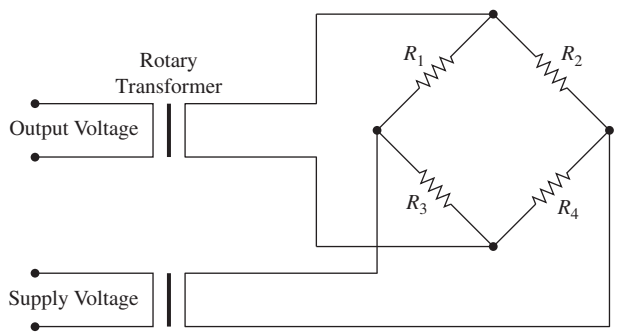


Figure 7.36

Wheatstone bridge with rotary transformers

7.8 Temperature Measurement

Temperature is a basic quantity in process control systems, and there are several types of sensors available to measure temperature. These include thermistors, thermocouples, RTD, and IC sensors. These different types will be discussed below. Table 7.3 lists and compares several properties of these sensors.

Property	Thermistor	Thermocouple	RTD	IC
Resolution	Very high	Average	High	High
Temperature Range	Small	Very broad	Broad	Limited
Output	Highly non-linear	Nonlinear	Almost linear	Linear
Accuracy	Very high	Limited	High	Limited
Ruggedness	Fragile	Very rugged	Rugged	Fragile

Table 7.3

Comparison of different temperature sensors

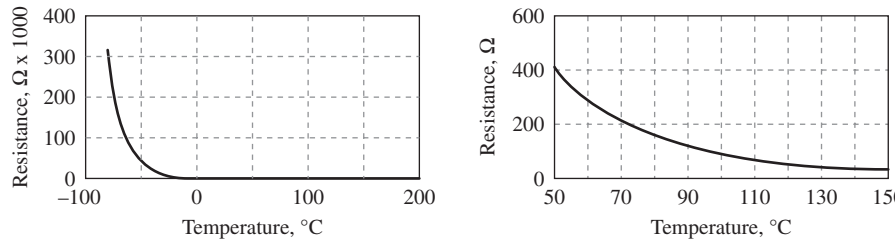
7.8.1 Thermistors

A thermistor is a resistance-based temperature measurement sensor made of a semiconductor material, and the thermistor resistance typically decreases with an increase in temperature. A thermistor has a very high sensitivity to temperature changes, but its output is highly nonlinear and is typically used over a limited temperature range that is less than 300° C. A typical resistance versus temperature plot for a thermistor is shown in Figure 7.37. Note the highly nonlinear relationship between the temperature and resistance. The resistance–temperature relationship for a thermistor can be approximately expressed by the exponential function:

$$R = R_0 e^{\beta\left(\frac{1}{T}-\frac{1}{T_0}\right)} \tag{7.17}$$

Figure 7.37

Typical resistance versus temperature plot for a thermistor



where β is a constant that depends on the thermistor material used and R_o is the resistance at the reference temperature T_o. A thermistor has the characteristic that the relationship between resistance and temperature is very precise, which allows some thermistors to have a precision of 0.05° or less.

Thermistors are available in several forms including two-lead, surface mount, and leadless chip form. Typical two-lead thermistors are shown in Figure 7.38. For the two-lead thermistor, the thermistor can be epoxy coated or glass encapsulated.

Figure 7.38

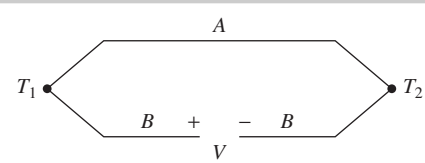
Typical leaded thermistors
(© sciencephotos/Alamy)

7.8.2 THERMOCOUPLES

Thermocouples are one of the most widely used temperature sensors. A thermocouple is a **thermoelectric** type sensor and operates on the principle that an electromotive force (EMF) is created when two junctions of different metals are operated at different temperatures. This characteristic behavior was discovered by Seebeck in 1821. Figure 7.39 illustrates this situation and shows two dissimilar metals *A* and *B* connected at two different points. If one of the junctions is at a known reference temperature, then the voltage between the nodes is a function of the difference between the temperatures of the two junctions. This fact is used to indicate the temperature of the other node.

Figure 7.39

Thermocouple junctions

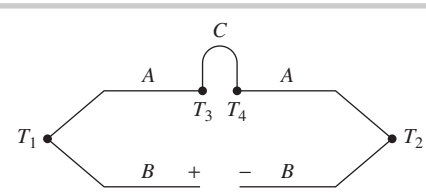


There are several laws or properties that apply to thermoelectric circuits.

Law of Intermediate Metals This law states that a third metal introduced into a thermocouple circuit will not affect the EMF output of the circuit provided that the two junctions introduced by the third metal are at the same temperature. This situation is illustrated in Figure 7.40 where a third metal *C* is introduced into

Figure 7.40

Illustration of the law of intermediate metals



the circuit. Provided that the two junctions of the metal C with metal A are at the same temperature or $T_3 = T_4$, the EMF output of the circuit is not affected. Note that this law still applies if the third metal C was introduced at either junction of metals A and B.

Application of this law permits the insertion of a measuring device into the circuit or brazing or welding of the junction without affecting the temperature measurement function of the thermocouple circuit.

Law of Homogenous Circuits This law states that if the thermocouple conductors are homogenous, then they are not affected by intermediate temperatures of the conductors away from the junctions. This situation is illustrated in Figure 7.41 where the lead wires away from the junctions have a temperature that is different from T_1 and T_2, but that does not affect the output voltage of the circuit. Application of this law permits the use of thermocouple grade extension wires and implies that shielding of lead wires is not needed in thermocouple circuits.

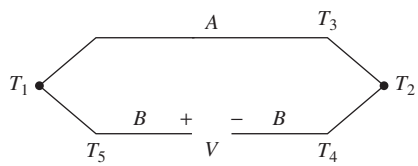


Figure 7.41

Illustration of the law of homogenous circuits

There are different types of commercially available thermocouples. These include the 'J' thermocouple (iron-constantan), the 'K' thermocouple (chromel-alumel), and the 'T' thermocouple (copper-constantan). Constantan is an alloy that is primarily made up of copper and nickel. The temperature measurement range for these types of thermocouples is shown in Table 7.4. Note that type *J* is designed to be used in reducing environments, while type *K* is used in oxidizing environments since its nickel-chromium alloy is resistant to oxidation at high temperature. Type *T* is suitable for ambient and sub-freezing environments.

	Type J	Type K	Type T
Temperature	−200 to 1000°C	−250 to 1372°C	−250 to 400°C
Range	−328 to 1832°F	−418 to 2502°F	−418 to 752°F

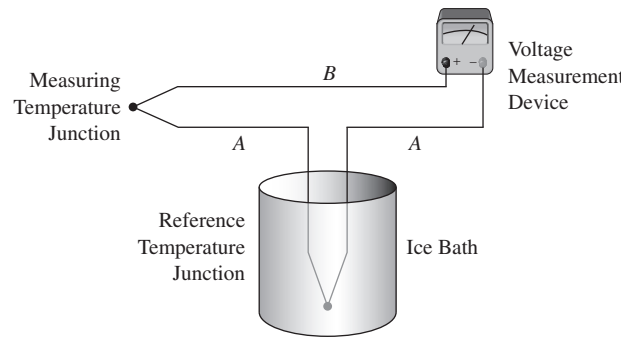
Table 7.4

Temperature range of J, K, and T thermocouples

The measuring junction of a thermocouple can be enclosed in a probe cover or can be exposed directly to the measuring atmosphere. An exposed junction has a faster response time than an enclosed one, but it is not suitable for use in a corrosive environment. When a thermocouple is used to measure temperature, it is typically wired as shown in Figure 7.42. One of the junctions is inserted into an ice bath to create a reference junction with a known temperature, while the other junction is used to measure the temperature. The leads of the thermocouple circuit are connected to a voltage-measuring device, which reads the voltage output of the thermocouple circuit, which is typically in millivolts. Some form of filtering is usually needed to reduce electromagnetic noise picked by the thermocouple wires, which act as an antenna. To improve the resolution, the output can be also amplified before being read by the voltage-measuring device. Thermocouple reading

Figure 7.42

Typical thermocouple configuration



instruments replace the ice bath mixture with a solid-state sensor or thermistor to measure the temperature of the reference junction.

The output voltage of a thermocouple does not vary linearly with temperature except for K-type thermocouples, where for average accuracy one can assume a linear relationship between output voltage and temperature over the range of 0 to 1000°C. For other thermocouples or for better accuracy, a calibration curve needs to be used to relate the temperature to voltage. The **calibration curve** takes the form:

(7.18)

$$T = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n$$

where T is the temperature in degrees C, x is the EMF voltage of the thermocouple, and $a_0, a_1, \ldots, a_n$ are the curve coefficients, which are a function of the particular thermocouple material used. The National Bureau of Standards publishes tables of these coefficients for different types of thermocouples. The order n varies from 5 to 7. For these coefficients, the accuracy varies from +/−0.1°C for type J to +/−0.7°C for type K thermocouples.

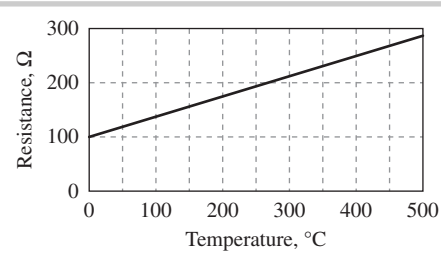
7.8.3 RTD

The resistance temperature detector (RTD) is another resistance-based temperature sensor that is based on the principle that the resistance of certain metals increases/decreases in a defined fashion with an increase/decrease in temperature. Platinum is the most commonly used material in making RTDs, but RTDs are also made using copper and nickel. Platinum offers the advantage of a broad temperature measuring range (−200 to 850°C), high temperature stability, and good accuracy.

An RTD does not have the high sensitivity of a thermistor, but its temperature resistance relationship is not highly nonlinear. Figure 7.43 shows a typical temperature resistance relationship for platinum RTD. Note that RTDs are made with different nominal resistance at 0°. A common value is 100 ohms, but RTDs with 500 or 1000 ohms are also made. RTDs are made with specific **temperature coefficient (TC)** or alpha factor. Two commonly used TC factors for platinum RTDs

Figure 7.43

Resistance versus temperature relationship for platinum RTDs



are the European Standard and the American Standard. The European Standard has a TC value of 0.00385 Ω/Ω/°C over the range of 0 to 100°C, while the American Standard has a TC value of 0.00392 Ω/Ω/°C. Using the temperature coefficient, the resistance of an RTD can be approximated as

$$R = R_0(1 + \alpha(T - T_0)) \tag{7.19}$$

where R_0 is the nominal resistance at the nominal temperature T_0. Note that the exact resistance at any temperature can be obtained from RTD manufacturers who publish tables with exact values of resistance at different temperatures for a given TC and nominal 0° resistance. RTDs are also available with different tolerance levels.

Although RTDs do not have the large temperature measuring range as thermocouples, they are more linear than thermocouples and inherently more stable, and there is no need for a reference junction. However, an RTD has a slower response than a thermocouple.

RTDs are available in different forms. These include thin film and wire wound. In the **thin-film form**, a small layer of platinum is deposited on a substrate, and then wires are attached to the substrate. The substrate is then coated in epoxy. In the **wire-wound form**, a wire coil is either inserted inside a cylindrical glass or ceramic tube, or wound around a glass or ceramic core and then covered with glass or ceramic material. RTDs are available with two-, three-, or four-wire configuration. The two-wire configuration is most sensitive to errors resulting from the additional resistance introduced by the lead wires. With the three- or four-wire configuration, a compensating bridge-type circuit (discussed in Section 7.10.3) can be constructed to compensate for the lead wire resistance.

7.8.4 IC Temperature Sensors

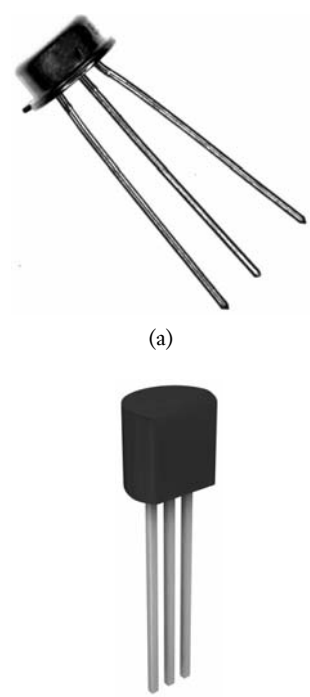
Due to advancements in integrated circuit technology, IC temperature sensors are becoming widely available. These sensors are based on transistor technology, specifically the fact that the difference in forward voltage of a silicon pn junction is directly proportional to temperature. While IC temperature sensors have a smaller temperature measurement range than thermocouples or RTDs, they, however, give an output that is linearly proportional to temperature, are inexpensive, and are fairly accurate. IC temperature sensors are available with either analog or digital output. The latter type includes an integrated A/D to convert the analog voltage or current signal into a digital signal that is transmitted using a PWM, a I^2C, or an SPI interface. An example of an analog IC temperature sensor is the **LM35C sensor** manufactured by National Semiconductor and shown in Figure 7.44. Another is the AD590 sensor manufactured by Analog Devices. The LM35C sensor can measure temperature over the range of −40 to 110°C. An example of an IC temperature sensor with digital output is the TMP05 sensor manufactured by Analog Devices, which has an accuracy of +/−1°C and provides its output in PWM format.

The LM35C sensor is available in several packages (see Figure 7.44), including the hermetic TO-46 metal can package and the TO-92 plastic package. The sensor has three leads, one for 4 to 30 VDC input power, the other for ground, and the third for the analog voltage output from the sensor. The analog output of the sensor is proportional to temperature in degrees C with sensitivity of 10.0 mV/°C. This sensor is very suitable for use with microcontrollers since it draws very little current (less than 60 μA), and its output is directly calibrated in degrees Celsius, thus avoiding any conversion operations. Note that the LM35C is designed to measure temperature in ambient air and not to be immersed in a liquid. To use in liquid, the sensor has to be encapsulated.

Figure 7.44

LM35C sensor (a) TO-46 metal can package and (b) TO-92 plastic package

(Courtesy Digi-Key Corporation)



(a)

(b)

7.9 Vibration Measurement

Vibratory motion (for further reading, see [30]) commonly occurs in machinery and flexible structures. Measurement of vibration is important for machine health monitoring of motors, pumps, fans, gearboxes, machine tool spindles, blowers, and chillers. Vibration measurement is also important in safety devices such as automotive airbags. Vibration is measured by either accelerometers or vibrometers. The two devices have a similar operating principle but differ in their natural frequency and damping. They are based on the measurement of the motion of a small, spring and damper supported mass that is placed in a housing. The mass is commonly referred to as a **seismic mass**, and the housing is attached to the structure whose vibration motion needs to be measured. The motion of the structure is transferred to the seismic mass through the support spring and damper. The motion of the seismic mass can be obtained using different transducers. These include resistance-type strain gages and piezoelectric and piezoresistive crystals. The following section illustrates the theory behind the operation of such a device.

7.9.1 Seismic Mass Operating Principle

With reference to Figure 7.45 let us assume that the support base has a displacement of y_b, and the seismic mass has a displacement of y_m. From a free-body diagram of the seismic mass, we can write the following equation of motion for the seismic mass:

$$-k(y_m - y_b) - c(\dot{y}_m - \dot{y}_b) = m\ddot{y}_m \tag{7.20}$$

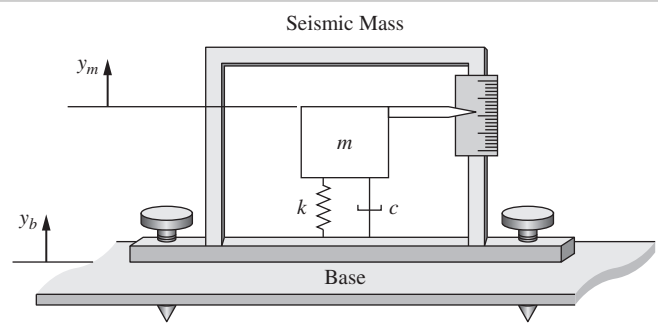


Figure 7.45
Schematic of a seismic mass

Let us define z **as the relative displacement** between the supported mass and the base or

$$z = y_m - y_b \tag{7.21}$$

Equation (7.20) can be written as

$$m\ddot{z} + c\dot{z} + kz = -m\ddot{y}_b \tag{7.22}$$

If we assume the base displacement to be a sinusoidal with amplitude Y and frequency ω as $y_b = Y\sin(\omega t)$, then Equation (7.22) can be written as

$$m\ddot{z} + c\dot{z} + kz = m\omega^2 Y\sin(\omega t) \tag{7.23}$$

Equation (7.23) is a second-order linear differential equation with a forced input. The steady-state solution of this equation is given by

$$z(t) = Z\sin(\omega t - \varphi_1) \tag{7.24}$$

where φ_1 is the phase shift angle and Z is the amplitude given by the expression

$$Z = \frac{m\omega^2 Y}{[(k - m\omega^2)^2 + (c\omega)^2]^{1/2}} = \frac{r^2 Y}{[(1 - r^2)^2 + (2\zeta r)^2]^{1/2}} \tag{7.25}$$

where r is the frequency ratio ω/ω_n and ω_n is the natural frequency given by $\sqrt{k/m}$. The phase angle φ_1 is given by the expression

$$\varphi_1 = \tan^{-1}\left(\frac{c\omega}{k - m\omega^2}\right) = \tan^{-1}\left(\frac{2\xi r}{1 - r^2}\right) \tag{7.26}$$

A plot of the ratio of Z/Y and the phase angle φ_1 as a function of the frequency ratio r is given in Figure 7.46. The plot shows that the ratio Z/Y is very dependent on the damping ratio ζ, and Z/Y approaches 1 for r greater than 3 regardless of the damping ratio ζ. The ratio Z/Y equals 1 means that the relative displacement of the seismic mass is equal to the displacement of the base. Thus, if we can measure the relative displacement, then we have a measurement of the motion of the base. This is the principal behind the operation of **vibrometers**, which provide an output that is proportional to the displacement of the base. To make r greater than 3, the natural frequency of the vibrometer has to be small. This is achieved by using a large mass m and a support spring with a small stiffness k. This results in a bulky instrument.

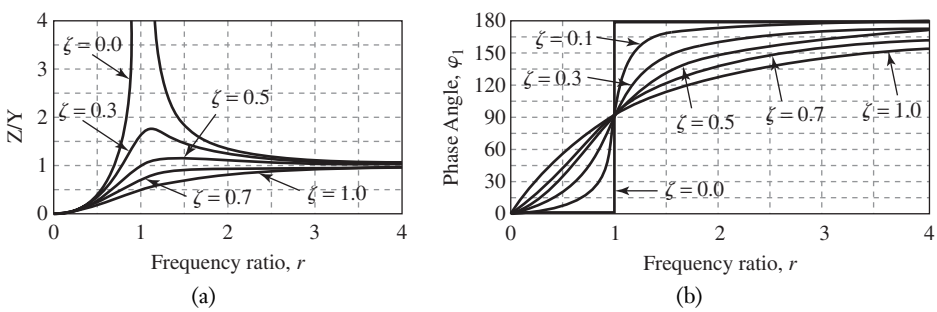


Figure 7.46

Plots of (a) Equation (7.25) and (b) Equation (7.26)

Notice the phase shift between the relative displacement and the base displacement in Figure 7.46(b). For ζ equal to 0, the phase shift is 180° for any r greater than 1. For ζ other than 0, the phase shift is dependent on the frequency ratio r. A phase shift means that relative displacement lags the base displacement by a time amount equal to φ_1/ω.

Since $r = \omega/\omega_n$, Equation (7.24) can be written as

$$-\omega_n{}^2 z(t) = \frac{-\omega^2 Y \sin(\omega t - \varphi_1)}{[(1 - r^2)^2 + (2\zeta r)^2]^{1/2}} \tag{7.27}$$

But the acceleration of the base is given by

$$\ddot{y}_b(t) = -\omega^2 Y \sin(\omega t) \tag{7.28}$$

This means that the term $-\omega_n{}^2 z(t)$ is proportional to the acceleration of the base. To make

$$-\omega_n{}^2 z(t) \cong -\omega^2 Y \sin(\omega t - \varphi_1) \tag{7.29}$$

the denominator of the right-hand side of Equation (7.27) has to be equal to 1, or

$$\frac{1}{[(1 - r^2)^2 + (2\zeta r)^2]^{1/2}} = 1 \tag{7.30}$$

Figure 7.47

A plot of the ratio $1/[(1 - r^2)^2 + (2\zeta r)^2]^{1/2}$

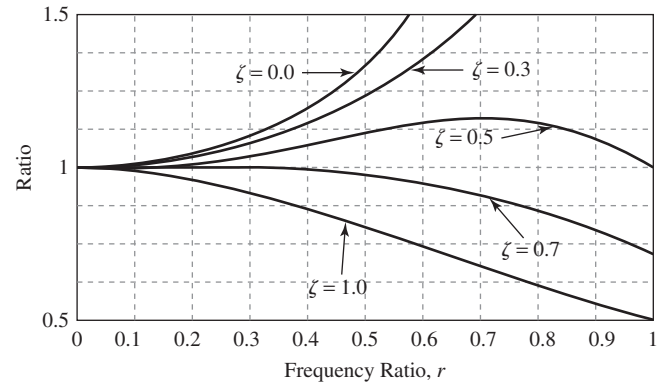


Figure 7.47 shows a plot of the left-hand side of Equation (7.30). The plot shows that the term is equal to 1 for all r less than 0.05, regardless of the value of the damping ratio ζ. For ζ equal to 0.7, the term has a value of 1 for r less than 0.2. This means that if the seismic mass is made to have a very high natural frequency, then the setup shown in Figure 7.45 can be used to directly measure the acceleration of the base through measurement of the relative displacement between the seismic mass and the base for cases where r is very small. This is the principle of operation of **accelerometers**, which give an output that is proportional to the acceleration. Note that the term ω_n^2 in Equation (7.27) is fixed for a given device and does not change with the applied frequency. The phase lag expression given by Equation (7.26) also applies for this case. To achieve low r ratio in practice, accelerometers are made with a small mass m and with a spring that has a large stiffness to result in a system with high natural frequency. Due to their compact size, accelerometers are more widely used than vibrometers, and the next section discusses piezoelectric accelerometers, one of the most commonly used types of accelerometers. Other types of accelerometers include piezo-resistive and strain-gage based accelerometers. Example 7.5 illustrates the effect of damping and frequency ratio on the output of an accelerometer.

Example 7.5 Accelerometer Error

An accelerometer with a natural frequency of 1 kHz and a damping ratio of 0.6 is used to measure the vibration of a motor rotating at 3600 rpm. The accelerometer gave a reading of 1 g. What is the actual acceleration of the motor?

Solution:

The rotation frequency of the motor is

$$3600 \times \frac{2\pi}{60} = 377 \text{ rad/s}$$

The frequency ratio $r = \omega/\omega_n$ is therefore $377/(1000 \times 2\pi) = 0.06$
From Equation (7.27), the actual acceleration is equal to

$$\text{Measured acceleration} \times [(1 - r^2)^2 + (2\zeta r)^2]^{1/2}$$

$$= 1\text{g} \times [(1 - 0.06^2)^2 + (2 \times 0.6 \times 0.06)^2]^{1/2}$$

$$\text{or } 1\text{g} \times 0.999 = 0.999\text{ g}$$

The error is therefore is 0.1%, which is small.

7.9.2 Piezoelectric Accelerometers

Certain naturally occurring materials such as quartz and Rochelle salt, and manufactured ceramic materials such as lead zirconate and barium titanate exhibit a property called the **direct piezoelectric effect** when subjected to a force or pressure. They give an electric charge proportional to the applied force. Conversely, these materials deform if they are subjected to an electric field in the direction of polarization of the material. The piezoelectric effect was discovered by the Curie brothers in 1880, and the direct piezoelectric effect is utilized in the design of accelerometers and force sensors. The converse piezoelectric effect is used in the design of piezoactuators that are used in precision positioning and machining applications. Note that the term 'piezo' is a Greek word for 'pressure'.

There are different designs of piezoelectric accelerometers. These include compression and shear type. A section view of a typical **compression-type piezoelectric accelerometer** is shown in Figure 7.48. The supported mass is sandwiched between a piezoelectric element and a compression spring. The compression spring could take the form of a bolt and a washer compressing the mass and the piezoelectric element. The accelerometer base is typically bolted or bonded to the structure whose acceleration needs to be measured, but accelerometers with magnetic bases are also available. The motion of the structure results in a motion of the supported mass, and the inertial force of the supported mass pushes against the piezoelectric element producing a charge signal. Since the supported mass is constant, the force exerted on the piezoelectric element is thus proportional to acceleration. In a **shear-type accelerometer**, the supported mass is designed to exert a shear force rather than a compressive force on the piezoelectric element. Shear-type accelerometers are used in flexible structures applications or where thermal gradients can cause distortion of the base. All of the elements of the accelerometer are housed in a metal housing, typically made of stainless steel, which is sealed to prevent the entrance of dust, water, or dirt.

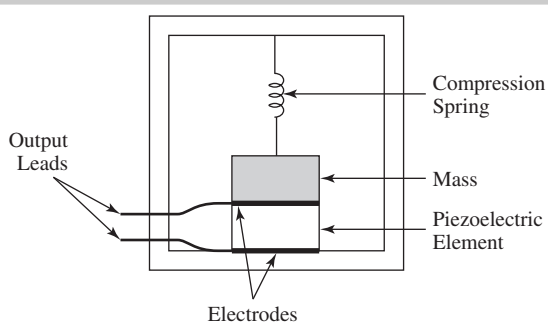


Figure 7.48

A section view of a compression-type accelerometer

Piezoelectric accelerometers have the advantage of compact size, high sensitivity, high-frequency measurement range, and rugged construction. Figure 7.49 shows several commercially available piezoelectric accelerometers. Accelerometers are available with a top or side connector location; single axis, biaxial, or triaxial measurement capability; frequency measurement range up to 25 kHz; acceleration measurement range up to 500 g; and sensitivity that ranges from 10 mV/g to 10 V/g. Example 7.6 illustrates the relationship between accelerometer sensitivity and range. Note that piezoelectric accelerometers are not very suitable for measuring low-frequency (a few Hertz) oscillations, and they give a zero output at zero frequency.

Figure 7.49

Commercially available piezoelectric accelerometers
(Courtesy of Wilcoxon Research, Inc., Germantown, MD)

Example 7.6 Accelerometer Selection

An accelerometer is needed to measure acceleration with a range up to ±20 g. If the analog output of the accelerometer is read by an A/D convertor with a ±10 volts input range, recommend a suitable sensitivity for the accelerometer.

Solution:

The voltage output of the accelerometer at the maximum desired acceleration affects the choice of the sensitivity. In this application, the accelerometer voltage output at 20 g should be equal or less than 10 V. This gives a maximum sensitivity of 10 V/20 g or 500 mV/g. Thus any accelerometer with a sensitivity of 500 mV/g or less would work in this application. Note, however, that high sensitivity is preferred since it results in a better signal-to-noise ratio.

Piezoelectric accelerometers are available with two different types of output. These include **high-impedance charge output**, and low-impedance voltage output. In the high-impedance version, the accelerometer gives a charge signal proportional to acceleration. The sensitivity of the accelerometer in this case is specified in pico (10^{-12}) Coulomb per g or pC/g. The charge signal cannot be read by a low-impedance device (such as a voltmeter) due to the large loading errors caused by the high source impedance and due to charge leakage through the load. The charge output is processed instead by a special amplifier called a **charge amplifier**, which takes the charge output from the accelerometer and produces a low-impedance analog output voltage. When using a charge amplifier, the capacitance of the cable that connects the accelerometer output to the charge amplifier input does not affect the output voltage of the amplifier. The output voltage is simply a function of the input charge and the feedback capacitance of the amplifier and is given by Equation (7.31):

$$V_{\text{out}} = \frac{q_{\text{in}}}{c_f} \tag{7.31}$$

where q_{in} is the charge produced by the accelerometer, and c_f is the feedback capacitance of the charge amplifier. This can be seen if we analyze the op-circuit shown in Figure 7.50. The input charge q_{in} is distributed as:

$$q_{in} = q_c + q_{inp} + q_f \tag{7.32}$$

But the charge is related to voltage by

$$q = cV \tag{7.33}$$

Substituting Equation (7.33) into (7.32), we get

$$q_{in} = (c_c + c_{inp})V^- + c_f V_o \tag{7.34}$$

But $V^- = 0$ since the inverting input potential is equal to the noninverting input, and the noninverting input is grounded in this circuit. Thus, we get

$$q_{in} = c_f V_o \tag{7.35}$$

This is not the case if the charge output was directly connected to a high-impedance voltage amplifier. In that case, the capacitance of the sensor, the cable, and the amplifier has to be taken into account.

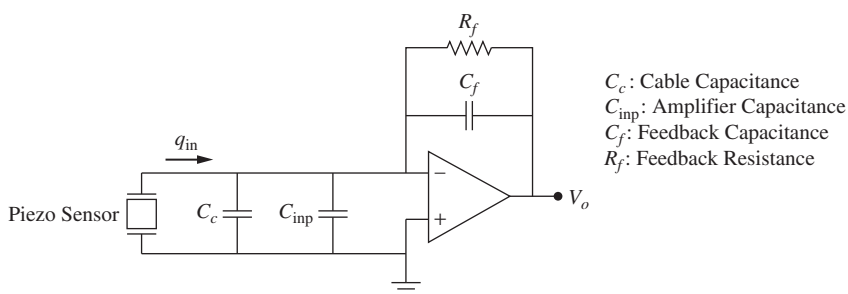


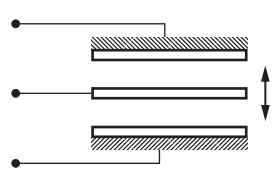
Figure 7.50
Charge amplifier wiring

Voltage-output accelerometers include a small built-in circuit in the accelerometer housing that converts the charge output into a low-impedance output voltage. Charge output accelerometers are self generating and require no external power; thus they can be used in high-temperature or high-radiation applications without any damage. This is not the case with voltage-output accelerometers, where the electronics could be damaged under such conditions.

7.9.3 Integrated Circuit (IC) Accelerometers

Integrated circuit (IC) accelerometers are low-cost sensors that are widely used in applications such as air bag deployment, computer hard drive protection, and virtual reality input devices. **IC accelerometers** are based on the use of silicon capacitive micromachined technology. The accelerometer consists of a surface mircomachined capacitive sensing cell (g-cell) and a CMOS signal conditioning circuit both housed in a single IC package. The g-cell is a mechanical structure that is constructed using wafer processing techniques. The sensing cell can be modeled as two stationary plates with a moveable center plate placed in between the fixed plates as seen in Figure 7.51. The center plate deflects in response to the acceleration of the part to which the accelerometer is attached. This deflection changes the distance between the center plate and the two fixed plates and effectively the capacitance of the two capacitors that are formed between the fixed plates and the center plate. The change in capacitance of these two capacitors is then used as a measure of the acceleration.

Figure 7.51
Model of a silicon capacitive micromachined accelerometer



IC accelerometers are designed to be mounted on circuit boards, and they are available with different sensitivities and acceleration measurement ranges. Figure 7.52 shows a photo of the MMA1250EG z-axis sensor made by Freescale Semiconductor. This sensor has a sensitivity of 380 to 420 mV/g and a measurement range of ±5g.

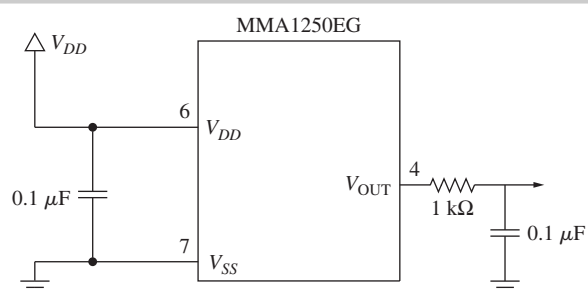
Figure 7.52

MMA1250EG sensor
(Jouaneh, University of Rhode Island)

The wiring diagram for this sensor is shown in Figure 7.53 which shows that only three wires are needed to be interfaced to the sensor: supply voltage, ground, and output line. The sensor simply needs a DC supply voltage in the range of 4.75 to 5.25 VDC. With a 5 VDC supply voltage, the sensor output is about 2.65 V at zero gravity. An advantage of IC accelerometers is their low cost, their low current consumption (about 2 mA), their built-in signal conditioning, and their linear output. IC accelerometers can also be economically interfaced to microcontrollers since their output is DC voltage in the range that can be directly read by an A/D convertor, and they do not require a charge amplifier. However, their temperature operating range is more restrictive than charge output piezoelectric accelerometers, and they are not as rugged.

Figure 7.53

Wiring diagram for MMA1250EG sensor



7.10 Signal Conditioning

In many situations, the raw output of the sensor may not be in a form suitable to be interfaced to a measurement device or an A/D convertor, and some form of signal conditioning is needed to be applied to the sensor output. These signal conditioning operations include filtering, amplification, or using a bridge circuit. These operations are discussed next.

7.10.1 Filtering

In dynamic measurements, the output of a sensor or a transducer consists of many frequency components. The output can also be corrupted with noise. The **noise** can be from external disturbances such as heat or vibration, imperfections in the sensor component materials, or from external signal interference. The noise reduces the sensor resolution and worsens the repeatability error. **Filtering** is the

process of attenuating unwanted components or noise from the sensor output and allowing other components to pass. We will discuss different types of filters. These include low-pass, high-pass, notch, and bandpass. The naming of these filters is in reference to desired frequency characteristics of the filter.

A filter can be implemented using analog circuit components such as op-amps and capacitors and is thus referred to as an **analog filter**, or using computer code in a digital signal processer chip and thus referred to as a **digital filter**. Digital filters offer flexibility over analog filters since changing the filter characteristics involves only code change. However, analog filters are inexpensive and more robust. Analog filters can be classified as passive or active. A passive filter does not require any external power to operate. An example of a passive analog filter is the first-order RC-circuit filter. An active filter, on the other hand, requires external power to operate. Active filters use components such as op-amps.

The filtering action is best described using frequency response plots and transfer function concepts. Frequency response plots are reviewed in Appendix B. A **frequency response plot** shows the output–input relationship for the filter as a function of frequency. The plot has two components: magnitude and phase. The magnitude is the ratio of the output signal to the input signal, while the phase is the time shift between the output and the input signals. A positive phase angle means that the output signal leads the input signal, while a negative phase shift means that the output signal lags the input signal. The magnitude plot is normally shown in units of dB, where 1 dB $= 20 \log_{10}$(output/input). The ideal magnitude frequency response characteristics for the low-pass, high-pass, bandpass, and notch filters are shown in Figure 7.54, where f_c is the filter corner frequency. The figure shows that over the frequencies at which the filtering is active the output signal is zero. In reality, these ideal frequency characteristics cannot be achieved. First, filters with zero passing are difficult to construct. Second, the sharp transitions shown in the figure cannot be achieved.

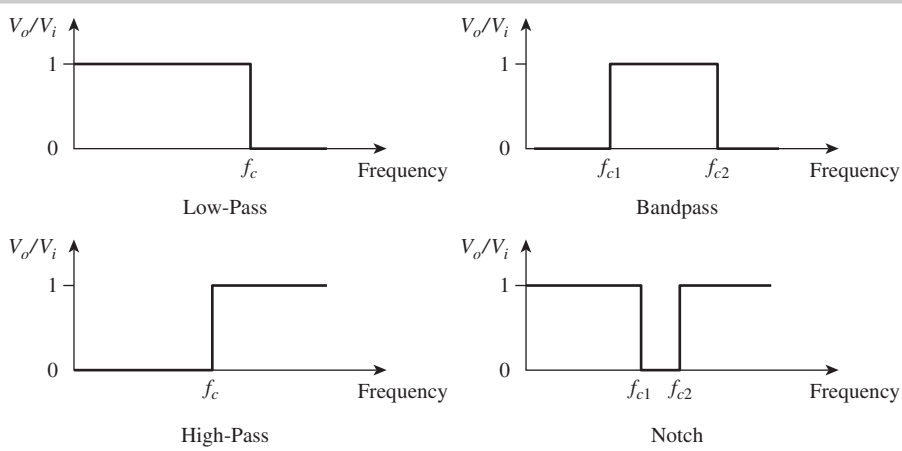


Figure 7.54

Ideal magnitude frequency response characteristics of various filters

Low-pass filters are used to attenuate frequency signals above the corner frequency f_c such as the 60-Hz interference signals from AC power operated equipment. These filters are commonly used to process signals from tachometer and temperature sensors. There are many forms of low-pass filters, so we will restrict our discussion to the first-order, low-pass filter given by the transfer function

$$H(s) = \frac{1}{\tau s + 1} = \frac{1}{\frac{s}{2\pi f_c} + 1} \tag{7.36}$$

where τ is the time constant of the filter. The **time constant** τ in seconds and the **corner frequency** f_c in Hz are related by

$$f_c = \frac{1}{2\pi\tau} \tag{7.37}$$

The order of the filter refers to the highest power in the denominator of the filter transfer function. A first-order filter is also known as a single-pole filter, since the characteristic equation of the filter has only one pole or one root. The magnitude and phase frequency response plots of this filter are shown in Figure 7.55. The magnitude plot shows that the filter output-to-input magnitude has a unity gain before the corner frequency at ω_c, where ($\omega_c = 1/\tau$), but the gain decreases slowly as the frequency approaches the corner frequency. At the corner frequency, the output magnitude is −3 dB with the output at 70.7% of the input. After the corner frequency, the magnitude starts dropping much faster at the rate of 20 dB/decade, where a decade means a factor of 10 increase or decrease in the frequency. Example 7.7 illustrates the computation of the filter output at any frequency. The phase plot shows that the filter has nearly zero phase lag at low frequencies. At the corner frequency, the phase angle is −45 degrees, and the phase angle approaches −90 degrees for frequencies much higher than the corner frequency.

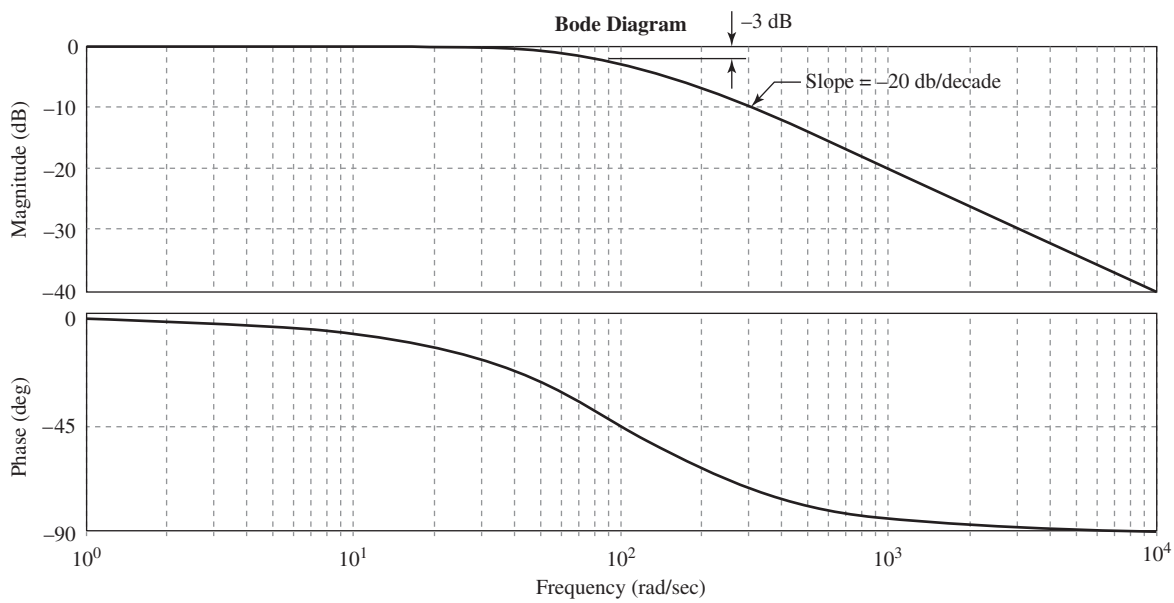


Figure 7.55

Magnitude and phase for first-order low-pass filter (ω_c = 100 rad/s)

Example 7.7 Filter Output

If the corner frequency of the low-pass, first-order filter is 100 Hz, determine the filter output at 500 Hz.

Solution:

We will obtain the output using two methods. The first is an exact method that is based on plugging the value of the frequency into the transfer function. The second is based on using the slope information.

From Equation (7.36), if we substitute $s = j\omega$, where ω is the circular frequency, we obtain

$$H(s) = \frac{1}{\tau j\omega + 1} = \frac{1 - \tau j\omega}{1 + \tau^2\omega^2}$$

and the magnitude of the above complex expression is given by

$$\left|\frac{V_o}{V_{in}}\right| = \frac{1}{\sqrt{1 + \omega^2\tau^2}}$$

Replacing ω with $2\pi(500)$ and τ with $1/(2\pi 100)$ in the above expression, we obtain an output-to-input ratio of 0.196, or the output is 19.6% of the input.

Using the slope information, we first determine the number of decades between the 500 Hz and the 100 Hz frequency by taking the logarithm of the ratio of the two frequencies or $\log_{10}(500/100) = 0.699$. A first-order filter has a slope of −20 dB/decade, so at 500 Hz, the filter output is −13.98 dB (0.699 × −20 dB) or 20% of the input. Thus, there is an 80% attenuation at a frequency of 500 Hz. The result is very close to that using the transfer function method.

Note that if we had used a Butterworth filter with a slope of −60 dB/decade, then the attenuation at 500 Hz would have been 99.2%.

A simple passive low-pass filter can be easily constructed using a resistor and a capacitor circuit as shown in Figure 7.56(a). The filter time constant τ is the product of the resistance and the capacitance or $\tau = RC$. An active low-pass filter is implemented using op-amps as shown in Figure 7.56(b), where the filter time constant is given as $\tau = R_2C$. Note for the active filter, the filter gain is given by $(-R_2/R_1)$.The attenuation rate of the first-order low-pass-filter is, however, not high. It is desirable to have high attenuation rates such as −60 dB/decade. Such an attenuation rate can be achieved by other types of filters such as Butterworth and Chebyshev. The details of these and other filter types can be found in many signal-processing texts such as [14].

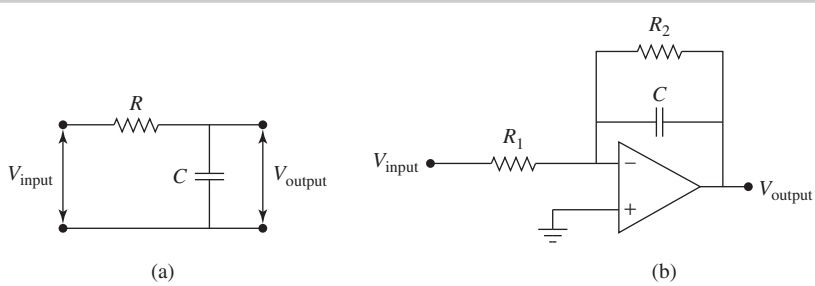


Figure 7.56

Circuit for (a) passive low-pass RC filter and (b) active low-pass filter

If the sensor data is available digitally, then one can use a digital filter instead. A **digital low-pass** filter can be implemented using Equation (7.38).

$$y(k) = (1 - \alpha)x(k) + \alpha y(k - 1) \tag{7.38}$$

where $y(k)$ is the output sequence, $x(k)$ is the input sequence, k is the current index, and α is a factor that is dependent on the filter corner frequency f_c and the sampling time T. Alpha is given by:

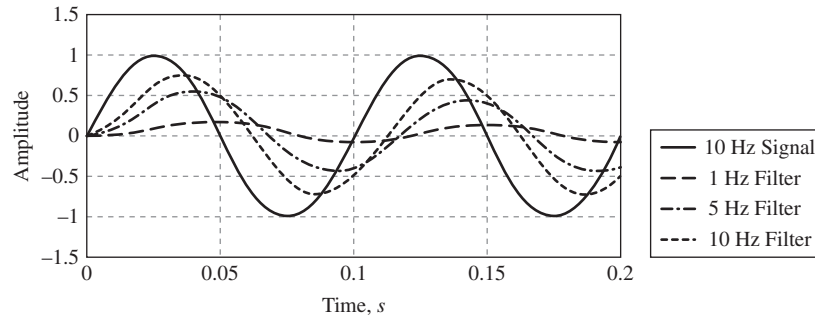
$$\alpha = e^{(-2\pi f_c T)} \tag{7.39}$$

For example, if $f_c = 10$ Hz, and T is 1 ms, then α is 0.9391, and the digital filter equation is

$$y(k) = 0.0609x(k) + 0.9391y(k - 1) \tag{7.40}$$

Figure 7.57

Digital filter output for different corner frequencies



The above iterative equation can be easily implemented in code using a *For-Loop* or similar construct. Figure 7.57 shows the filter output for a 10 Hz sinusoidal input signal for three corner frequencies: 1, 5, and 10 Hz and a sampling time of 1 ms. Notice how the output magnitude is smaller than the input magnitude for all the three cases.

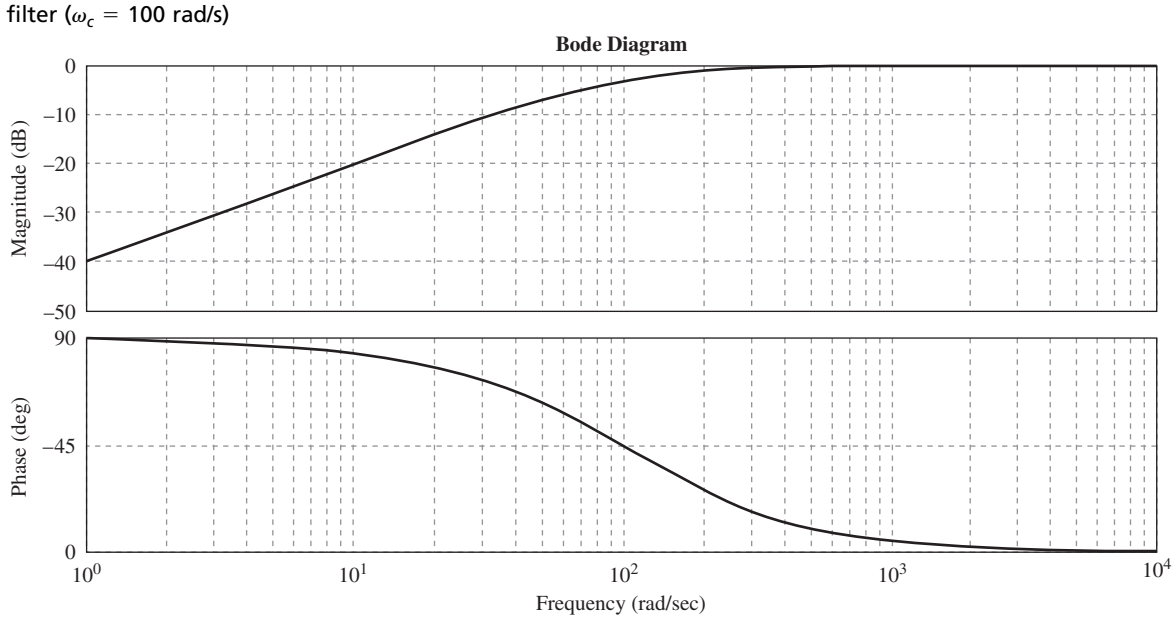
High-pass filters are used to attenuate frequency signals below the corner frequency, f_c. These filters are used attenuate dc and low-frequency components from a signal so the high-frequency components of the signal become more distinguished. Similar to the low-pass case, there are many forms of high-pass filters, so we will restrict our discussion to the first-order, high-pass filter given by the transfer function

$$H(s) = \frac{\tau s}{\tau s + 1} \tag{7.41}$$

where τ is the time constant of the filter. The magnitude and phase frequency response plots of this filter are shown in Figure 7.58, and they are opposite to those of the low-pass filter. The plot shows that the filter has nearly a unity gain above the corner frequency ω_c. Before this frequency, the magnitude increases at the rate of 20 dB/decade.

Figure 7.58

Magnitude and phase for first-order, high-pass filter (ω_c = 100 rad/s)



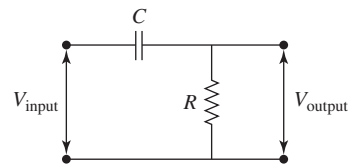


Figure 7.59

Circuit for a first-order, high-pass filter

A simple high-pass filter can also be constructed using a resistor and capacitor circuit as shown in Figure 7.59. The circuit is similar to the low-pass filter, but the resistor and the capacitor locations are interchanged.

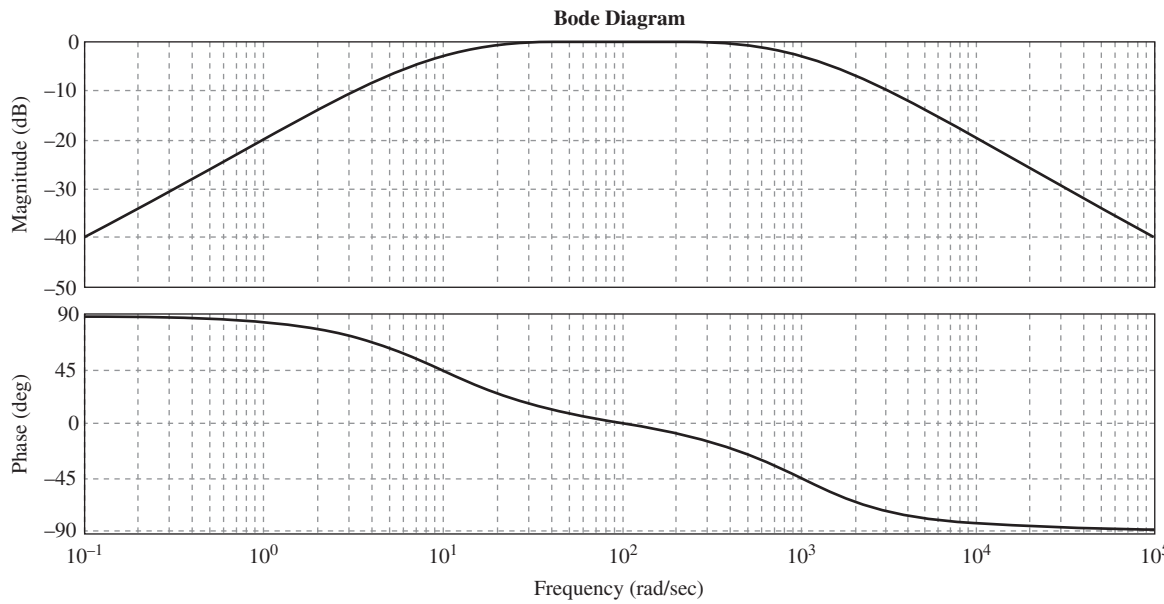
Bandpass filters are used to allow signals in a certain frequency range and to attenuate all signals outside of this range. The two corner frequencies that define this frequency range are f_{cl} and f_{ch}. A bandpass filter is a cascade combination of a low-pass and a high-pass filter, with the transfer function given as

$$H(s) = \frac{\tau_1 s}{(\tau_1 s + 1)(\tau_2 s + 1)} \tag{7.42}$$

where $\tau_1 = 1/(2\pi f_{cl}) = 1/\omega_{cl}$ and $\tau_2 = 1/(2\pi f_{ch}) = 1/\omega_{ch}$. The magnitude and phase frequency response plots of this filter are shown in Figure 7.60. The magnitude plot shows that the filter has nearly a unity gain inside the desired frequency band. Outside this band, the magnitude increases or drops at the rate of 20 dB/decade. The phase angle is close to zero around the center of the pass band, but approaches +/−90 degrees away from the frequency band.

Figure 7.60

Magnitude and phase for bandpass filter (f_{cl} = 10 rad/s, f_{ch} = 1000 rad/s)



Notch filters are used to attenuate a narrow band of frequencies from a signal. For example, if we know that the noise is originating at a particular frequency such as 60 Hz, then we can design a filter to eliminate this noise. A notch filter is a second-order system and has the following transfer function:

$$H(s) = \frac{\tau^2 s^2 + 1}{\tau^2 s^2 + 4\tau s + 1} \tag{7.43}$$

where, as before, the time constant τ and the notch frequency f_c are related by Equation (7.37). The magnitude and phase frequency response plots of this filter

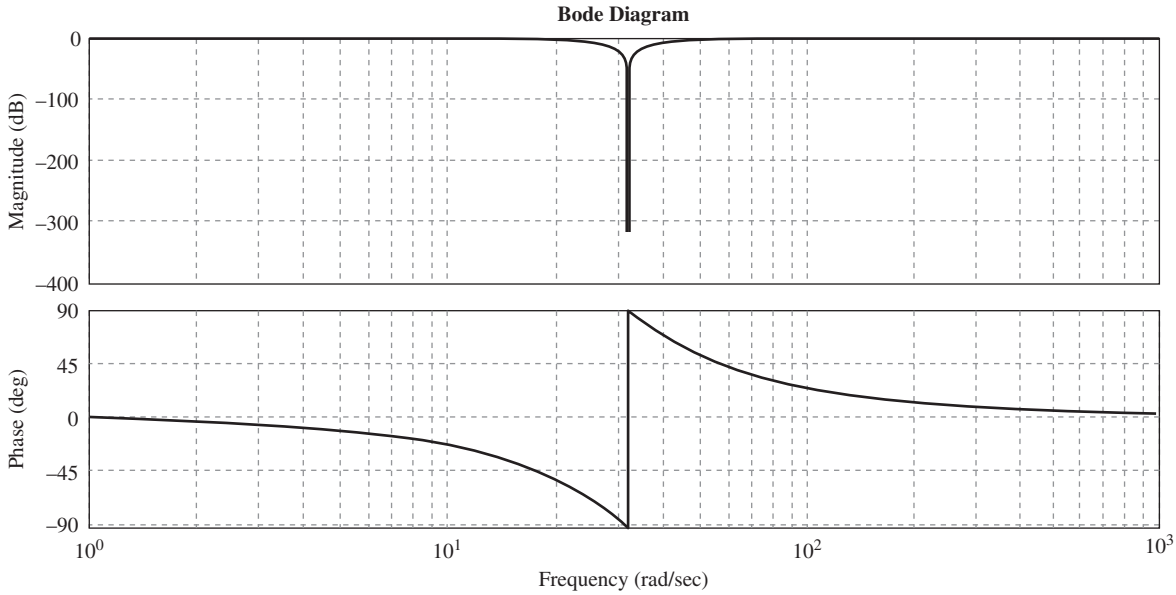


Figure 7.61

Magnitude and phase for notch filter

are shown in Figure 7.61. The magnitude plot shows that the filter has zero output at the notch frequency (10π rad/s), with a sharp roll down and roll up at either side of the notch frequency. The phase plot shows that away from the notch frequency, the phase angle is zero. Note that if the disturbance frequency is different from f_c, then the disturbance frequency is not completely eliminated.

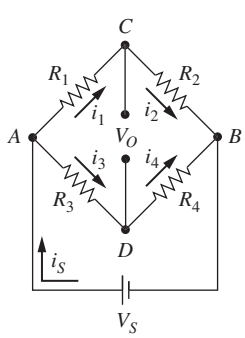
7.10.2 Amplification

In many cases, a sensor output voltage is too small to provide a meaningful resolution and some form of amplification is needed to amplify the output voltage. To have a good accuracy, the impedances of the sensor and the amplifier have to be taken into account in selecting an amplifier. In general, it is desirable for a sensor to have a very small output impedance and for the amplifier to have a very large input impedance. However, amplifiers are not ideal devices, but have finite impedances that could affect the accuracy of the amplification process. When the output impedance of the sensor is not very small compared with the input impedance of the amplifier, the input voltage seen by the amplifier is different than the open circuit voltage of the sensor. This condition is called loading and was discussed in detail in Chapter 2.

Amplifiers are available as single IC circuits, or can be built using a combination of discrete elements such as transistors, diodes, and resistors, or can be built using op-amps. Op-amp amplifiers have high input impedances and thus are very suitable for use in amplification circuits. Chapter 2 discussed different types of op-amp circuits. These included simple amplification, amplification with either integration or differentiation, voltage following, and differential amplification. The next chapter discusses amplifiers that are used for motor control applications.

Figure 7.62

Wheatstone bridge circuit



7.10.3 Bridge Circuits

A bridge circuit is used to improve the accuracy and sensitivity of the output of certain sensors. It is commonly used to process signals from resistive, capacitive, and inductive type sensors. Consider the bridge circuit shown in Figure 7.62 made of resistive elements with a constant DC voltage applied to it. This four-arm bridge circuit is commonly known as the **Wheatstone bridge** and is commonly used to process the signal output from strain gages and resistance-based thermal sensors. The strain

gage resistance or the temperature sensor resistance form one arm of the bridge. External power from a battery or a power supply is applied to two opposite vertices of the bridge, while the output of this circuit is read from the remaining two opposite vertices. Normally, the resistances of the arms of the bridge are chosen so that the bridge is initially balanced (output voltage is zero). A bridge circuit can then be used in one of two ways. In the first way (the **null method**), when one of the resistances changes (such as R_1) due to a change in the variable that is being measured, another resistance is changed (such as R_2) to bring the output of the bridge back to zero. This assumes that means are available to adjust the resistance R_2. This technique is only used with slowly varying resistance changes. In the other way (**deflection method**), the three resistances of the bridge are fixed, and only one is changed. When one of the resistances of the bridge changes, the change in the output of the bridge is used as a measure of the resistance change. The second way is suitable for dynamic measurements. The bridge is known as a deflection bridge in this mode of operation.

As a first step, let us determine the conditions that make the output of the bridge to be balanced. If the bridge is balanced, then the voltage V_O is zero. The voltage drop across R_1 is given by

$$V_1 = \frac{R_1}{R_1 + R_2} V_S \tag{7.44}$$

because the resistances R_1 and R_2 act as a voltage-dividing circuit. In a similar fashion, the voltage drop across R_3 is given by

$$V_3 = \frac{R_3}{R_3 + R_4} V_S \tag{7.45}$$

V_O is then given as $V_1 - V_3$ or

$$V_O = \left(\frac{R_1}{R_1 + R_2} - \frac{R_3}{R_3 + R_4} \right) V_S = \left(\frac{R_1 R_4 - R_2 R_3}{(R_1 + R_2)(R_3 + R_4)} \right) V_S \tag{7.46}$$

For V_O to be zero, the numerator in Equation (7.46) has to be zero or

$$\frac{R_1}{R_2} = \frac{R_3}{R_4} \tag{7.47}$$

Thus if the resistances of the bridge follow the ratios given by Equation (7.47), then the bridge will be balanced and the output voltage V_O will be zero. Notice that the balance condition is independent of the magnitude of the supply source.

Now let us assume that the resistance R_1 of the bridge has changed by a small amount δR_1. We want to determine the resulting change in the bridge output due to this change. From Equation (7.46), the output of the bridge will be

$$V_O + \delta V_O = \left(\frac{R_1 + \delta R_1}{R_1 + \delta R_1 + R_2} - \frac{R_3}{R_3 + R_4} \right) V_S \tag{7.48}$$

The change in output is then given by subtracting Equation (7.46) from (7.48) or

$$(V_O + \delta V_O) - V_O = \left(\frac{R_1 + \delta R_1}{R_1 + \delta R_1 + R_2} - \frac{R_1}{R_1 + R_2} \right) V_S \tag{7.49}$$

If we let $R_1 = R_2 = R_3 = R_4 = R$ and $\delta R_1 = \delta R$, Equation (7.49) becomes

$$\delta V_O = \frac{\delta R}{4R + 2\delta R} V_S \tag{7.50}$$

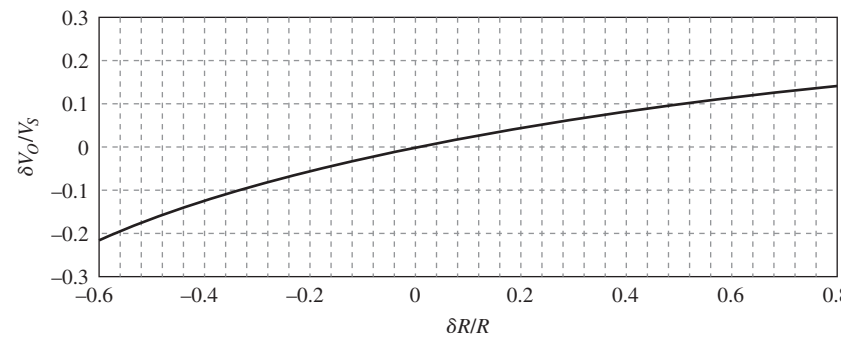
or

$$\frac{\delta V_0}{V_S} = \frac{\delta R/R}{4 + 2\delta R/R} \tag{7.51}$$

Equation (7.51) is plotted in Figure 7.63, and it shows the output change of the bridge is nonlinear especially for negative $\delta R/R$ values.

Figure 7.63

Plot of Equation (7.51)



However, if we assume that $\delta R/R$ is small, then Equation (7.51) can be approximated as

$$\frac{\delta V_0}{V_S} = \frac{\delta R}{4R} \tag{7.52}$$

The factor ¼ in Equation (7.52) is the sensitivity of the voltage output change due to resistance change. Equation (7.52) can be solved to get the resistance change as a function of the voltage output change of the bridge, or

$$\frac{\delta R}{R} = \frac{4\delta V_0}{V_S} \tag{7.53}$$

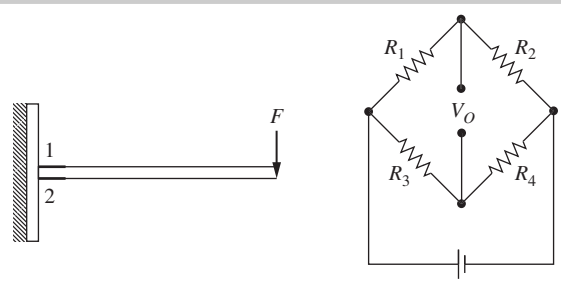
In some cases, more than one resistance in the Wheatstone bridge can be active. From Equation (7.46), we can derive an expression for δV_O using differentiation rules from calculus. This gives

$$\frac{\delta V_0}{V_s} = \frac{(R_2\delta R_1 - R_1\delta R_2)}{(R_1 + R_2)^2} - \frac{(R_4\delta R_3 - R_3\delta R_4)}{(R_3 + R_4)^2} \tag{7.54}$$

For example, we can mount two strain gages on a beam subjected to bending to increase the measurement sensitivity as shown in Figure 7.64. When the beam end deflects downward under the applied loading, the resistance of the upper gage will increase and the resistance of the lower gage will decrease by the same amount. For

Figure 7.64

Beam with two strain gages



the case where $R_1 = R_2 = R$, we have here $\delta R_1 = \delta R$ and $\delta R_2 = -\delta R$. Substituting into Equation (7.54), we get

$$\frac{\delta V_0}{V_s} = \frac{R\delta R + R\delta R}{(2R)^2} = 2\frac{\delta R}{4R} \tag{7.55}$$

Notice here how the bridge output is now double the case if we had just used a single strain gage (Equation 7.52). We can generalize this case and rewrite Equation (7.55) in the form:

$$\frac{\delta V_0}{V_s} = K\frac{\delta R}{4R} \tag{7.56}$$

where K is the **bridge constant** and is defined as

$$K = \frac{\text{Output of bridge}}{\text{Output of bridge if only one strain gage is active}} \tag{7.57}$$

Examples 7.8 and 7.9 illustrate the use of bridge circuits.

Example 7.8 Strain Gage Output

If the strain gage considered in Example 7.4 was connected to a Wheatstone bridge with Vs = 10 V, determine the output change of the bridge for the loading given in that example.

Solution:

For the 120 Ω gage, the change in resistance was found to be 0.00955 Ω. Using Equation (7.52) (since $\delta R/R$ is very small), the change in the bridge voltage output is given as

$$\delta V_O = \frac{\delta R}{4R}V_S = \frac{0.00955}{4 \times 120}10 = 0.199\ \text{mV}$$

Note if the bridge was used in the deflection mode, this output can be amplified 1000 times or more before being read by an A/D due to the absence of a large fixed voltage component. This is one of the main reasons why a Wheatstone bridge is used to process the output of a strain gage. This is not the case if the gage was one of the resistors in a two-resistor voltage-dividing circuit, as shown in Figure 7.65. The presence of the initial voltage drops across the gage resistor makes it very difficult to measure the minute change in voltage drop due to the applied loading.

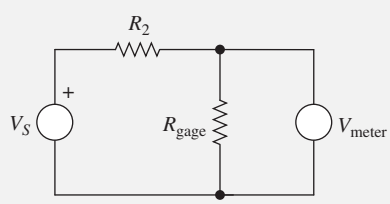


Figure 7.65

There are several varieties of bridge circuits. The bridge circuit that was previously analyzed is known as a **constant-voltage resistance bridge** due to the use of a constant voltage source as the supply source. If the supply source is a constant-current source (through the use of a current-regulated DC power source), then the bridge circuit is known as a **constant-current bridge circuit**. The constant-current bridge has a better linearity than the constant-voltage bridge, and

Example 7.9 Bridge Circuit with an RTD Sensor

A Wheatstone bridge similar to that shown in Figure 7.62 is used to process the output from a two-lead platinum RTD. The RTD has an alpha of 0.00385 Ω/Ω/°C and a nominal resistance of 100 Ω at 0°C. The bridge was initially balanced when the temperature is 20°C. What will be the output voltage of the bridge if the sensor temperature was increased by 10°C? Assume that the bridge supply voltage is 10 VDC.

Solution:

The resistance of an RTD sensor is given by Equation (7.19). At $T = 20°C$, the resistance is $R_1 = R_o(1 + \alpha(T - T_o)) = 100(1 + 0.00385(20 - 0)) = 107.7\ \Omega$. Since the bridge is balanced at this temperature, we will assume the other three arms to have the same resistance.

For a 10° temperature increase, the change in resistance is equal to ($R_o \alpha \Delta T$) or $\delta R = 100 \times 0.00385 \times 10 = 3.85\ \Omega$. Using the linear relationship first (Eq. (7.52)), the output of the bridge circuit will be

$$\delta V_O = \frac{\delta R}{4R} V_S = (3.85 \times 10/(4 \times 107.7)) = 89.4 \text{ mV}$$

If we have used the exact relationship (Eq. (7.50)), the output will be

$$\delta V_O = \frac{\delta R}{4R + 2\delta R} V_S = \frac{3.85}{4 \times 107.7 + 2 \times 3.85} 10 = 87.8 \text{ mV}$$

The error due to the linear approximation is 1.9%.

the output voltage change for the case where $R_1 = R_2 = R_3 = R_4 = R$, and $\delta R_1 = \delta R$, (see Problem 7.12) is given by the relationship

$$\delta V_0 = \frac{\delta R R}{4R + \delta R} i_S \tag{7.58}$$

where i_S is the supply current. Similarly, an **AC bridge circuit** is one in which the supply voltage is alternating. With an AC supply, the bridge can be used to measure inductance, capacitance, or resistance. A bridge circuit can be used to compensate for lead wire resistance. This is needed in situations where the sensor is located away from the signal conditioning equipment and long leads are used to connect the sensor. The RTD temperature sensor which was discussed in Section 7.8 is available with two- or three-lead configuration. In the **two-lead configuration** (see Figure 7.66(a))

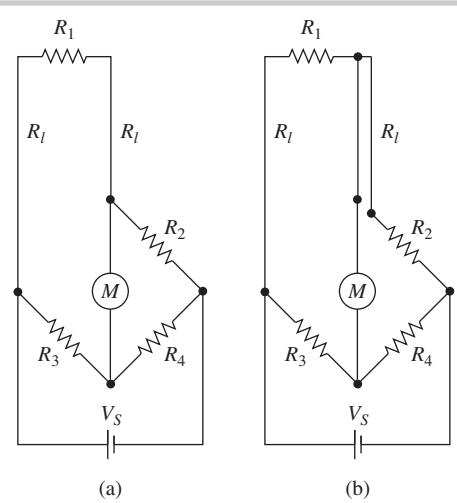


Figure 7.66
(a) Two-lead and (b) three-lead connections to a bridge

the bridge output is sensitive to the variation of the resistance of the lead wire due to environmental influences. However, in the **three-lead configuration** (Figure 7.66(b)), any environmental effect on the resistance of the two lead wires will equally effect both "legs" of the bridge assuming the same wire type, wire thickness, and wire length is used in both leads; thus, the sensor is insensitive to lead wire effects.

A bridge circuit can also be used to **compensate for temperature change** in strain gages. This is done by using a dummy strain gage located next to where the actual measurement is made and placed on material very similar to the test material. The connection diagram is shown in Figure 7.67. Any temperature change will change the R_1 and R_2 resistances by the same amount, making the circuit just sensitive to variation in the R_1 resistance due to loading.

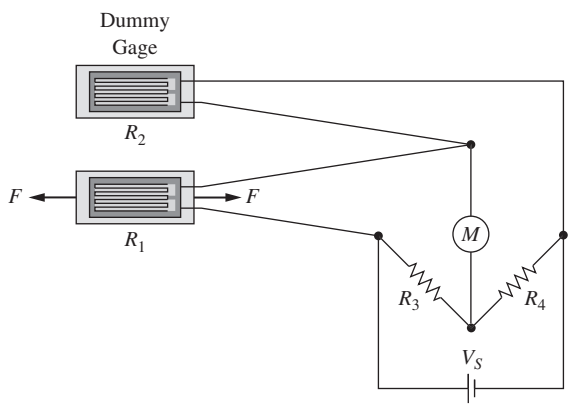


Figure 7.67

Dummy gage for temperature compensation

7.11 Sensor Output

We have seen that some sensors produce an analog signal while others produce a digital signal as their output. Many sensors produce or can be made to produce a current output commonly known as **4-20 mA** as their output. This section will discuss this output method. In this output method, the sensor uses a current transmitter, which produces a current signal proportional to pressure, temperature, or any other physical quantity that is being measured by the sensor. The output current from the transmitter varies from 4 mA at one end of the measurement to a maximum of 20 mA at the other end.

Current transmitters are available in different forms, with the **two-wire transmitter** being very commonly used. A schematic of a wiring diagram that uses a two-wire transmitter is shown in Figure 7.68. An external DC voltage is supplied to the transmitter through a wire with a lead resistance R_L. The output current from the transmitter goes through another wire with a resistance R_L, and then through a receiver resistor with a resistance R_C. The current produced by the transmitter passes through all the elements in the circuit. Since it is easier to measure voltage than current, the voltage drop across the **precision receiver resistor** is measured and is

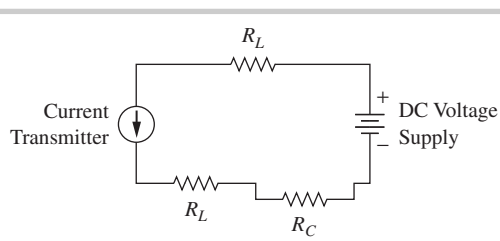


Figure 7.68

Wiring for a two-wire current transmitter

used as an indication of the sensor output. For example, if a 250 Ω receiver resistor is used, then the voltage across the receiver resistor will vary between 1 and 5 V.

An advantage of the current output method is its insensitivity to supply-voltage variation and to noise in the circuit. Because the current transmitter has a high impedance (Mega ohms (MΩ)), any noise voltage in the circuit is dropped mostly across the current transmitter with very little voltage dropped across the receiver resistor. Because of this feature, current transmitters are used in many industrial applications to transmit sensor output over long distances.

7.12 Chapter Summary

This chapter discussed the different types of sensors used in mechatronic and measurement applications. The chapter started by discussing the static and dynamic characteristics that characterize a sensor performance. It then discussed and explained the principle of operation for many types of sensors used in various measurement applications. These sensors include the following.

- Potentiometers, LVDTs, encoders, and resolvers for displacement measurement.
- Hall-effect, inductive-type, ultrasonic, and contact-type proximity sensors for proximity measurement.
- Tachometers and encoder-type sensors for speed measurement.
- Strain-gage sensors for strain, force, and torque measurement, and force-sensitives resistors for force measurement.
- Thermistors, thermocouples, RTDs, and integrated circuit sensors for temperature measurement.
- Piezoelectric and integrated circuit sensors for vibration measurement.

The chapter also discussed the use of signal-conditioning operations (such as filtering, amplification, and bridge circuits) that are applied to the sensor output, making it suitable to be interfaced to a measurement device or an analog-to-digital converter. The last topic covered in this chapter was the 4-20 mA sensor output method.

Questions

7.1 Name two static performance characteristics of sensors.

7.2 Define what is meant by sensor bandwidth.

7.3 What is the difference between sensor accuracy and repeatability?

7.4 Explain why repeatability error cannot be improved by calibration.

7.5 List several sensors that are covered in this chapter that produce an analog output.

7.6 List several sensors that are covered in this chapter that produce a digital output.

7.7 List some disadvantages of potentiometers.

7.8 What are the differences between resolvers and encoders?

7.9 What is the Hall effect?

7.10 Explain the operation of ultrasonic proximity sensors.

7.11 Which type of non-contact proximity sensors can detect only metal objects?

7.12 What is 'ripple' in tachometers?

7.13 What is the principle of operation of a strain gage?

7.14 Explain the difference between reaction and rotary torque sensors.

7.15 What is a thermistor?

7.16 What is the difference between a thermocouple and an RTD sensor?

7.17 What is the difference between a vibrometer and an accelerometer?

7.18 Why is a Wheatstone bridge circuit used in some signal conditioning circuits?

7.19 What condition causes a Wheatstone bridge's arms to be balanced?

7.20 Why are filters used?

7.21 Name three different filter types.

7.22 What is the advantage of the 4-20 mA sensor output method?

PROBLEMS

P7.1 Research and identify the type of sensors used in the following applications.

a. Kitchen oven

b. RPM indicator in vehicles

c. Back-up sensor in certain vehicles

d. Trunk compartment closure

e. Refrigerator door closure

f. Laptop cooling system

g. Vehicle engine cooling system

h. Servo robot

P7.2 A rotary potentiometer with a resistance of 5 kΩ and a measuring range of 325° uses a 10 V supply. The potentiometer output was read by a measuring device with a resistance of 100 kΩ. Determine the angle measured by the potentiometer if the measuring device output is 6 V. What would the measured angle be if the measuring device resistance is 1 MΩ instead?

P7.3 A geared DC motor has a built-in incremental encoder that is connected to the motor side. The encoder disk has 1250 lines, and the gear ratio is 9:1. Determine the angular resolution of this encoder, assuming that the encoder is operated in quadrature mode.

P7.4 A single-turn, 10-bit absolute encoder has a gray code output. Determine a) the resolution of this encoder and b) the encoder output for the first 16 angular positions read by the encoder.

P7.5 Draw a circuit to interface the output of a two-wire NO proximity sensor (see Example 7.3) that uses a 24 VDC power supply to the digital input line of an MCU that operates with V_{DD} of 5 V.

P7.6 A strain gage with a resistance of 120 Ω and a gage factor of 2 has a resistance change of 0.005 Ω. a) Determine the microstrain measured by the gage. b) If this microstrain is due to the axial elongation of a rectangular steel bar with a cross sectional area of 0.4×10^{-3} m^2, determine the axial force acting on the bar.

P7.7 A platinum RTD sensor has a nominal resistance of 100 Ω at 0° and a TC factor of 0.00392 Ω/Ω/°C. Determine the temperature read by the sensor if the RTD resistance is 200 Ω.

P7.8 An IC digital temperature sensor has an output sensitivity of 10 mV/°C. The output of the sensor was read by a 10-bit A/D with a 5 V reference. Determine the actual temperature if the A/D reading is 84.

P7.9 An IC vibration sensor has a range of ±5 g and a sensitivity of 400 mV/g. The signal-conditioning circuit for this sensor gives an output of 2.65 V at 0 g. For positive acceleration, the output increases above 2.65 V, and for negative acceleration, the output decreases below 2.65 V. Determine the actual acceleration if the sensor output is a) 2.0 V or b) 3.8 V.

P7.10 A Wheatstone bridge similar to that shown in Figure 7.62 is used to determine the unknown resistance R_1 by adjusting the resistance R_2. The resistance R_3 was found to be 151 Ω when the bridge was first balanced. When the resistances R_2

and R_4 were interchanged and the bridge was balanced again, R_3 was found to be 182.5 Ω. What is the value of R_1?

P7.11 Show how Equation (7.54) was obtained.

P7.12 Derive Equation (7.58) for a constant-current source bridge.

P7.13 For the strain gage considered in Example 7.8, design an op-amp circuit to amplify the bridge output voltage signal by a factor of 1000.

P7.14 For the assembly system shown in Fig. 6.12, do the following:

a. Select an appropriate sensor to detect the part presence on conveyer *A* at the location where the part is picked up by robot #1. Assume that the part is metal, made of steel, has dimensions of $2 \times 2 \times 1.5$ in., and it is detected using one of its side surfaces. Justify your selection, and explain how the selected sensor will be interfaced to a PC or a microcontroller system.

b. Select an appropriate sensor to check for the 'quality' of the completed assembly at station #3. Assume that the quality of the assembly is done by measuring the height of the two-part assembly relative to the indexing table surface. Justify your selection, and explain how the selected sensor will be interfaced to a PC or a microcontroller system.

P7.15 Consider a back-up obstacle warning system (similar to that found in some vehicles) that just uses one ultrasonic distance sensor. Assume that the sensor provides a 0 to 10 VDC analog voltage output that is proportional to the distance of the object from the sensor. The sensor output is zero when the object is at the near limit setting of the sensor. Design a circuit that can process the output of this sensor to inform the user of the closeness of the object to the sensor. You can use multiple LEDs or a buzzer as an indication of the output. Make any reasonable assumptions.

Laboratory/Programming Exercises

L/P7.1 Build an RC filter circuit with R = 5.1 kΩ and C = 0.1 μF. Use the function generator to apply a sinusoidal signal with frequencies ranging from 10 to 3000 Hz. Record the amplitude of the input and output signals as well as the phase shift as a function of frequency. Plot the filter response data (magnitude and phase) and compare to that expected from theory.

L/P7.2 Write code (use MATLAB or Excel) to simulate the operation of a low-pass digital filter (Equation 7.38) with a corner frequency of 5 Hz and a sampling frequency of 1 ms. Generate plots to show the filtering effects for sinusoidal signals with a frequency of 1, 10, and 20 Hz.

L/P7.3 Use MATLAB to create a notch filter with a notch frequency of 60 Hz. Apply sinusoidal signals with the same amplitude but with the following frequencies: 54 Hz, 59 Hz, 60 Hz, and 61 Hz. For each signal, record the steady state output amplitude of the notch filter.

L/P7.4 Using any PIC MCU, develop a program to read the voltage output from an analog sensor such as the LM35C temperature sensor. Convert the read voltage value to temperature in engineering units, and then display the temperature reading to the user every 1 second through an RS-232 interface to a terminal program (such as *HyperTerminal* or *PuTTY*).

L/P7.5 Develop a VBE program that can display the room temperature using the LM35C temperature sensor. Connect the sensor output to one of the A/D channels on the data acquisition card that the PC has. Display the temperature every one second in a textbox. (Note: this lab exercise assumes the availability of a data-acquisition card with a software library for accessing the A/D).

CHAPTER 8

Actuators

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- List the advantages of electrically powered actuators
- Explain the common configurations of brush-type DC motors
- Model the electro-mechanical behavior of a PM brush DC motor
- Explain the working principle of BLDC, AC, stepper, universal, and hobby servo motors
- Explain the construction and drive techniques for stepper motors
- Interpret the torque-speed characteristics of an actuator
- Explain drive methods and amplifiers for different actuators
- Select an appropriate actuator for a given mechatronic application

8.1 Introduction

Actuators are the key components of all mechanized equipment. An **actuator** is a device that converts energy to mechanical motion. There are many types of actuators available to drive machinery such as electric, internal-combustion, pneumatic, and hydraulic. Electrically powered actuators are widely used in equipment (such as pumps, compressors, machine tools, and robots). Internal-combustion actuators are normally used for mobile applications (such as in vehicles, boats, and in power generation equipment). Pneumatic actuators, which use compressed air as the power source, are mostly used in machinery applications (such as pick-and-place robots and in air-powered tools). Hydraulic actuators, which use a pressurized oil to drive a piston, are used in applications for lifts and presses, among others.

This chapter focuses on electrically powered actuators, which are commonly used in mechatronic systems. Electrically powered actuators have the characteristics that they are clean (no leaking pressurized fluids), require no extra equipment (no need for air storage tanks, fuel tanks, or air filters), can operate indoors without the need for exhaust systems, and can be easily controlled (especially DC-type electric motors). One disadvantage of electrically powered actuators is their low power-to-size ratio. In this chapter, the characteristics of several types of electric motors are discussed, along with information on how to select and size an actuator for a mechatronic system.

Electric motors can be broadly classified into direct current (DC) and alternating current (AC), which refer to the type of electric power that drives the motor. Within each category, the motors can be further classified depending on the

physical construction of the motor and how the motor is wired. All electrically powered motors consist of two main components: a stator and a rotor, which are separated by an air gap. The **stator** is the fixed part of the actuator, while the **rotor** is the movable part of the actuator. In rotary-type electric motors, the stator takes the form of a hollow cylinder that is attached to the housing, while the rotor has a shaft that is supported by bearings. For further reading, see [31-33].

8.2 DC Motors

Direct-current or DC motors are motors that require non-alternating or steady voltages to be operated. DC motors are available in variety of sizes ranging from a few watts up to 5 horsepower (hp). There are two classes of DC motors: brush and brushless. We will start by taking about brush-type motors.

8.2.1 Brush DC

A brush type DC motor is a very common type of electric motor. This motor is used in many battery-powered devices such as electric toys, and in industrial applications such as conveyers, indexing tables, and material handling. A brush-type motor, as the name suggests, uses a pair of **brushes** to transfer the electric current to the rotating coil to allow continuous flow of current in the same direction as the rotor rotates. The brushes, which historically used to be made of copper bristles, are constructed out of spring-loaded graphite, and are designed to contact the **commutator**, which consists of a ring of alternating conducting and insulating segments that is attached to the rotor. To illustrate the working principle of the brush DC motor, consider Figure 8.1, which shows a simplified arrangement of two brushes and a two-piece commutator. The leads of the rotor coil, which are situated between a magnetic field, are attached to the commutator. A brush-type DC motor develops a torque (and hence rotation of the rotor) when a current flows in the rotor coils through the magnetic field created by the stator. The torque is the result of the two equal but opposite forces acting on the sides of the coil. The force is given by **Lorentz's law** which states that when a conductor carries a current through a magnetic field, a force acts on the conductor and is given as the vector cross product of the current and the magnetic field, or mathematically:

$$\boldsymbol{F} = \boldsymbol{I} \times \boldsymbol{B} \tag{8.1}$$

Where $\boldsymbol{I}$ is the current vector and $\boldsymbol{B}$ is the magnetic field vector. For the current arrangement shown in Figure 8.1, these forces will cause the coil to rotate clockwise

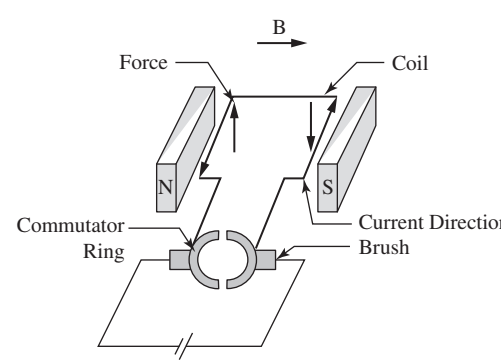


Figure 8.1
Simplified construction of a brush DC motor

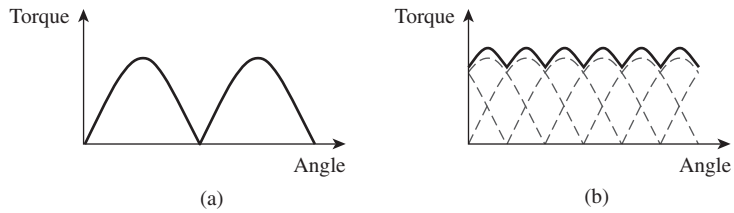


Figure 8.2

Resultant torque output as function of the angle (a) single-coil and (b) three-coil segments

as seen from the brush end. Without the use of a commutator to maintain the direction of the current, the torque direction will reverse the moment the coil passes through the vertical plane (called the **commutation plane)**, producing no useful motion. Using a two-piece commutator, the resulting torque will not be smooth and will exhibit the ripple shown in Figure 8.2(a). Note that the torque is maximum when the coil is horizontal because at this position, the moment arm distance between the forces acting on the coil is maximum. Similarly, the torque is zero when the coil is in the commutation plane because the moment arm distance is zero. In general, the torque is a function of the sine of the angle between the magnetic field direction and the vector normal to the plane of the coil, θ, where θ is 90° for the coil position shown in Figure 8.1. If we had used a six-piece commutator (and also three coils, one for each commutator pair) instead of using a two-piece commutator, the torque would be smoother (Figure 8.2(b)) since the torque at any point in time is the sum of all the torques in all the coils. If the stator magnetic field could be made radial [34], then the ripple is eliminated since the angle θ will always be 90°. To improve the torque characteristics, commercial motors have a commutator that is split into 50 or more segments. Figure 8.3 shows the brushes and commutator of a small DC motor.

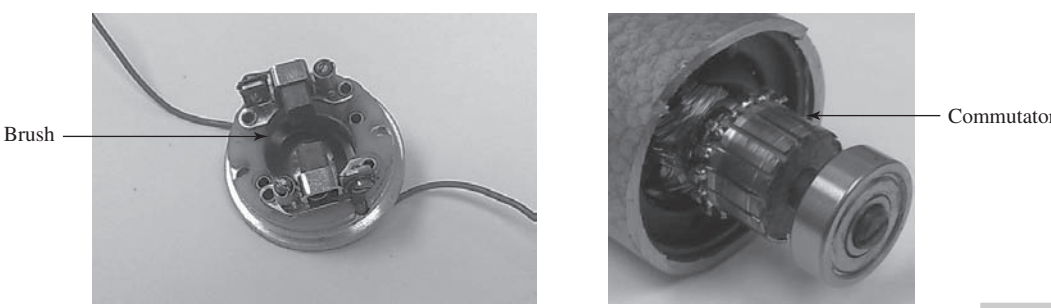


Figure 8.3

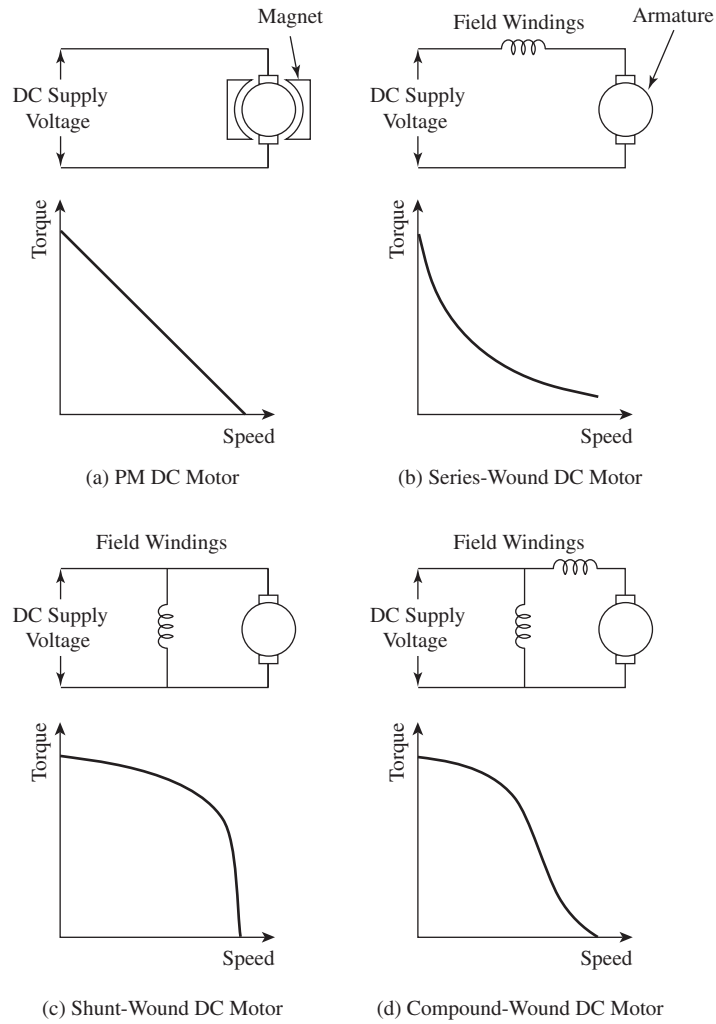
Commercial brush and commutator of a brush DC motor
(Jouaneh, University of Rhode Island)

Note that the mechanical contact between the brushes and the commutator leads to wear of these components as well as the formation of arcs which require these components to be serviced periodically.

One common configuration of brush DC motors is the **permanent magnet** DC motor where the stator is constructed of permanent magnets, and the rotor is made of wire coils (see Figure 8.4(a)). This configuration has the property of linear torque-speed characteristics. Other configurations can be obtained by using electromagnets for the stator and by how the stator and rotor coils are wired. These configurations include series-wound motors, shunt-wound motors, and compound-wound motors (see Figure 8.4). In a **series-wound** motor, the stator and rotor coils are connected in series. In a **shunt-wound** motor, the stator coils and the rotor coils are connected in parallel, while in the **compound-wound** configuration both series and parallel fields are used. Notice how the torque-speed curve is nonlinear

Figure 8.4

Common configurations of brush-type DC motors



for series, shunt, and compound-wound configurations. Notice also that in the **series-wound configuration** the motor speed increases uncontrollably if the load acting on the motor was accidentally disconnected since the speed at zero torque is not limited. In small motors, the internal friction is usually sufficient to limit the breakdown speed, but in large motors, external safety devices need to be implemented. Series-wound motors are used where there is a need for a very large torque at low speed. An example would be moving a very heavy load from rest such as an electric train, an elevator, or a hoist. **Shunt-wound motors** have a nearly constant speed under varying loads. This makes them attractive to drive machine tools and rotating equipment such as fans and blowers where it is desirable to have steady speeds. **Compound-wound motors** combine the characteristics of both series-wound and shunt-wound motors. They are used where there is a need for both a high starting torque and a constant speed operation such as in punch presses, and shears. They are also safer to use in cases where the load might disconnect such as in cranes since they have a controlled no-load speed.

Now let us examine the torque-speed characteristics of a PM brush DC motor. Such a motor can be modeled as shown in Figure 8.5. On the electrical side, the

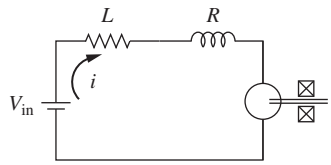


Figure 8.5

Electro-mechanical model of a PM brush DC motor

rotor coil inductance and its electrical resistance are modeled as an inductor and a resistor in series. The **back electromotive force (EMF)** voltage due to the rotation of the conducting coil in the stator magnetic field is represented as a voltage source. On the mechanical side, the rotating coil is represented as inertia with friction acting on it. The motor inductance will be neglected in the modeling since it is usually small. This point is illustrated in Example 8.3. The current through the motor coil is then given by

$$I = V/R = (V_{in} - V_{bemf})/R \tag{8.2}$$

The torque developed by the motor is proportional to the current through the motor coil, and is given by

$$T = K_T I \tag{8.3}$$

where K_T is the torque constant. Since the back EMF voltage (V_{bemf}) is proportional to the rotating speed of the coil, we can write

$$V_{bemf} = K_E \omega \tag{8.4}$$

where ω is the rotational speed of the motor and K_E is the back EMF constant.

Combining Equations (8.2) through (8.4), we get

$$T = K_T \frac{V_{in}}{R} - K_T K_E \frac{\omega}{R} \tag{8.5}$$

If we set $K_T V_{in}/R$ to be the starting torque T_s and $K_T K_E/R$ to be the constant α, then the torque speed relationship is written as

$$T = T_s - \alpha\omega \tag{8.6}$$

The above relationship is plotted in Figure 8.6 for three different values (V_1, V_2, and V_3) of the supply voltage V_{in}, where $V_3 > V_2 > V_1$. The three plots have

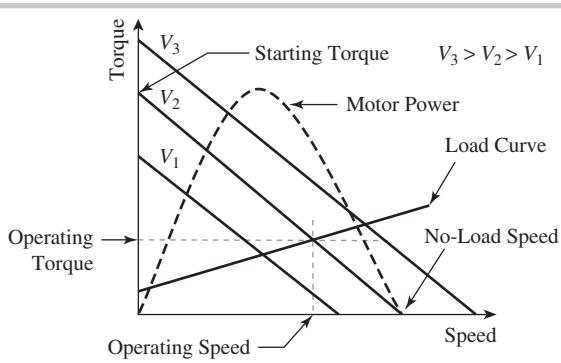


Figure 8.6

Typical torque–speed characteristics of a PM brush DC motor

the same slope, but the starting torque increases (decreases) with an increase (decrease) in the supply voltage. For a given supply voltage, the torque speed curve is linear and is defined by two parameters, the starting torque or the stalled torque, and the no-load speed. The **starting torque** is the maximum torque that can be obtained from the motor. This maximum torque is obtained when the motor is stationary. As shown by Equation (8.5), as the motor starts rotating, the back EMF voltage generated due to the rotation of the conducting rotor through the magnetic field generated by the stator acts against the voltage applied to the motor leads. This causes the current through the motor coils to reduce, and thus the output torque, since the output torque is linearly related to the current through the coil. So, as the motor picks up speed, the torque further reduces and it will go to zero at the **no-load speed**, which is the maximum speed that can be obtained from a DC motor. Figure 8.6 also shows the output power of the PM brush DC motor as a function of the operating speed. Notice that the **maximum power** is obtained when the motor is operating at a speed equals to half the no-load speed. This can be verified by writing an expression for the power, differentiating it, and setting it equal to zero to solve for the speed at which the power is maximum. When a motor drives a load, the operating speed of the motor will be less than the no-load speed, and it will be at the point where the load torque matches the motor torque as seen in the figure. Notice that if the load torque changes linearly with speed, the operating speed of the motor then linearly increases (decreases) with an increase (decrease) in the supply voltage. The easy adjustment of operating speed through voltage input control is one advantage of PM brush DC motors over other types of electric motors. Example 8.1 shows how to determine the operating conditions of a load driven by a PM brush DC motor, while Example 8.2 considers the dynamic modeling of a mechanical system driven by a PM DC-motor.

Example 8.1 Operating Conditions of a Load Driven by a PM DC-Motor

A one-quarter hp DC-geared motor is used in a lift mechanism to lift a load of 10 kg using a simple pulley arrangement, as shown in Figure 8.7. The no-load motor speed is 300 rpm, and the starting torque is 23.8 N · m (or 210 lb-in.). The frictional resistance in the pulley drive is 2 N.m. Neglecting the inertia of the rotor, the pulley, and the cable, determine:

a. The initial acceleration of this load.
b. The steady-state lifting speed of the load.
c. Output horsepower of the motor at steady state.

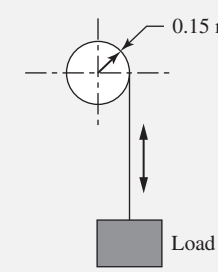


Figure 8.7

Solution:

a. At startup, the available torque to accelerate the load is the starting torque minus the friction torque and the gravity torque. Or $23.8 - 2 - 10 \times 9.81 \times 0.15 = 7.1$ N · m. At a radius of 0.15 m, this corresponds to a starting force of $7.1/0.15 = 47.3$ N. Thus, the starting acceleration of the load is $47.3/10 = 4.73$ m/s^2. Note that as the motor speed starts increasing, the acceleration will decrease.
b. At a steady state, the load is moving up at a constant speed. The torque that the motor needs to overcome will be the sum of the friction torque

and gravity torque. This is given by:

$$T_{load} = 2 + 10 \times 9.81 \times 0.15 = 16.7 \text{ N} \cdot \text{m}$$

When the motor is used to lift the load, the steady-state speed is determined from

$$T_{motor} = 23.8 - (23.8/300)\,\omega = T_{load} = 16.7 \text{ N} \cdot \text{m}$$
$$=> \omega = 89.5 \text{ rpm}$$
$$V_{steady} = \omega r = (89.5 \times 2\pi/60) \times 0.15 = 1.41 \text{ m/s}$$

c. At steady state, the output horsepower of the motor is hp $= 16.7 \times 89.5 \times 2\pi/60 \times 1/746 = 0.21$ hp. Notice that the output hp is less than the rated hp for the motor since the steady-state speed is not half the no-load speed.

Example 8.2 Modeling of a Mechanical System Driven by a PM DC Motor

For the drive system shown in Figure 8.8 with a gear ratio of N:1, assume that the motor is a PM DC motor. Develop a dynamic model that relates the input voltage applied to the motor to the motor speed as measured by a tachometer mounted on the motor shaft. The tachometer has a sensitivity of k_{tach} V/rpm. Let the viscous damping coefficient at the input shaft be b_1 and at the output shaft be b_2. Assume that the shafts are rigid, and let I_1 represents the combined inertia of the motor shaft, input shaft, coupling, and the pinion, and I_2 represents the combined inertia of the gear, the output shaft, and the load link.

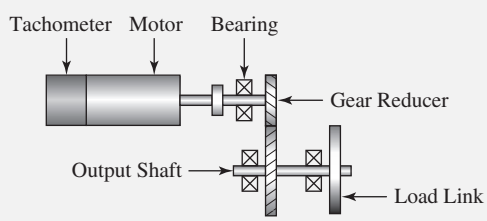


Figure 8.8

Solution:

With the shafts assumed to be rigid, and since there are no external torques acting on the system, the torque (T_m) supplied by the motor is only used to overcome the inertia and damping in the system. Thus, we can write

$$T_m = I_1\ddot{\theta}_1 + b_1\dot{\theta}_1 + (I_2\ddot{\theta}_2 + b_2\dot{\theta}_2)/N \quad (1)$$

where θ_1 is the angular velocity of the input shaft, and θ_2 is the angular velocity of the output shaft. Due to gearing,

$$\dot{\theta}_2/\dot{\theta}_1 = 1/N \quad (2)$$

Using Equation (2) to express the velocity and acceleration of the output shaft in terms of the input shaft, we obtain

$$T_m = I_1\ddot{\theta}_1 + b_1\dot{\theta}_1 + \frac{(I_2\ddot{\theta}_1 + b_2\dot{\theta}_1)}{N^2} = I_{eff}\ddot{\theta}_1 + b_{eff}\dot{\theta}_1 \quad (3)$$

Where $I_{\text{eff}} = I_1 + \frac{I_2}{N^2}$, and $b_{\text{eff}} = b_1 + \frac{b_2}{N^2}$. Note that similar to inertia, the damping coefficient is reduced by the factor N^2 when reflected to the input shaft. Equation (3) is a model for this system that relates the input torque to the angular acceleration of the input shaft. Using the angular velocity of the motor shaft as the output variable, the model is

$$T_m = I_{\text{eff}}\,\dot{\omega}_1 + b_{\text{eff}}\,\omega_1 \tag{4}$$

From Equation (8.5), the torque supplied by a PM DC motor is given as

$$T_m = K_T \frac{V_{\text{in}}}{R} - K_T K_E \frac{\omega_1}{R} \tag{5}$$

and the tachometer output voltage is related to the motor speed by

$$V_{\text{tach}} = k_{\text{tach}}\, k_2\, \omega_1 \tag{6}$$

Where $k_2 = 60/2\pi$ is a conversion factor from rad/s to rev/min. Combining Equations (4) through (6), we obtain:

$$\dot{V}_{\text{tach}} = \frac{1}{RI_{\text{eff}}}\left(K_T V_{\text{in}} k_{\text{tach}} k_2 - (K_T K_E + b_{\text{eff}} R) V_{\text{tach}}\right) \tag{7}$$

Using Laplace transform, the transfer function between V_{in} and V_{tach} is

$$\frac{V_{\text{tach}}}{V_{\text{in}}} = \frac{K_T k_{\text{tach}} k_2}{RI_{\text{eff}}\, s + (K_T K_E + b_{\text{eff}}\, R)} = \frac{k_m}{\tau_m\, s + 1} \tag{8}$$

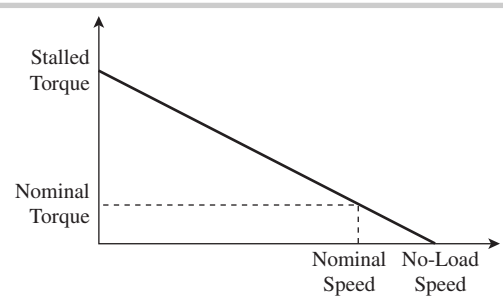
Where k_{m} and τ_{m} are constants that can be obtained from Equation (8). As shown by Equation (7) or (8), the model is a first-order system.

In manufacturer data for a PM DC motor, the manufacturer typically lists the **nominal speed** (or rated speed) and **nominal torque** (or rated torque) values for the given motor in addition to the stall torque and no-load speed. The relationships between these parameters are shown in Figure 8.9. Typically, the nominal speed is 75 to 90% of the no-load speed, and the nominal torque is 10 to 25% of the stall torque. Figure 8.10 shows manufacturer data for a small PM brush DC motor made by Pittman, and Example 8.3 explains some of this data. Note that PM brush DC motors are not suitable for continuous duty operation due to thermal effects. The nominal torque parameter defines the maximum torque that can be applied to the given motor for continuous duty operation with the temperature of the motor winding not exceeding the permissible temperature for operation at room temperature.

For intermittent duty operation, the applied torque can exceed the nominal torque. The duration of this over torque operation depends on the thermal time constant for the given motor, and it can range from few seconds to few minutes.

Figure 8.9

Nominal speed and torque for a PM DC-motor



Brush Commutated DC Servo Motors

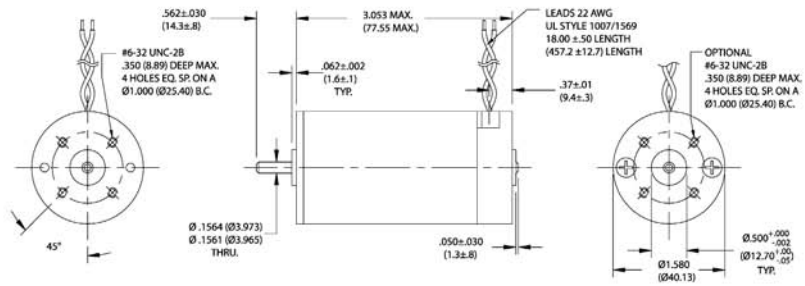
9236 Series

PITTMAN®

Figure 8.10

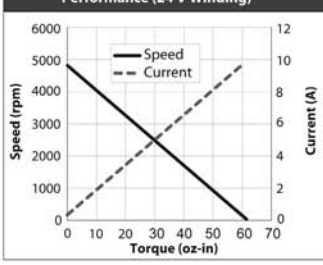
Manufacturer data for a Pittman 9236 Series PM brush DC motor

(Courtesy of AMETEK, Kent, OH)



		Part/Model Number							
Specification	**Units**	**9236 9.55 V**	**9236 12.0 V**	**9236 15.2 V**	**9236 19.1 V**	**9236 24.0 V**	**9236 30.3 V**	**9236 38.2 V**	**9236 48.0 V**
Supply Voltage	VDC	9.55	12.0	15.2	19.1	24.0	30.3	38.2	48.0
Continuous Torque	oz-in	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50
	Nm	0.0671	0.0671	0.0671	0.0671	0.0671	0.0671	0.0671	0.0671
Speed @ Cont. Torque	RPM	3530	3750	3850	3880	3980	3980	4010	3990
Current @ Cont. Torque	Amps (A)	4.52	3.65	2.88	2.26	1.82	1.44	1.14	0.90
Continuous Output Power	Watts (W)	25	26	27	27	28	28	28	28
Motor Constant	oz-in/sqrt W	3.7	3.9	4.0	4.1	4.1	4.2	4.2	4.2
	Nm/sqrt W	0.026	0.028	0.028	0.029	0.029	0.03	0.03	0.03
Torque Constant	oz-in/A	2.62	3.25	4.12	5.24	6.49	8.24	10.4	13.1
	Nm/A	0.019	0.023	0.029	0.037	0.046	0.058	0.073	0.093
Voltage Constant	V/krpm	1.94	2.40	3.05	3.87	4.80	6.09	7.66	9.69
	V/rad/s	0.019	0.023	0.029	0.037	0.046	0.058	0.073	0.093
Terminal Resistance	Ohms	0.50	0.71	1.07	1.64	2.49	3.91	6.14	9.72
Inductance	mH	0.43	0.66	1.06	1.72	2.63	4.24	6.70	10.7
No-Load Current	Amps (A)	0.40	0.33	0.26	0.20	0.16	0.13	0.10	0.080
No-Load Speed	RPM	4730	4800	4800	4750	4820	4790	4810	4770
Peak Current	Amps (A)	19.1	16.9	14.2	11.6	9.64	7.75	6.22	4.94
Peak Torque	oz-in	49.0	53.9	57.5	60.0	61.5	62.8	63.4	63.7
	Nm	0.3459	0.3805	0.406	0.4236	0.4342	0.4434	0.4476	0.4497
Coulomb Friction Torque	oz-in	0.80	0.80	0.80	0.80	0.80	0.80	0.80	0.80
	Nm	0.0056	0.0056	0.0056	0.0056	0.0056	0.0056	0.0056	0.0056
Viscous Damping Factor	oz-in/krpm	0.053	0.053	0.053	0.053	0.053	0.053	0.053	0.053
	Nm s/rad	3.56E-6	3.56E-6	3.56E-6	3.56E-6	3.56E-6	3.56E-6	3.56E-6	3.56E-6
Electrical Time Constant	ms	0.86	0.93	0.99	1.0	1.1	1.1	1.1	1.1
Mechanical Time Constant	ms	10	10	8.9	8.5	8.4	8.2	8.1	8.0
Thermal Time Constant	min	14	14	14	14	14	14	14	14
Thermal Resistance	Celsius/W	14	14	14	14	14	14	14	14
Max. Winding Temperature	Celsius	155	155	155	155	155	155	155	155
Rotor Inertia	oz-in-sec2	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010
	kg-m2	7.06E-6	7.06E-6	7.06E-6	7.06E-6	7.06E-6	7.06E-6	7.06E-6	7.06E-6
Weight (Mass)	oz	13.8	13.8	13.8	13.8	13.8	13.8	13.8	13.8
	g	391.2	391.2	391.2	391.2	391.2	391.2	391.2	391.2

Performance (24 V Winding)



Standard Features

- Sintered Bronze Bearings
- 2-Pole Stator
- Ceramic Magnets
- 7-Slot Armature
- Heavy-Gage Steel Housing
- Silicon Steel Laminations
- Copper-Graphite Brushes
- Diamond-Turned Commutator

Complementary Products

- Encoders
- Gearboxes
- Brakes

Notes

1 All values specified at 25°C ambient temperature and without heat sink.
2 Peak values are theoretical and supplied for reference only.

AMETEK TECHNICAL & INDUSTRIAL PRODUCTS
343 Godshall Drive, Harleysville, PA 19438
USA: +1 215-256-6601 - Europe: +44 (0) 845 366 9664 - Asia: +86 21 5763 1258
www.ametektip.com

AMETEK®
TECHNICAL & INDUSTRIAL PRODUCTS

Example 8.3 Characteristics of a PM Brush DC Motor

Using the motor data in Figure 8.10 for 24 V operation, do the following.

a. Verify the listed peak torque and no-load speed values.
b. Verify the listed electrical and mechanical time constants.

Solution:

a. The torque-speed relationship for a PM brush DC motor is given by Equation (8.5). Using SI units, V_{in} is 24 V, K_T is 0.046 N-m/A, K_E is 0.046 V/rad/s, and R is 2.49 Ω. Substituting these values into Equation (8.5) and evaluating, we get

$$\text{T} = 0.443 - 8.50 \times 10^{-4}\,\omega \tag{1}$$

The peak (or stalled) torque is obtained at $\omega = 0$. This gives 0.443 N-m. The max no-load speed is obtained by solving Equation (1) for $T = 0$. This gives a rotational speed of 521 rad/s. These values are very close (0.443 versus 0.434) and (521 versus 505) to the listed values in Figure 8.10.

b. The electrical time constant (τ_E) of the motor is a measure of the relationship between the input voltage to the motor and the output torque. If we do not neglect the armature inductance, then KVL applied to Figure 8.5 gives

$$V_{\text{in}} = L\frac{di}{dt} + iR + K_E\,\omega \tag{2}$$

Using Equation (8.3) and converting to the algebraic domain using the Laplace transform, we get the following expression for the motor torque:

$$T(s) = \frac{K_T}{Ls + R}\left(V_{\text{in}}(s) - K_E\,\omega(s)\right) \tag{3}$$

The transfer function between input voltage and output torque is then

$$\frac{T(s)}{V_{\text{in}}(s)} = \frac{K_T}{Ls + R} \tag{4}$$

Substituting the given values for K_T, L, and R, we get

$$\frac{4.6e^{-2}}{2.63e^{-3}s + 2.49} \text{ or } \frac{1.85e^{-2}}{1.06e^{-3}s + 1} \tag{5}$$

or an electrical time constant of 1.06 ms (or in general, τ_E is given as L/R), which matches the value listed in Figure 8.10. We note that if the electrical time constant is small (such as in this case), then the inductance can be neglected in dynamic modeling without appreciably changing the accuracy of the dynamic model.

The mechanical time constant can be found from Equation (8) in Example 8.2, or

$$\tau_m = \frac{RI_{\text{eff}}}{K_T K_E + b_{\text{eff}}R} \tag{6}$$

For the motor data, $b_{\text{eff}} = 3.56e^{-6}$ N-m-s/rad, and I_{eff} is $7.06e^{-6}$ kg $\cdot$ m^2. Substituting these values and the values of K_T, K_E, and R into Equation (6), we get

$$\tau_m = \frac{2.49 \times 7.06e^{-6}}{4.6e^{-2} \times 4.6e^{-2} + 3.56e^{-6} \times 2.49} = 8.27 \text{ ms}$$

which is very close to the listed value of 8.4 ms. Note that in practice, the effective load inertia is added to the rotor inertia giving a larger time constant.

8.2.2 Brushless DC

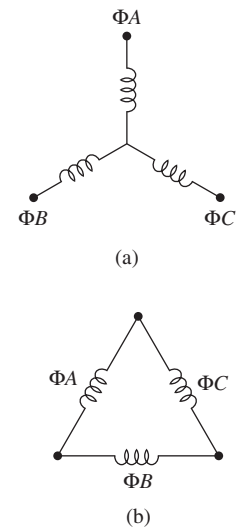
Unlike a brush DC motor, a brushless DC (BLDC) motor, as the name suggests has no brushes. Because there are no brushes, a BLDC motor produces little electrical and acoustic noise, and does not suffer from the wear of brushes and the need to replace them periodically. Thus a BLDC motor is more reliable than a brush DC motor. In a BLDC motor, the rotor is made of permanent magnets, and the stator is constructed of coils. There is no wiring to the rotor. Because the rotor is lighter than that in a brush motor, a BLDC motor can operate at much higher speeds than a brush motor. In addition, a BLDC is more efficient than a brush motor due to the absence of brush friction, and is thermally better suited to dissipate heat since the powered coils are located on the exterior portion of the motor. Brushless motors are normally used in machinery applications that require fast response, low heat generation, and long life. Hence, they are used in high-end machine tools and robots and computer disk drives.

In a BLDC motor, unlike a brush motor, commutation is not mechanically done, but is performed through electronic means in which the stator fields are electronically commutated, depending on the position of the rotor. In most cases, the rotor position is obtained from non-contact, Hall-effect type, proximity sensors (see Section 7.4.1) that are mounted on the stator, but encoder feedback also can be used. Some BLDC motor drivers perform sensorless control, which is based on using the EMF voltage over an unpowered coil to determine when to perform commutation of the current in the remaining coils. BLDC motors are available in single phase, two-phase, and three-phase configurations, where the phase refers to the number of independent windings on the stator, with the three-phase being the most common configuration for industrial motors. For a typical **three-phase winding**, the phases are wired in either a delta or a Y configuration (see Figure 8.11). The Y configuration is more commonly used and is electrically more efficient. A three-phase BLDC motor cable typically has three wires, one for each motor phase, in addition to the five wires for the three Hall Effect sensors (one wire for each sensor output plus ground and supply voltages wires). The brushless amplifier drives two of the three motor phases with DC current during each 120° rotation segment of the rotor.

To illustrate the operation principle of a BLDC motor, let us consider a simplified three-phase BLDC motor with a two-pole rotor as shown in Figure 8.12. The stator coils are labeled phase A (ΦA), phase B (ΦB), and phase C (ΦC) and wired using a single circuit similar to that shown in Figure 8.11a. Real motors have multiple circuits that are wired in parallel to each other, and a corresponding number of

Figure 8.11

(a) Y wiring and (b) delta wiring of a three-phase BLDC motor



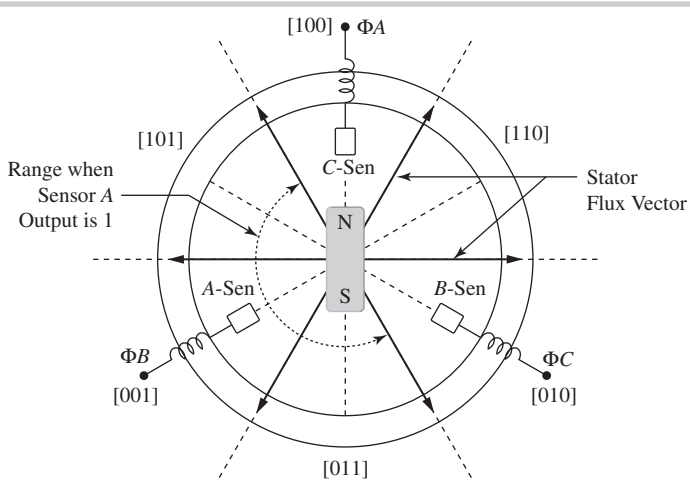


Figure 8.12

Schematic of an simplified three-phase BLDC motor

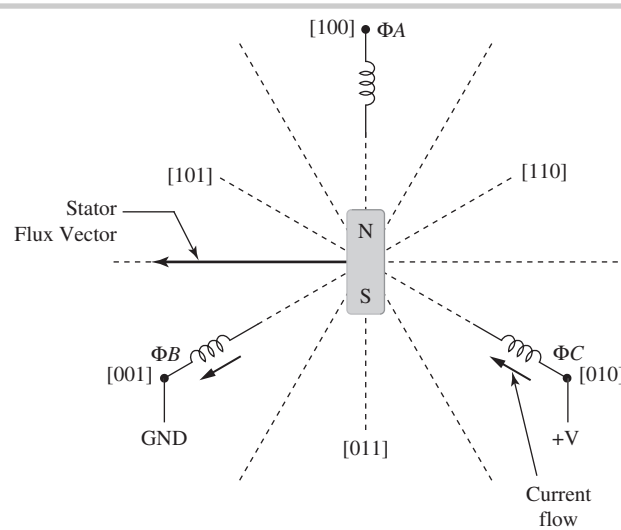
multi-pole rotors [35]. In the simplified schematic of Figure 8.12, one electrical revolution of the motor corresponds to one mechanical revolution. If two electrical circuits were used, then there are two electrical revolutions per one mechanical revolution. The figure also shows three Hall-effect sensors labeled *A*, *B*, and *C* that are placed on the stator. Each sensor outputs a high signal for 180° of electrical rotation and a low signal for the other 180°. The sensor output is high when the north pole of the rotor is pointing towards the sensor. Using this sensor arrangement, there are six combinations of the sensors' output, with one combination or state for each 60° of electrical rotation. Each combination is indicated in Figure 8.12 using the notation [CBA], where the least significant bit corresponds to sensor *A* output, and the most significant bit corresponds to sensor *C* output. In practice, the particular labeling of each sensor (i.e., whether it is *A* or *B*) is not important. What is important is the association of the sensors output states with the position of the rotor.

In a BLDC motor, the phases are e**lectronically commutated** as the rotor moves from one sensors state to another. For each combination of the sensors' output, two of the three phases are activated such as to produce an angle close to 90° between the stator and rotor flux vectors [36]. There are six possible **stator flux vectors** shown as arrows in Figure 8.12. A particular stator flux vector is generated for a given position of the rotor and a desired direction of rotation. For the rotor position shown in Figure 8.12, the stator flux vector should be horizontal and pointing toward to the left for CCW rotation or horizontal and pointing to the right for CW rotation. Figure 8.13 shows the generation of this stator flux vector for CCW rotation through activation of phases *C* and *B* and leaving phase *A* floating, with phase *C* connected to high voltage and phase *B* connected to low voltage. The commutation sequence for all possible sensors states are listed in Table 8.1 for both CW and CCW rotation. The reader can verify the commutation sequence for other positions. Using this commutation sequence, on can create a drive timing diagram for CW rotation of the rotor as shown in Figure 8.14. The 0° electrical position corresponds to the 12 o'clock position.

A brushless motor requires a special driver that can provide the proper excitation voltages to the stator coils. The driver is of the bipolar type and is commonly referred to as a **three-phase bridge driver** (see Figure 8.15). Such a driver consists of three parallel half H-bridge legs. As seen in Figure 8.15, closing switch S1 on the first leg and switch S4 the second leg and keeping the remaining switches open causes power to be applied to phases *A* and *B* of the motor with Phase *C* floating.

Figure 8.13

Illustration of phase activation to produce a particular stator flux vector



Sensor Output			CW Rotation			CCW Rotation		
C	***B***	***A***	**Φ*A***	**Φ*B***	**Φ*C***	**Φ*A***	**Φ*B***	**Φ*C***
1	0	0	NC	Hi	Low	NC	Low	Hi
1	0	1	Low	Hi	NC	Hi	Low	NC
0	0	1	Low	NC	Hi	Hi	NC	Low
0	1	1	NC	Low	Hi	NC	Hi	Low
0	1	0	Hi	Low	NC	Low	Hi	NC
1	1	0	Hi	NC	Low	Low	NC	Hi

Table 8.1

Commutation sequence for CW and CCW rotation

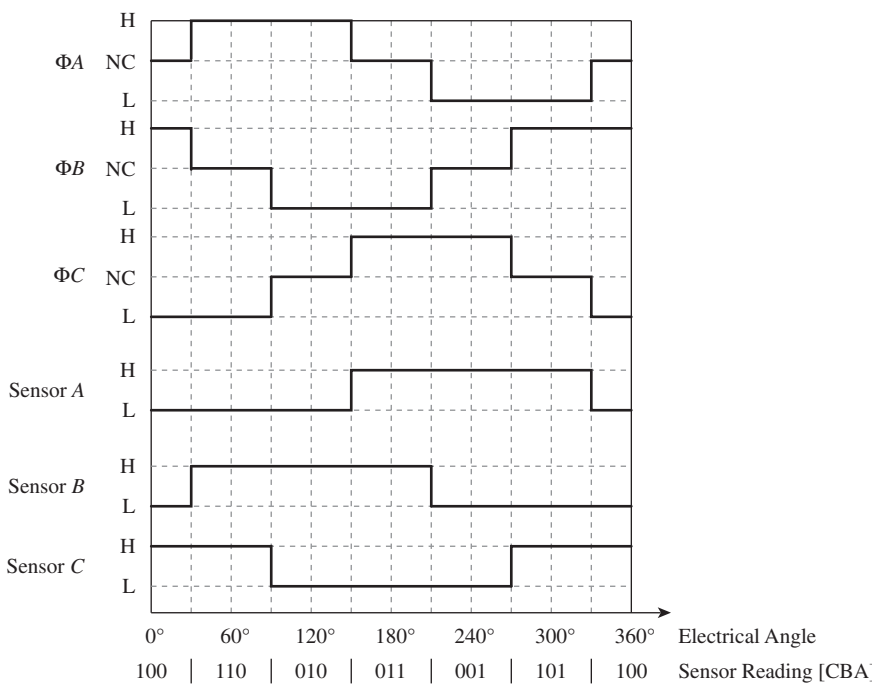


Figure 8.14

Drive timing diagram for CW rotation

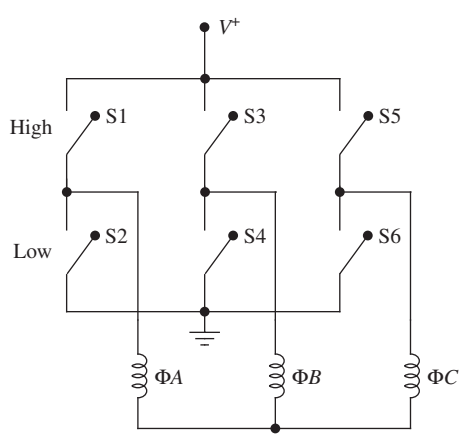


Figure 8.15

Three-phase bridge driver for driving a BLDC motor

Figure 8.16

Brushless DC cooling fan (Photos.com)

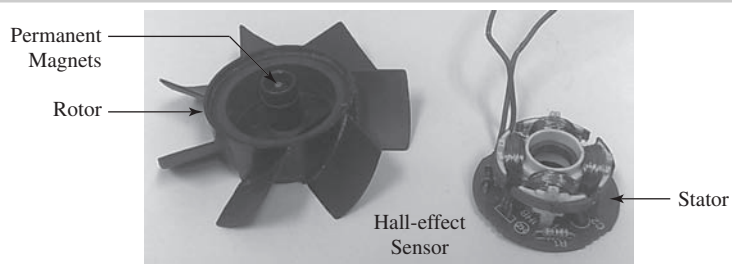
To reverse the current flow for phases *A* and *B*, switch S3 on the second leg is closed with switch 2 on the first leg. The particular switches to close are determined from a **commutation table** (such as Table 8.1), which provides the commutation sequence for correctly driving the motor. Obviously, the switches in Figure 8.15 are a representation for transistors in actual implementation.

An example of a brushless DC-motor is the motor that powers the small cooling fans in personal computers (see Figure 8.16). Since BLDC are very light, and produce little electrical and acoustic noise, they are preferred to use for this application. These fans are typically constructed using a two-phase BLDC motor.

A layout of the components of a BLDC fan is shown in Figure 8.17. The rotor has surface-mounted permanent magnets, while the stator has two-phase coils. The fan uses a single Hall-effect sensor that is mounted on the stator circuit board. When the rotor axis is aligned with the sensor, the sensor sends out two complementary 50% duty cycle waves. These cycles are fed to the transistor gate input that controls the current flow through each of the two coils. Since the two square waves are complementary, only one coil will be active at a time. Increasing the voltage supplied to the motor causes the motor to increase its speed. Some BLDC fans come with an output that indicates the speed of the fan.

Figure 8.17

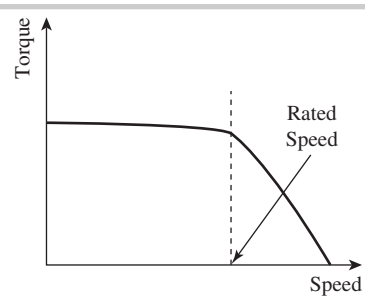
Components of a BLDC fan (Jouaneh, University of Rhode Island)



The torque–speed characteristics of a BLDC motor are different from a brush motor. Typical characteristics are shown in Figure 8.18. The figure shows that a brushless motor has a constant or slowly decreasing torque over a wide speed range up to the rated speed. After the rated speed, which is normally above 3000 rpm, the torque starts decreasing more rapidly. In the constant-torque region, the motor horsepower increases linearly with speed.

Figure 8.18

Torque–speed characteristics of a BLDC motor



8.2.3 Servo Drives

For motion control applications, servo drives or amplifiers are used to amplify the reference signal sent to the actuator since they place in one convenient package all the components needed to amplify the input signal. Drives can be broadly classified

as digital or analog. **Digital drives** can take input signals in various forms such as analog, digital (step and direction or PWM and direction), RS-232, or through an Ethernet connection, while **analog drives** take as an input analog (+/−10 VDC such as that supplied by a D/A converter) or PWM and direction signals. In a digital drive, the drive is configured using a software program running on a PC, while an analog drive is configured using switches and potentiometers. Analog amplifiers can be further classified as brush or brushless amplifiers, referring to the type of electric motor that they can drive. Brush-type analog amplifiers are designed to drive brush-type DC motors, and other inductive-type loads such as voice-coil motors, and magnetic bearings.

An example of an analog, brush-type amplifier is the **12A8 amplifier** made by Advanced Motion Controls and shown in Figure 8.19. This amplifier is suitable for driving small motors, takes ±10 VDC as input, is rated at 24 W, and uses an H-bridge circuit for the amplification. It requires a single unregulated DC power supply that can supply a DC voltage in the range of 20 to 80 V. This amplifier allows the user to adjust the following parameters using 14-turn potentiometers: loop gain, current limit, input gain and offset.

Loop Gain: The loop gain factor when the amplifier is operated in the voltage or velocity mode (see below).

Current Limit: The peak and continuous current that can be supplied by the amplifier. When the current limit is adjusted, the ratio of the continuous to peak current is maintained.

Input Gain: The ratio between the output variables (voltage, current, or velocity) and the input signal

Offset: A signal that can be used to adjust any imbalance in the input signal or the amplifier output.

This amplifier can operate in various modes including voltage, velocity, and current (torque) modes. In **voltage mode**, the amplifier produces an output voltage signal that is proportional to the input voltage signal regardless of variation in the supply power. The output amplification or loop gain is variable and set by adjusting the *Loop Gain* pot. In **velocity mode**, the amplifier supplies a voltage to maintain the motor at a specific speed. For this mode, the amplifier uses the actual motor speed, as measured by the tachometer that is attached to the motor shaft, as a feedback signal. In **current mode**, the amplifier produces an output current signal to the motor. Because the output torque of DC motors is proportional to the input current, the current operation mode is sometimes referred to as torque mode.

This amplifier, like many other commercial servo amplifiers, gives a high frequency (36 kHz) pulse width modulated output (or PWM output). Rather than changing the amplitude of the output signal, a PWM amplifier varies the duty cycle of fixed-amplitude, fixed-frequency signal as means of modulating the output power. This results in a very efficient way to deliver power to the load since it reduces the power lost in the output stage of the drive.

Another analog drive that is suitable for driving small motors is shown in Figure 8.20. This drive takes only PWM and direction signals as input, and thus, it is very suitable to be interfaced to microcontrollers, many of which can provide this type of output. Figure 8.21 shows the minimum wiring needed to drive a motor using this amplifier. The amplifier has other connections (such as current output monitor, fault output, and inhibit line), but these do not need to be connected for the amplifier to operate, and thus are not shown.

Figure 8.19

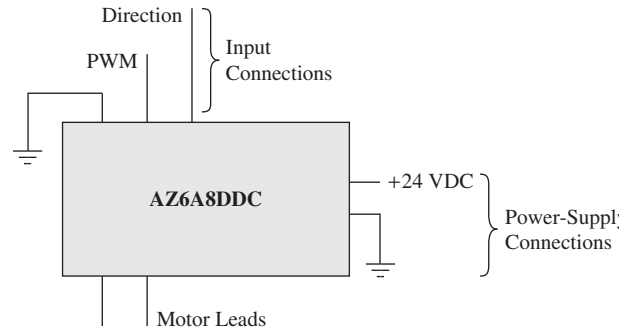
12A8 PWM amplifier
(Courtesy of ADVANCED Motion Controls, Camarillo, CA)

Figure 8.20

AZ6A8DDC analog drive
(Courtesy of ADVANCED Motion Controls, Camarillo, CA)

Figure 8.21

Minimum wiring for the AZ6A8DDC drive



8.2.4 PWM Control of DC Motors

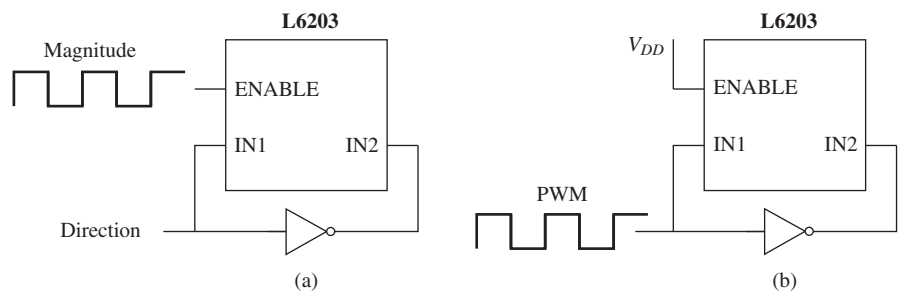
As discussed in the previous section, most commercial servo drives accept PWM and direction signals as input. Also, many servo drives produce a PWM output voltage signal to the load such as a motor coil instead of a linearly variable output voltage. This section will discuss PWM motor control in more detail. A plot of a PWM signal was shown in Figure 4.8, and we explained in Section 4.7.5 how one can set the parameters of the PWM signal using a PIC MCU. The duty cycle of a PWM voltage signal corresponds to the average voltage delivered to the load. For example, if the amplitude of the PWM signal is 12 V, then at a 25% duty cycle, an average voltage of 3 V is delivered to the load. Similarly, at 75% duty cycle, an average voltage of 9 V is delivered to the load. Normally, the PWM signal pulse period is much smaller than the time constant of the actuated load, so the load does not respond to each ON and OFF portion of the PWM signal but to the average value of the PWM signal.

There are actually two methods of using PWM signals in motor control. The first is called the sign-magnitude method and the other is called the locked anti-phase method. In the **sign-magnitude method**, the controller (such as a PIC MCU) provides two signals: a PWM signal to control the magnitude of the voltage supplied to the load and a digital two-level signal to control the direction or sign. In the sign-magnitude method, the higher the duty cycle of the PWM signal, the higher the average voltage (or magnitude) delivered to the load. At 0% duty cycle, no voltage is supplied to the load, while at 100% duty cycle, the full voltage signal is supplied to the load. For example, the sign-magnitude method can be used to drive the servo drive shown in Figure 8.21 or an H-bridge driver such as the L6203 shown in Figure 3.59. In using the L6203 bridge driver, additional circuitry is needed to interface the sign and magnitude signals to the IN1, IN2, and ENABLE leads of Figure 3.59 (see Figure 8.22(a)).

In the **locked anti-phase method**, a single PWM signal is used to control the voltage delivered to the load and the direction of rotation. In this method, opposite

Figure 8.22

Wiring of L6203 H-bridge for (a) sign-magnitude drive method and (b) locked anti-phase drive method



pairs of switches in an H-bridge circuit (such as S1 and S4 or S2 and S3 in Fig. 3.57) alternate between being ON and OFF according to the duty cycle. If the duty cycle is 50%, the opposite pairs of switches in the H-bridge circuit will close for the same duration in each PWM cycle, resulting in the net voltage applied to the motor leads to be zero and hence no rotation of the motor. If the duty cycle is 60% for example, then a net 20% (60% − 40%) of the available voltage is applied to the motor leads in each PWM cycle. The direction of rotation is set by which motor lead has more positive net voltage in each cycle. Thus when the duty cycle is less than 50%, then the motor will rotate in one direction, and when the duty cycle is above 50%, the motor will rotate in the opposite direction. At both 0 and 100% duty cycles, the maximum voltage is applied to the motor in each case but in opposite directions.

On advantage of the locked anti-phase method is that a single control signal is used to control the speed and direction of a motor load. A disadvantage of this method is that it creates more ripple currents in the motor. In servo drives (such as the one shown in Figure 8.21) or H-bridge drives (such as LMD18200) with dedicated direction and PWM input lines, the PWM input line is wired to high logic, and the PWM control signal is applied to the direction line in using the locked anti-phase drive method. For the L6303 H-bridge driver of Figure 3.59, it is wired as shown in Figure 8.22(b) using the locked anti-phase drive method.

8.3 AC Motors

AC motors can be broadly classified into single-phase and multi-phase ones. A **single-phase AC** motor runs on the electrical power commonly available in homes and light industrial settings. A **three-phase motor** on the other hand uses three similar voltage signals, but each signal is 120° out of phase with the other. Both single-phase and multi-phase motors can be further classified into asynchronous and synchronous types. AC motors are very rugged, are very widely used in industrial applications, and are available in power ratings up to several thousand hp.

The single-phase, asynchronous AC motor, commonly called an **induction motor**, is one of the most widely used AC motors. It is commonly used in applications such as fans, pumps, and compressors. The rotor of an induction motor consists of a stack of thin, flat disks of steel called laminations. The laminations have holes in them through which copper wire is passed and looped around to form a set of continuous coils or windings. The rotor is commonly referred to as squirrel cage (see Figure 8.23).

Figure 8.23

Rotor (squirrel cage) of an AC induction motor (Jouaneh, University of Rhode Island)

The stator is made up of coils to which the electrical leads are connected. The lack of any wiring to the rotor leads to a simple construction that is rugged and requires very little maintenance. When the time-varying AC voltage signal is applied to the stator windings, voltages are induced in the rotor. The induced voltage causes the rotor to rotate, but the rotor rotates at speed that is slightly lower than the speed of rotating stator fields or the synchronous speed. This reduction in speed is called **slip** and is about 3 to 5% for most motors. Because the motor rotates at a speed that is lower than the excitation speed, these motors are known as asynchronous motors. The synchronous speed of an AC motor is a function of the excitation frequency (50 or 60 Hz) and the number of poles in the stator. For four-pole motors at 60 Hz frequency, the synchronous speed is 1800 rpm, and the operating speed is about 1725 to 1750 rpm due to slip. In general, the **synchronous speed** (in rev/min) is given as

$$N_S = 120\frac{f}{p} \tag{8.7}$$

where f is the frequency of the AC power supply in Hz and p is the number of poles. The operating speed is

$$N = N_S\left(\frac{100 - \%\ \text{Slip}}{100}\right) \tag{8.8}$$

In a **synchronous motor** on the other hand, the alternating voltage is supplied to both the stator and rotor windings. This causes the rotor to rotate at the same speed as the excitation frequency, and hence the name synchronous motor. Similar to brush DC motors, means must be provided to connect the voltage leads to the rotating rotor. This is accomplished by the use of a **slip ring** which performs the same function as the commutator ring. Synchronous AC motors are used in timers and instruments and in applications where the speed of several motors must be synchronized such as the case of multiple conveyer belts running together.

One problem of the AC induction motor is that it is **not self starting**. This means that an additional coil or circuitry is needed to cause the motor to start rotating. This additional circuitry causes a net initial torque to be applied to the rotor to give it the initial push. Another limitation of the AC induction motor is that it normally operates at one speed. To allow for variable speed operation of AC motors, adjustable frequency controls are used that vary the frequency of the supply voltage and hence the rotational speed of the motor. Single-phase AC induction motors are available in power rating up to 3 hp. For a higher power rating, three-phase motors can be used.

There are **several varieties of AC single-phase motors** that affect how the motor starts.

Shaded Pole: A short circuit is used to make one side of the field magnetize before the other side.

Split Phase: Uses two windings, one with higher resistance than the other

Capacitor Start: Uses two windings and a capacitor on one of the windings to create a leading phase

The torque–speed characteristics of an AC induction motor depend on the design of the motor. A typical characteristic is shown in Figure 8.24. On the vertical axis, the torque developed by the motor as a percentage of full-load (or rated load) is shown. On the horizontal axis, the rotational speed as a percentage of synchronous speed is shown. Similar to a DC motor, the starting torque is larger than the rated torque. After an initial dip, the motor torque increases with speed until it reaches the breakdown point, at which point the torque start decreasing. The steeper the torque–speed curve after the breakdown point, the more the motor has almost constant speed operation as the load varies.

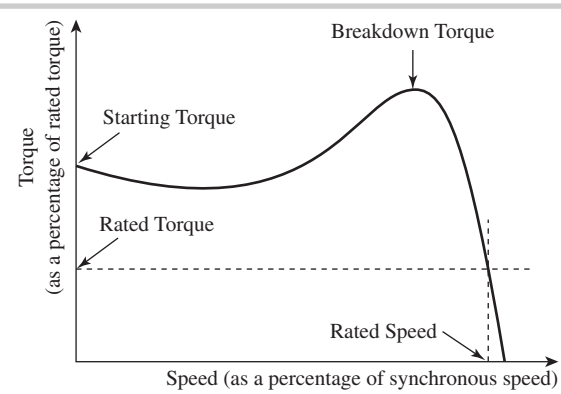


Figure 8.24

Typical torque–speed characteristics of a single-phase AC induction motor

Typical torque–speed data for a 1 hp induction motor made by LEESON Electric are shown in Figure 8.25. Notice how the speed of this motor changes as the load changes. At the rated load (3 lb-ft of torque) and at 230 V operation, the rated speed is 1747 rpm. If the load reduces by 50%, the speed increases to only 1774 rpm. Similarly, if the load increases by 50%, the speed decreases to just 1708 rpm. The motor speed changes by less than +/−2.5% from its rated speed for a load change of +/−50% from the rated load. This "near constant" speed operation under load

Catalog No	110167.00	Model	M6C17DB7J
Product type	AC MOTOR	Stock	Stock
Description	1HP..1725RPM.56.DP./V.1PH.60HZ.CONT.MANUAL.40C.1.15SF.RIGID.GENERAL PURPOSE.M6C17DB7J		

Information shown is for current motor's design — View Outline | View Connection

Engineering Data

Volts	115	Volts	208-230	Volts	
F.L. Amps	12.8	F.L. Amps	6.4	F.L. Amps	
S. F Amps	13.6	S. F Amps	6.8	S. F Amps	
RPM	1800	Hertz	60		
HP	1	Duty	CONTINUOUS	TYPE	CD
KW	.75				
Frame	E56	Serv. Factor	1.15	Phase	1
Max Amb	40	Design	N	Code	K
Insul Class	B	Protection	MANUAL	Therm.Prot.	CEJ50CA
Eff 100%	75	Eff	75%	PF	68
UL	Yes	CSA	Yes	Bearing OPE	0
CC Number		CE	No	Bearing PE	0
Load Type		Inverter Type	NONE	Speed Range	NONE

Performance

Torque UOM	LB-FT	Inertia (WK^2)	.1 LB-FT^2	
Torque	3(Full Load)	6.9(Break Down)	6.7(Pull Up)	9(Locked Rotor)
CURRENT (amps)	6.4(Full Load)	0(Break Down)	0(Pull Up)	33(Locked Rotor)
Efficiency (%)	0(Full Load)	72.8(75% Load)	69.7(50% Load)	56(25% Load)
PowerFactor	0(Full Load)	58(75% Load)	45.6(50% Load)	31.1(25% Load)

Load Curve Data @60 Hz, 230 Volts, 1 Horsepower

Load	Amps	KW	RPM	Torque	EFF	PF	Rise By Resis	Frame Rise
0.0	4.6	0.148	1798	0.0	0.0	14.0	0.0	-
0.25	4.76	0.34	1786	0.75	56.0	31.1	0.0	-
0.5	5.17	0.542	1774	1.5	69.7	45.6	0.0	-
0.75	5.8	0.774	1762	2.25	72.8	58.0	0.0	-
1.0	6.42	0.993	1747	3.0	75.0	67.2	52.0	40.0
1.15	6.85	1.135	1737	3.45	75.0	72.0	56.0	44.0
1.25	7.56	1.271	1728	3.75	72.4	73.1	0.0	-
1.5	8.82	1.552	1708	4.5	70.3	76.5	0.0	-

Figure 8.25

Torque speed data for a 1 hp, single-phase AC induction motor made by LEESON Electric

(Courtesy of Leeson Electric Corporation)

variation is typical of several types of AC motors. Example 8.4 illustrates the use of an AC motor's torque speed characteristics in determining the operating conditions, while Example 8.5 explains some of the characteristics shown in Figure 8.25.

Example 8.4 Operating Conditions of a Load Driven by an AC Induction Motor

Assume that the PM DC-motor in Example 8.1 was replaced by a geared one-quarter hp AC single-phase motor with 20:1 gear ratio. If the motor has the torque–speed data given in Table 8.2 (with no gearing), determine the steady-state lifting speed of the load.

Table 8.2*

Rated Load	0.75 lb-ft					
% of Rated Load	25	50	75	100	125	150
Speed (rpm)	1786	1772	1757	1748	1735	1719

***Data is for LEESON C6C17FK48C, General Purpose, AC single-phase motor under 230V/60Hz excitation**

Solution:

From Example 8.1, the steady-state load torque at the gear output side is 16.7 Nm or

$$16.7 \text{ Nm} \times 1 \text{ lb-ft}/1.36 \text{ Nm} = 12.3 \text{ lb-ft}$$

Because of the 20:1 gear ratio, the load torque as seen by the motor is

$$12.3/20 = 0.615 \text{ lb-ft.}$$

For the above motor, this load represents

$$0.615/0.75 \times 100\% = 82\%$$

of the rated load.

Interpolating from the above table gives the motor speed as

$$1757 + (82 - 75) \times (1757 - 1748)/(75 - 100) = 1754 \text{ rpm}$$

or the lifting speed of the load to be

$$V_{\text{steady}} = \omega\, r = (1754/20 \times 2\pi/60) \times 0.15 = 1.38 \text{ m/s}$$

Example 8.5 Characteristics of an AC Induction Motor

Explain the performance data for the AC motor of Figure 8.25 under 230 V, 60 Hz operation.

Solution:

We will explain the performance data using three load conditions: 50%, 100%, and 150% of rated load. Table 8.3 shows the output mechanical horsepower, the input mechanical horsepower, and the real electrical power delivered to the motor for these three load conditions.

The output mechanical horsepower is given by the formula:

$$\text{Output mechanical power} = T\, n/63025$$

where T is in lb-in and n is in rpm. For a 50% rated load, the output mechanical horsepower is then $\frac{1}{2} \times 3 \times 12 \times 1774/63025 = 0.507$ hp.

The input mechanical horsepower is the output mechanical horsepower divided by the motor efficiency. For the 50% rated load case, the input mechanical horsepower is

$$0.507/0.697 = 0.727 \text{ hp}$$

The real electrical power delivered to the motor should be equal to the input mechanical horsepower. From Equation (2.36), the real power is given as

$$\text{P (in Watts)} = V_{rms}\, I_{rms}\, \text{Power\_factor}$$

For the 50% rated load case, the real electrical power is

$$230 \times 5.17 \times 0.456 = 542 \text{ W} = 0.727 \text{ hp}$$

which is the same as the input mechanical horsepower. The data for the 100% and 150% load given in Table 8.3 show a similar agreement.

Table 8.3

	50% Rated Load	100% Rated Load	150% Rated Load
Output Mechanical Horsepower	0.507	0.998	1.46
Input Mechanical Horsepower	0.727	1.33	2.08
Real Electrical Horsepower	0.727	1.33	2.08

8.4 Stepper Motors

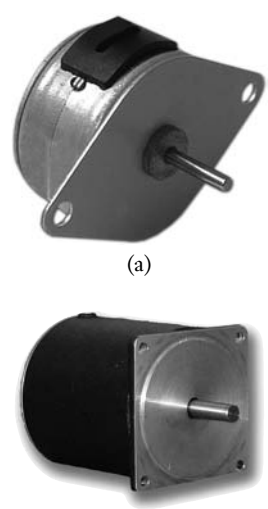
A stepper motor can be classified as a DC motor, since it is driven by non-alternating voltages, but its construction and operation is distinct from a DC motor. Stepper motors, as the name suggests, can move in small angular increments, or steps, ranging from 0.9° per step to 90° per step, depending on the construction of the motor and on how it is driven.

One feature of stepping motors is that they can be used in position control applications without the need for a position sensor. As long as the motor is operated within its specified limits, the nominal position of the stepper motor can be controlled by the number of steps that were sent to the motor. Another feature is that they can be easily controlled with digital circuits, since the stepper motor driver, which generates the appropriate signals to drive the poles of the motor, requires two digital input signals: a pulse signal and a direction signal. A third feature is that there are no wires connected to the rotor, which eliminates the need for brushes and a commutator. A fourth feature is that they can generate a large torque at low speed, which eliminates the need for gears.

There are three types of stepper motors. These are permanent magnet (PM), variable reluctance (VR), and hybrid. Figure 8.26 shows photos of a PM and a hybrid stepper motor. The configurations differ primarily in the construction of the rotor. In a **PM stepper motor**, the rotor is a permanent magnet and has no teeth, while in a **VR motor** the rotor is constructed from non-magnetized soft iron material and has teeth. A VR motor has the advantage of a faster dynamic response, while a PM stepper motor has the capability of exerting a small holding torque,

Figure 8.26

(a) PM and (b) hybrid stepper motors
(Courtesy of Anaheim Automation, Anaheim, CA)



(a)

(b)

called a **detent torque**, when the stator is not energized due to the use of a magnetized rotor. PM motors are widely used in nonindustrial applications such as computer printers and typewriters. A **hybrid motor**, as the name suggests, combines features from both PM and VR motors. Its rotor is a permanent magnet but also has teeth. Furthermore, in a hybrid motor, the magnet is magnetized along the axis of the rotor with the upper half of the rotor having one polarity while the lower half has another polarity. The hybrid configuration is the most widely used in industrial applications. In all configurations, the stator is constructed from pairs of electromagnets commonly referred to as poles.

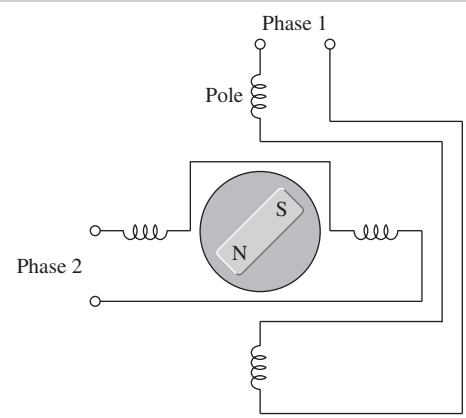
8.4.1 Drive Methods

To understand how stepper motors work, let us first consider the two-phase PM stepper motor shown in Figure 8.27. A **phase** refers to a coil winding; thus, a two-phase motor has two separately activated coil windings, and the coil windings are placed perpendicular to the rotor. This motor has four **poles** with two poles for each phase. The motion of the motor depends on how the stator coils or phases are actuated. There are four possible ways of actuating the phases:

- Wave Drive
- Full Stepping
- Half Stepping
- Micro Stepping

Figure 8.27

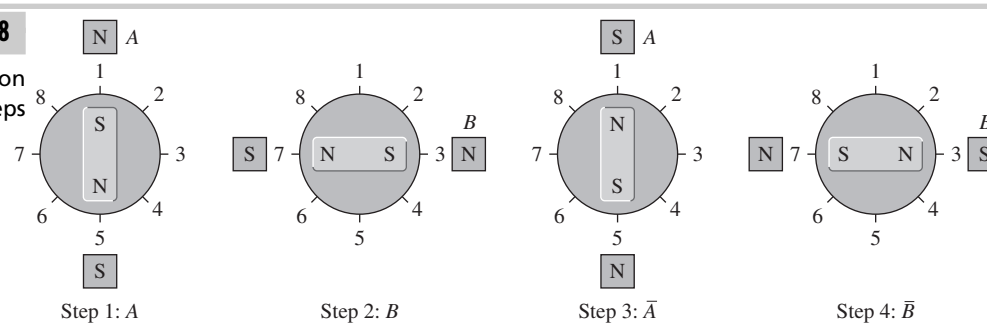
A schematic of a two-phase PM stepper motor



We will illustrate these drive techniques for the two-phase PM motor. Figure 8.28 shows the actuation steps for a **wave drive**. The two phases are labeled *A* and *B*. The rotor goes through the positions 1, 3, 5, and 7 in 90° steps in this drive

Figure 8.28

Wave drive actuation steps



method. To do a complete rotation, the motor has to go through the four steps shown. If the phases are activated in the reverse fashion, the motor will rotate in the opposite direction. Notice that in each of these steps, the rotor as shown is in equilibrium, and moves only if the polarity of the stator coils has changed. Notice also that in a wave drive, only one phase is active or on at a time. This means that only 50% of the available coils are active, which limits the torque applied to the rotor.

The **full-stepping actuation** sequence is shown in Figure 8.29. Here both phases of the stator are active at any point. The resulting motion is similar to wave drive actuation (90° between steps) but the rotor moves through positions 2, 4, 6, and 8 in this case. Due to both phases being on, the torque applied to the rotor is higher in this case than in the wave drive.

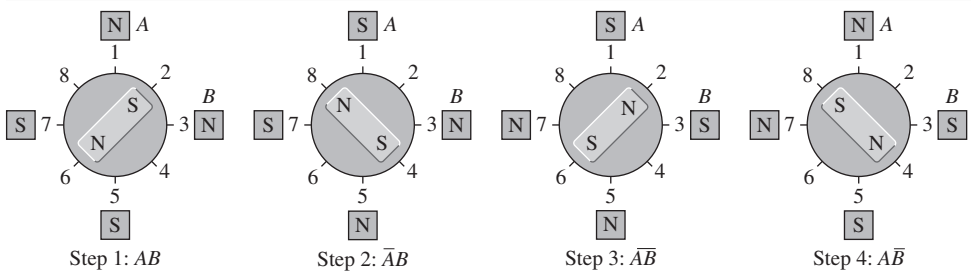


Figure 8.29
Full-stepping actuation

The **half-stepping actuation** sequence is shown in Figure 8.30. This actuation method alternates between activating one phase and two phases at a time, and it takes eight steps to complete one rotation. The rotor is this case travels in 45° steps from position 1 to position 8. Similar to the previous two actuations methods, the direction of rotation can be reversed by simply reversing the sequence of actuation steps.

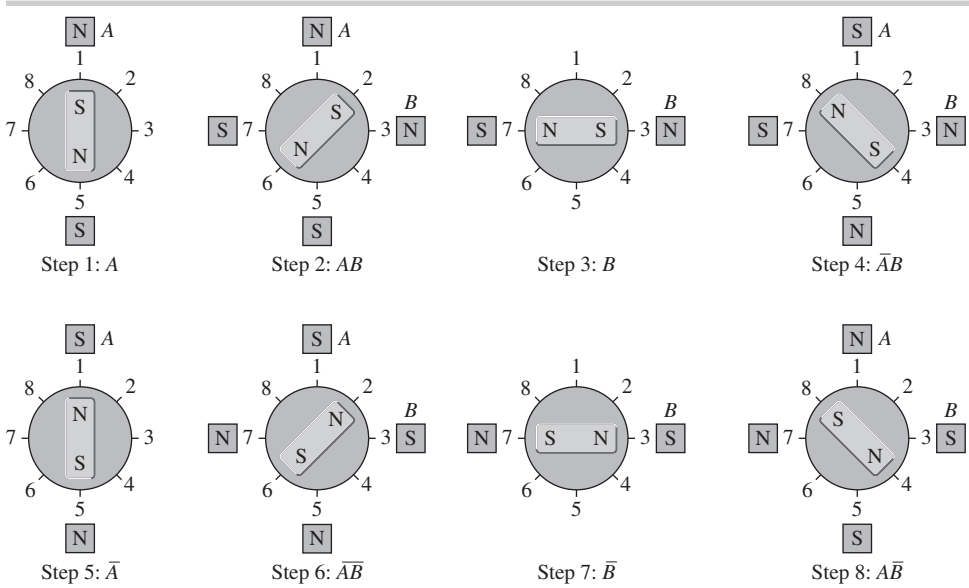


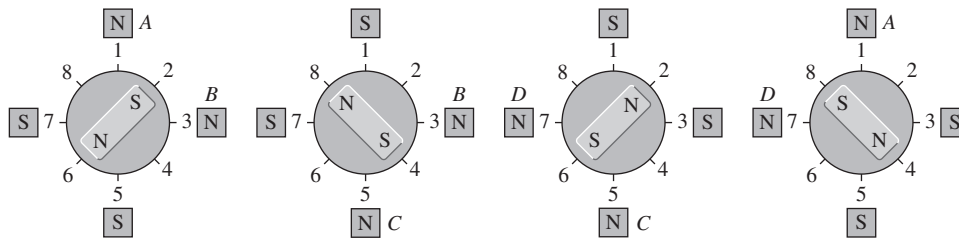
Figure 8.30
Half-stepping actuation

Notice that in the above three drive techniques, the voltage polarity to the stator coil is reversed in some of the steps. This is called **bipolar excitation** and requires supplying a voltage of opposite polarity to the coil. If we are only interested in a unipolar voltage excitation, then we need to use a four-phase motor instead. A

four-phase motor has four coils, each of which can be activated separately. In practice, a four-phase motor is constructed such that there are two coils for each set of stator poles. These two coils are wound in opposite fashion, and only one of them is energized at a time. This is called **bifilar winding** as opposed to **unifilar winding**, which is the winding used with the two-phase motor. Using a four-phase motor, the excitation steps for full stepping are shown in Figure 8.31. The four phases are labeled *A*, *B*, *C*, and *D*. Notice here how the *A* and *C* phases activate the same set of stator poles but in an opposite fashion. Notice also here that only one-half of the coils available are excited at any point of time compared to all the coils in the bipolar two-phase motor case. For example, in step 1 coils *A* and *B* are ON while *C* and *D* are OFF.

Figure 8.31

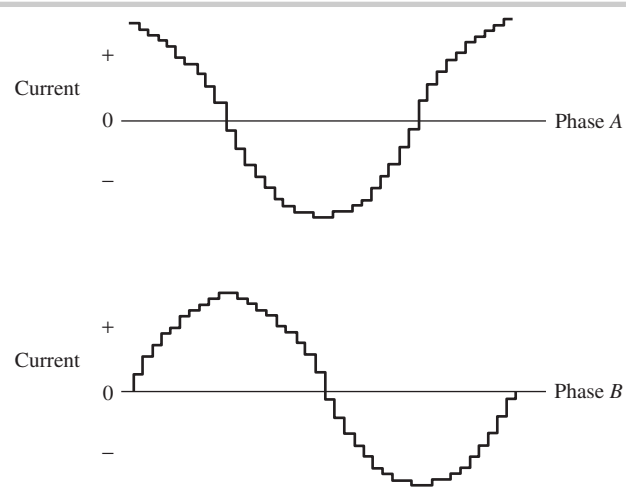
Full stepping excitation for a four-phase unipolar PM rotor



Rather than having the phases fully ON or OFF as illustrated above, in **microstepping drive**, the current to both phases is varied in small steps as shown in Figure 8.32. This allows the motor to have a smaller resolution than that of full or half stepping since it increases the number of equilibrium positions for the rotor. The resolution can be increased by a factor of up to 250 or more in microstepping actuation. Microstepping actuation results in a smoother motion of the motor with less vibration, but the applied torque to the rotor is reduced by 30 to 40% compared to full-stepping actuation.

Figure 8.32

Microstepping excitation



Notice that in the two-phase PM motor discussed above the step angle is 90° for full stepping. In general, the **full-stepping step angle** of a PM stepper motor is given by the formula

$$\Delta\theta = \frac{360}{2PS} \tag{8.9}$$

where

P is the number of rotor pole pairs

S is the number of stator pole pairs

In the two-phase PM motor discussed before, the motor has one rotor pole pair and two stator pole pairs. Thus, its angular resolution is 90° according to Equation (8.9).

For industrial applications, the **hybrid motor** is widely used. A cut-out view of a hybrid motor is shown in Figure 8.33(a). The rotor has two toothed cups, each of which has a separate polarity (N or S). Each cup has 50 teeth that are equally spaced, and the teeth in one cup are offset from those in the adjacent cup by a half a tooth pitch or 3.6°. The stator also has teeth. A typical cross section of a two-phase hybrid motor is shown in Figure 8.33(b). This motor has four poles per phase, with the poles 180° from each other having the same polarity, and those 90° from each other having opposite polarity. In this configuration, the motor advances 1.8° per step in wave or full-stepping mode. The motion of this motor for the different actuation methods is very similar to that of a PM stepping motor shown in Figures 8.28 through 8.30 but replacing the 90° step with 1.8° step.

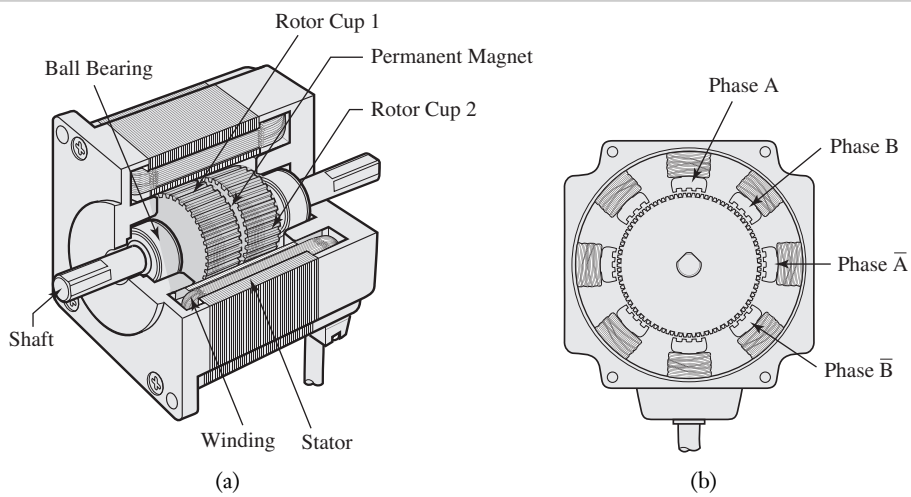


Figure 8.33

Hybrid stepper motor (a) major components and (b) cross-section.
(Images Courtesy of Oriental Motor Corp USA)

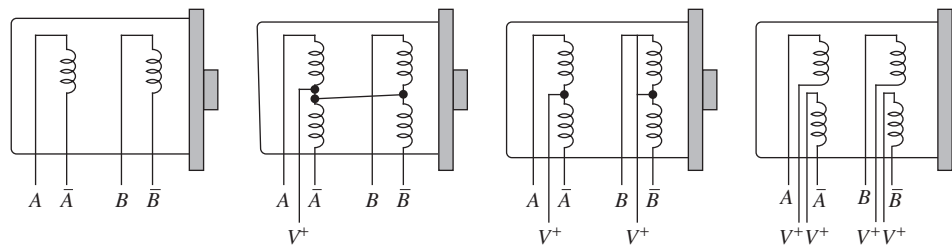
From the previous discussion we see that permanent-magnet stepper motors are constructed somehow similar to BLDC motors but without the use of Hall-effect sensors. In both motor types, the rotor is a permanent magnet, and the stator is made of a number of coils that are activated in sequence. While stepper motors are designed to operate in open-loop fashion, open-loop operation of BLDC motors would result in a very coarse positioning, even in unstable operation.

8.4.2 Wiring and Amplifiers

Stepper motors are available with different lead configurations (see Figure 8.34). The four-lead configuration is not bifilar wound and is only used with bipolar excitation. The six-lead configuration is very commonly used for four-phase unipolar motors, but can also be used with bipolar excitation. Note that in this configuration, one lead serves as a common connection for each pair of bifilar wound coils. The five-lead configuration is not very common. In this configuration, the common connection to all the coils is brought out as one lead. In the eight-lead configuration, the leads of

Figure 8.34

Lead wires for stepper motors



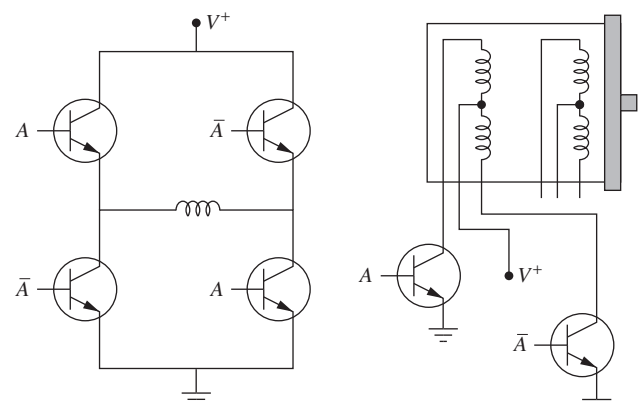
(a) Four-lead (b) Five-lead (c) Six-lead (d) Eight-lead

each bifilar wound coil are brought out separately. This configuration gives the greatest flexibility in the wiring options for the motor.

The driver for a stepper motor can be constructed in different ways. On a basic level, one can use a transistor to drive each phase or coil winding for unipolar drives or an H-bridge for bipolar drives (see Figure 8.35).

Figure 8.35

Unipolar and bipolar drive wiring: (a) bipolar wiring for phase *A* and (b) unipolar wiring for phases *A* and $\overline{A}$



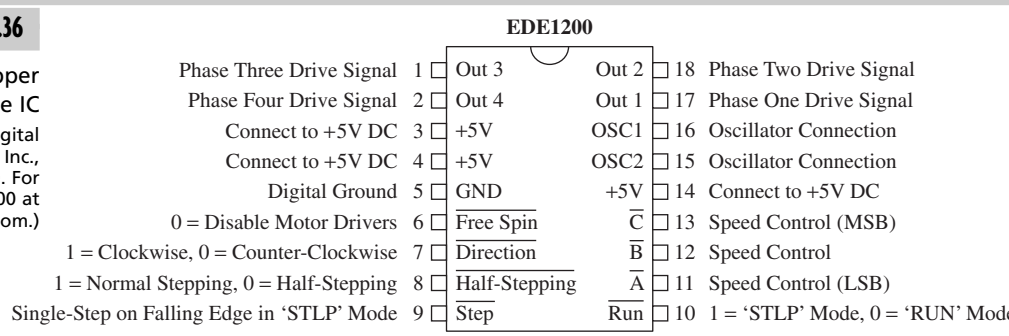
(a) (b)

Alternatively, one can use a commercial stepper motor interface IC (such as the **EDE1200** Unipolar Stepper Motor IC shown in Figure 8.36). The EDE1200 has very low current-output rating (25 mA), and power transistors or a transistor array IC (such as ULN2003A) need to be placed between the EDE1200 output and the motor coils (see Problem 8.9). One nice feature of the EDE1200 is that it can run in two modes: 'Step' mode and 'Run' mode. In the 'Step' mode, the phases are driven in response to external step and direction signals applied to the IC, while in

Figure 8.36

The ED1200 stepper motor interface IC

(Courtesy of E-Lab Digital Engineering, Inc., Independence, MO. For EDE1200 see PDN1200 at www.paladinsemi.com.)



the 'Run' mode, the IC uses an external clock source to self-clock. The pulse rate in the 'Run' mode can be set to one of eight values. A more powerful driver (but now obsolete) is the UCN 5804 translator/driver chip, which supplies a continuous output current of 1.25 A per phase with a 35 V output sustaining voltage.

In industrial applications, a stepper motor drive system is setup as shown in Figure 8.37. It consists of a computer or a programmable logic controller (PLC) that interfaces to an indexer. The indexer in turn interfaces to a driver, which is connected to the stepper motor.

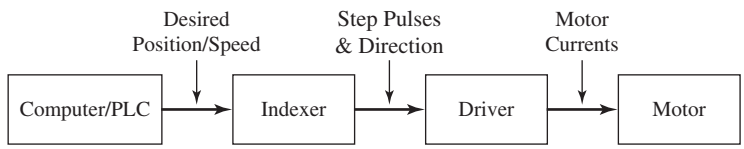


Figure 8.37

Typical stepper motor drive system

The computer or PLC sends the desired position and speed information to the indexer. The **indexer** converts this information into a sequence of pulses at the appropriate frequency and a direction signal. The step pulses and the direction signals are interpreted by the driver to generate the voltages and currents that drive the different phases of the motor to obtain the desired motion. In certain applications, the indexer can be eliminated, and the computer can directly send the step pulses and direction information to the driver. Figure 8.38 shows a commercial stepper motor driver that can operate from 110/220 V outlet power. The driver can

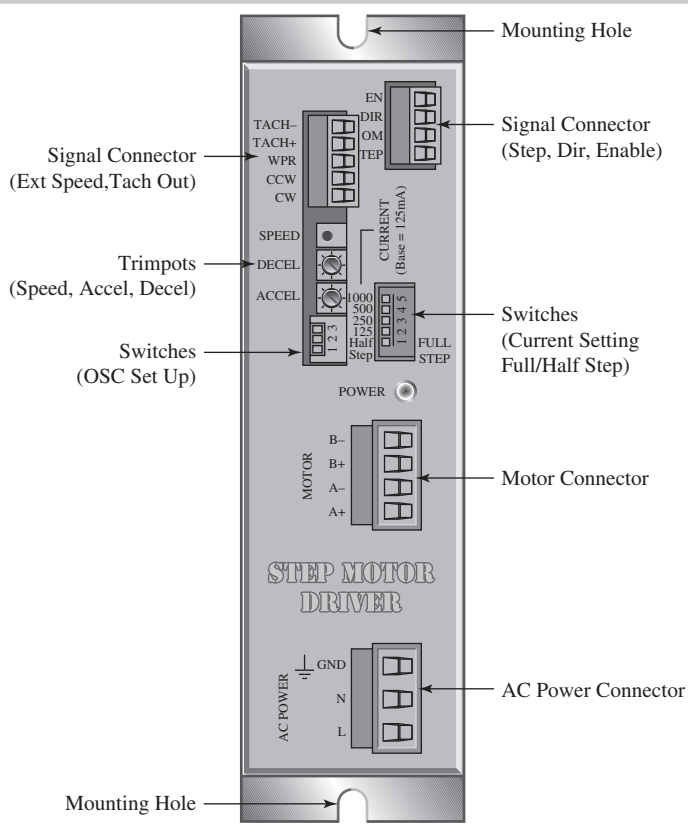


Figure 8.38

Stepper motor driver
(Courtesy of Applied Motion Products, Inc.)

accept pulse and direction inputs or can use the built-in potentiometer to control the speed. The current per phase, as well as the full- and half-stepping modes are set with dip switches. This driver has a dual bipolar H-bridge amplifier. The amplifier outputs are available at the four-position 'motor connector' to which the stepper motor leads are connected. Four-lead stepper motors are simply connected to the four position connecter. For a six-lead motor, the motor leads can be connected in one of two ways: series connection and center tap connection. Figure 8.39 shows these two connections. In the series connection, the motor torque will be higher than the center-tap connection. For an eight-lead motor, many possibilities are available for connection including parallel, series, and two of four windings (see Figure 8.40). Note that the amount of current per winding is set differently for each type of connection.

Figure 8.39

Connections for a six-lead stepper motor with a four-position amplifier: (a) Series connection and (b) center tap connection

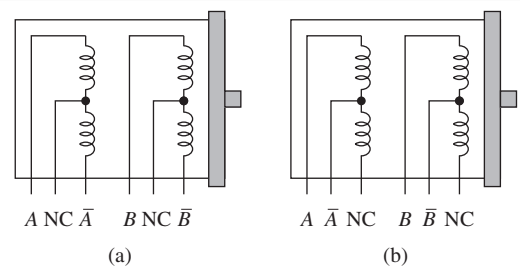
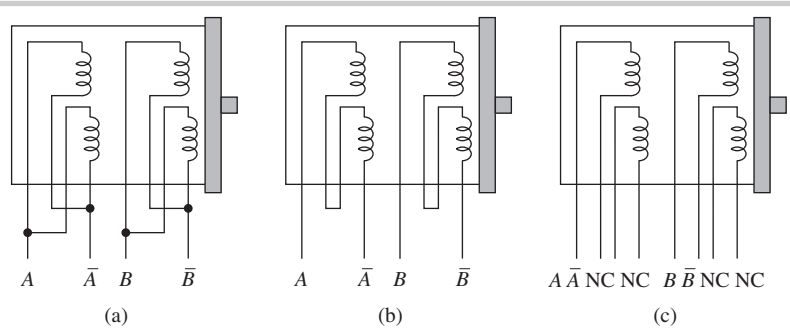


Figure 8.40

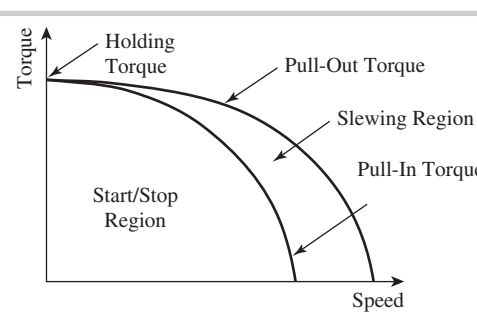
Connections for an eight-lead stepper motor with a four-position amplifier: (a) Parallel connection, (b) series connection, and (c) two windings



Typical torque speed characteristics of a stepper motor are shown in Figure 8.41. The speed is typically given in terms of pulses per second (PPS) or Hz. The figure shows two regions of operation. These are the start/stop region or the locked step region, and the slewing region. In the **start/stop region**, the motor can start, stop, or reverse direction instantly without losing any steps. In the **slewing region**, on the other hand, the motor cannot be instantaneously started, stopped, or reversed. There must be a gradual acceleration of the motor to enter this region,

Figure 8.41

Torque speed characteristics of a stepper motor



and a gradual deceleration of the motor to leave this region. To operate in this region, the motor must start first in the start/stop region. The curve that defines the torque–speed limits of the start/stop region is called the **pull-in torque** curve, while the curve that defines the limits of the slew region is called the **pullout torque**. For a stepper motor, the torque at zero speed is defined as the **holding torque**, which represents the maximum torque that can be applied to a powered (but not rotating) motor without moving it from its rest position and causing spindle rotation. It should be noted that the torque speed characteristics are a function of both the motor and the driver that is used to actuate the motor phases. For a given motor, its torque–speed characteristics will change if the driver was changed or the driving mode was changed. Stepper motor manufacturers usually provide only the pull-out torque curve. Example 8.6 examines the performance data of a commercial stepper motor.

Example 8.6 Stepper Motor Characteristics

A two-phase hybrid stepper motor with six-lead wires has the specifications given in Table 8.4.

Table 8.4*

Connection Type	Holding Torque (N.m)	Rated Current A/phase	Voltage VDC	Resistance Ω/phase	Rotor Inertia (kg.m^2)
Bipolar Series	0.43	0.85	5.6	6.6	68×10^{-7}
Unipolar	0.32	1.2	4	3.3	

*Data is for Oriental Motor PK245-01 motor

Explain the torque and current characteristics of this motor.

Solution:

Since this motor has six leads, it has bifilar windings. The wiring diagram for bipolar-series connection is as shown in Figure 8.39(a), while for unipolar connection it is as shown in Figure 8.34 (six-lead case). Since both coils are active in bipolar wiring, a correspondingly higher holding torque is obtained as shown in the above table. The rated current per phase is simply the voltage divided by the phase resistance or 5.6/6.6 = 0.85 A for bipolar series-connection. Note the resistance for unipolar connection is half that of bipolar-series connection due to the fact that only half of the available coils are activated in unipolar connection.

8.5 Other Motor Types

Universal Motor A **universal motor** is an electric motor that can be operated using both AC and DC voltage signals. It is mostly used in hand tools such as drills and in appliances such as vacuum cleaners, mixers, and blenders. The motor uses brushes for commutation and its construction is similar to that of a series-wound DC motor with a wound rotor and a wound stator. The current in the rotor and the field coils changes polarity at the same time, making the direction of the resultant force acting on the rotor constant. The universal motor is also known as an **AC series motor** or an **AC commutator motor**. One feature of the universal motor is that it allows variable speed control of the motor similar to a DC motor but using

AC power, thus eliminating the need for an AC to DC transformer. The speed control can be implemented using phase control with SCR (see Chapter 3), a rheostat, or a chopper drive (uses PWM duty cycle to control the effective voltage). Universal motors have a high power-to-size ratio, and they have a high no-load speed (20,000–40,000 rpm) which is much higher than the line frequency of either 50 or 60 Hz. However, a DC motor of the same size as a universal motor is more efficient than a universal motor.

Servo, Gear, and Brake Motors Many of the motors that were discussed before can be referred to by different names depending on the application. For example, a **servomotor** is a motor that is equipped with either a position or velocity feedback device to be used in closed-loop control applications. The motor itself can be a DC or AC type. Another example is a **gear motor**, which is a motor that has a gear attached to it to reduce the speed of the motor and increase the output torque of the motor. A third example is the **brake motor**, which has an attached brake to prevent the shaft from rotation when the power is disconnected to the motor.

Figure 8.42

A standard size hobby servo motor
(Courtesy of Hitec RCD USA, Poway, CA)

Hobby Motors A special class of motors is called **hobby** or **RC servo motors**. These motors are widely used in radio-controlled cars, planes and boats. A typical standard hobby servo motor is shown in Figure 8.42. Hobby servo motors are relatively inexpensive, are driven by low voltages (about 5 VDC), and are available in several sizes including standard, mini-micro, and quarter scale. The standard size is the most common, and has the advantage that its physical size and mounting holes are the same regardless of the manufacturer. The mini-micro size is half or smaller than the size of the standard servo. The rotational speed of the hobby servo is about 0.2 s for 60° of angular travel.

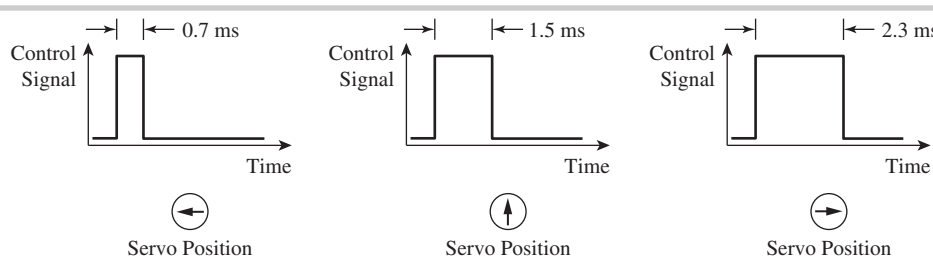
The hobby servo motor actually consists of four components that are packaged together. These are: a small DC motor, a gear reducer, a potentiometer, and a control board. These servos have three leads labeled power, ground, and control signal. The power signal ranges from 4.8 to 7.2 V and can be conveniently obtained from battery power packs. The control signal is a PWM signal (5 V) at a frequency of 20 to 60 Hz. This signal can be conveniently generated from a microcontroller or a commercial control board.

The pulse width of the control signal, which ranges from about 0.7 to 2.3 ms for most servos, controls the position of the servo. At 0.7 ms, the servo is at one extreme of its motion range, while at 2.3 ms it is at the other extreme. At 1.5 ms pulse width, the servo is in the center or mid-position. Most servos have a motion range of ±90°. Figure 8.43 shows the typical relationship between the pulse width and the hobby servo position.

Hobby servos operate in closed-loop position control fashion (see Chapter 9). On the control board, the duration of the PWM control signal is converted to a voltage signal. This voltage signal is compared with the voltage output from the

Figure 8.43

Hobby servo position as a function of pulse width



potentiometer that is connected to the motor shaft. The difference voltage between these two signals is then used to drive the motor so the error between these two signals goes to zero.

8.6 Actuator Selection

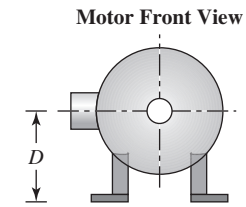
When selecting an electric motor for a mechatronic application, the selection should include the type of motor (DC or AC), the power rating and speed, the operating voltage and frequency, the motor frame size, and mounting details.

To provide means for interchangeability of motors, commercial motors are made in standard mounting sizes called **NEMA** (National Electrical Manufacturers Association) **frame sizes**. Small size motors have a NEMA frame sizes of 17 to 56, while large size motors have NEMA frame size of 240 or higher. The NEMA frame size specifies only the size of the mounting frame and not the motor body diameter. The two or three digit NEMA frame size specifies the '*D*' dimension of the motor which is the distance between the center of the motor shaft and the bottom of the base mount (see Figure 8.44). For a two-digit frame size, the '*D*' dimension distance in inches is obtained by dividing the frame size by 16. For a three-digit frame size, the '*D*' dimension is obtained by dividing the first two digits by 4. For example, a 34 frame size motor has a '*D*' dimension of 2.125 in. and a 405 frame size has a 'D' dimension of 10 in.

Motors can also differ in the details of their mounting and the type of enclosure they have. Common **mounting configurations** include foot mounted, cushion base, C-Face mounting, and vertical mounting. The common **enclosure types** are shown in Table 8.5. There are many things that should be considered when selecting an actuator. The most important of these would be that the torque-speed characteristics of the actuator should match that of the desired application. Secondary considerations include control method, cost, size, and ease of maintenance. Table 8.6 compares DC, AC, and stepper motors in several categories.

Figure 8.44

Illustration of the NEMA '*D*' dimension



Enclosure Abbreviation	Description	Comments
ODP	Open Drip Proof	For normal applications where low cost is important
TEFC	Totally Enclosed Fan Cooled	Commonly used motor enclosure. The enclosure is dust tight but cannot stand high water pressure. Uses a fan to cool the motor
TEAO	Totally Enclosed Air Over	Normally used with fans or blowers and utilizes the air drawn by the fan or blower for cooling
TEBC	Totally Enclosed Blower Cooled	Uses a blower for cooling. The enclosure is dust tight but cannot stand high water pressure
TENV	Totally Enclosed Non-Ventilated	For use with small hp rated motors and utilizes fins on the motor body for cooling
TEWC	Totally Enclosed Water Cooled	Expensive motors that utilize a double-shell body through which water flows for cooling

Table 8.5

Common enclosure type for electric motors

Table 8.6

Comparison of brush DC, brushless DC, AC, and stepper motors

Characteristic	Brush DC	Brushless DC	AC	Stepper
Supply voltage	Needs a simple DC voltage power source	Uses DC voltage but uses a special amplifier	Can be readily run from the AC line voltage	Uses DC voltage but requires special amplifier to drive each phase
Direction change	Easily done by reversing polarity to the motor leads	By activating the phases in a reverse fashion	By changing the wiring in the starting circuitry in single-phase motors, and by changing two of the phases in three-phase motors	By activating the phases in a reverse fashion or by changing the direction signal if a stepper driver is used
Speed change	Easily done by changing the value of the input voltage	By changing the rate of activating the phases	More difficult to do and requires a variable-frequency input device	By changing the rate of activating the phases or the pulse rate if a stepper driver is used
Starting	Self starting	Self starting	Single-phase motors are not self-starting and need a special starting winding/circuitry. Three-phase motors are self-starting	Self-starting
Maintenance	Need to periodically replace the brushes and resurface the commutator	No need to replace brushes or resurface the commutator	Requires less maintenance especially for AC induction motors	No wear problems due to the absence of brush contact
Available sizes	Few watts to several hp rating	Few watts to few hundred hp	Single-phase up to few hp and multiphase up to several thousand hp	Stepper motors do not have a hp rating since they do not rotate continuously. The equivalent max hp rating is a fraction of 1 hp. They could have torque rating up to a few thousand oz-in.

8.7 Chapter Summary

This chapter focused on electric actuators. The chapter covered brush and brushless DC motors, AC motors, universal motors, stepper motors, and hobby servo motors. Brush DC motors are commonly used in many consumer and industrial applications, and they use brushes to transfer the electric current to the rotating coil to allow continuous flow of current in the same direction as the rotor rotates. Brushless DC motors are more reliable than brush DC motors. In a brushless DC motor, the rotor is made of permanent magnets, and the stator is constructed of coils. There is no wiring to the rotor. Brushless DC motors use electronic commutation rather than mechanical commutation. AC motors are very rugged, are very widely used in industrial applications, and are available in power ratings up to several thousand hp. A universal motor is an electric motor that can be operated using both AC and DC voltage signals. Stepper motors are typically used in open-loop position control applications because they can operate without the need for a position sensor. They move in small angular increments or steps. Hobby servo motors are widely used in radio-controlled planes and boats, and they consists of four components that are packaged together. These are a small brush PM DC motor, a gear reducer, a potentiometer, and a control board.

The operating principles of each type of actuator are given as well as the torque-speed characteristics. Driving methods, amplifiers, and drive circuitry are also provided for some of the actuators. The information given in this chapter will enable the user to select the appropriate actuator for a given mechatronic application.

Questions

8.1 What type of DC motor has linear torque–speed characteristics?

8.2 What type of motor has a nearly constant speed over a large torque range?

8.3 Compare brush DC and brushless DC motors.

8.4 Explain what is meant by 'slip' in AC induction motors.

8.5 List several advantages of stepper motors.

8.6 Compare brushless DC and stepper motors

8.7 What type of driver is used to drive brushless DC motors?

8.8 Explain what is meant by microstepping.

8.9 What is a universal motor?

8.10 What type of sensor does a hobby servo motor have?

8.11 What is meant by NEMA 34 size motor?

Problems

P8.1 Research and identify the type of actuator used in the following devices. List also the approximate power rating and nominal voltage level for the actuator.

a. Kitchen sink garbage disposal

b. Powered car window

c. Powered car mirror

d. Food blender

e. Cordless electric drill

f. Camera focus system

g. Hybrid electric vehicle

h. Residential garage door opener

P8.2 Suggest brush DC, brushless DC, AC, or stepper motors for the following applications and explain your selection.

a. Low-maintenance, outdoor, and constant speed operation

b. Low heat generation and controlled speed operation

c. Low-cost and controlled position operation

P8.3 Research and explain how a) the speed and b) the direction of an AC induction motor can be changed.

P8.4 Determine the operating speed and power of a PM DC brush motor with the following operating characteristics: Starting torque = 200 oz-in. no-load speed = 4000 rpm, load = 80 oz-in.

P8.5 A geared PM DC brush motor has a gear ratio of 50:1. The input speed is 4000 rpm at an input power of one-half horsepower. Determine the output torque of this motor, assuming a 5% power loss in the gear drive.

P8.6 Show that the maximum power for a PM brush DC motor is obtained when the motor is operating at a speed equal to half the no-load speed.

P8.7 A stepper motor has a specification of 200 steps/revolution. The motor shaft drives a linear positioning table using a lead screw with a lead of 0.1 in./rev. Determine the linear speed of the table if the motor is operated in half-stepping mode at a rate of 500 pulses/s.

P8.8 A PM stepper motor has a 45° step angle in full stepping mode. If the motor has two stator pole pairs, determine the number of rotor pole pairs that this motor has.

P8.9 Draw a wiring diagram to drive a four-phase unipolar stepper motor using a PIC MCU and a stepper motor IC, such as the EDE1200. Use an interface IC such as ULN2003A between the stepper IC and the motor coils so that a stepper motor with a current requirement of up to 500 mA per phase can be driven by this circuit. Set the circuit so that the motor can run in half-stepping mode. Refer to the data sheets for the used ICs for pin details.

P8.10 Research and explain how a hobby servo motor uses the pulse width of the control signal to control the desired position of the motor.

P8.11 A 5-hp induction motor with 80% efficiency is operated from a 220 V line. If the power factor is 0.7, determine the amount of current drawn by the motor.

P8.12 A PM DC brush motor (data is for Pittman 14207/24V motor) is used to drive a lead-screw table motion system similar to that shown in Example 7.2. The table has a mass of 5 kg, the combined inertia of the coupling and the lead screw is 2.0×10^{-4} kg m^2, the viscous damping coefficient due to the bearings is 0.001 N · m.s/rad, and the lead is 2 mm. The motor has the following specifications:

Continuous (nominal) torque =	0.353 N · m
Speed at continuous torque =	2810 rpm
Stalled (peak) torque =	2.85 N · m
No-load speed =	3160 rpm
Rotor inertia =	4.73×10^{-5} kg m^2

Determine the maximum acceleration of the table. Make any reasonable assumptions in solving this problem.

Laboratory/Programming Exercises

L/P8.1 Implement the GUI for the DC-motor control problem shown in Figure 6.55 in MATLAB or VBE. Use a software model to simulate the dynamics of the motor to be controlled.

L/P8.2 Use any PIC microcontroller to drive a stepper motor by outputting pulse and direction signals to the stepper motor driver. Use the PWM feature in the MCU to generate the pulses. Play with setup features of the PWM function to allow the stepper motor to be driven at different frequencies.

L/P8.3 Use any PIC MCU and four transistors to build a system that drives a four-phase stepper motor. In this exercise, each phase of the motor is driven by a transistor using control signals that are sent by the PIC MCU. The PIC MCU uses one digital output line for each phase. The code inside the MCU should send out timed signals (use a short delay between actuation of the different phases(s)) to drive the four phases of the motor in either full or half-stepping modes. The full/half stepping mode is set by a digital I/O line that the user sets high/low.

L/P8.4 Use any PIC microcontroller to drive a hobby servo motor such as the Hitec HS-311 by outputting a pulse signal to the hobby servo motor. Use the PWM feature in the MCU to generate the pulse. Use a rotary potentiometer to adjust the pulse width (or duty cycle) setting of the control signal to change the angle of the motor. When the rotary pot is at one extreme of its travel, the pulse width setting should be at its minimum setting. Turning the pot clockwise from that position should increase the pulse width setting. When the rotary pot is at the other extreme of its motion, the pulse width setting should be at its maximum setting. Implement an infinite do-loop to read the desired pulse width and to adjust the duty cycle of the PWM signal. Add a small delay (100 ms) in each run through the loop.

CHAPTER 9

Feedback Control

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- Explain the difference between open- and closed-loop control systems
- Derive the closed-loop transfer function of a control system
- Obtain the steady-state error for first- and second-order systems under P, PI, or PID control
- Explain the digital implementation of a PID controller
- Simulate in Simulink a closed-loop control system
- Explain the effect of nonlinearities on control system behavior
- Explain the operation of a state feedback controller

9.1 Introduction

Modern society depends on feedback control systems for the luxury and convenience of living. From thermostat-controlled heating and cooling systems to high-speed elevators, feedback control is the key behind these conveniences. A feedback control system is one that tends to maintain a prescribed relationship between the output and the reference input by comparing these and using the difference as a means of control [37]. The components of a feedback control system are the controller, which is the mind of the system; the plant or process to be controlled; and the measuring element or sensor. In the previous chapters we have covered the details, models, and working principles for many of these components. For example, in Chapter Four, we covered microcontrollers which are commonly used as the computing medium to implement controllers. In Chapter Five, we discussed data acquisition and interfacing of digital systems with analog components, which is needed in digital implementation of controllers. In Chapter Six, we discussed control software structures for the implementation of discrete-event and feedback control systems. In the previous two chapters, we have covered sensors and actuators that are key elements in feedback control systems.

This chapter covers the basics of feedback control systems. The objective is to illustrate to the reader the design, simulation, and implementation of feedback control systems. The chapter starts by comparing open- and closed-loop control methods followed by a review of basic feedback control topics. It then discusses the PID controller, one of the most widely used controllers. For the PID controller, the chapter uses as an example the velocity and position control of a simple inertia

system. It discusses a dynamic model for the system and how the model parameters can be experimentally identified by performing an open-loop step input speed response measurement. It then discusses the effect of different control actions such as P, PD, PI, or PID on the response of the system. These control actions are illustrated using MATLAB simulations. The implementation logic for a digital PID controller with no reset-windup for the I-action is also discussed. Nonlinear control effects and other control schemes such as the on-off controller and state feedback controller are also discussed in this chapter.

9.2 Open- and Closed-Loop Control

Before we discuss in detail feedback control systems, which are also referred to as closed-loop control systems, let us contrast them with open-loop control systems. In an **open-loop system**, the actual output of the system does not influence the input to the system, while in a **closed-loop system** the input to the system is a function of *both* the actual output and the reference input. The terms open- and closed-loop come from the fact that in control systems each element in the control system is represented by a block that symbolizes the input–output relationship for that element. In an open-loop control system (see Figure 9.1), there is no closed loop that connects the blocks in the system while in a closed-loop control system (see Figure 9.2) there is one.

Figure 9.1

Block diagram of an open-loop control system

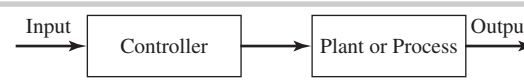
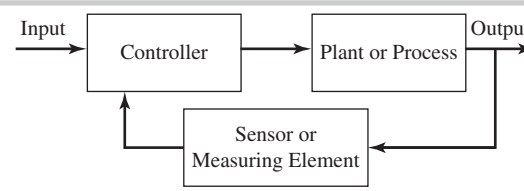


Figure 9.2

Block diagram of a closed-loop control system



Examples of open-loop control systems include the motion of a stepper motor–driven stage with no feedback sensors, or the operation of a window air-conditioner with no thermostat control. While simpler to implement than closed-loop control systems since stability of the control system is not an issue here, open-loop control systems have the following limitations:

- Accuracy of the system depends on having a proper calibration between the input and output, and the system has to be re-calibrated if the operating parameters change such as load variation in motor-driven systems.
- System performance degrades if there is any internal or external disturbance acting on the system such as noise, dirt, temperature fluctuation, or wear.

On the other hand, a closed-loop control system does not suffer from these limitations.

9.3 Design of Feedback Control Systems

To design a feedback control system means to select a particular control law and to determine the parameters of the controller such that acceptable transient and steady-state performance parameters are obtained for the system. Some of these performance parameters include overshoot, rise time, and steady-state error. Due to the possibility of instability in closed-loop control systems, which causes an unbounded output for a given bounded input, it is very important to analyze and simulate the designed feedback controller to make sure that it behaves as designed before it is implemented on the real system. Otherwise, breakage or damage to the physical system could occur if the controller is unstable.

In order to analyze and simulate a control system, a dynamic model of the system to be controlled needs to available. Dynamic models can either be obtained from application of the appropriate laws of physics such as Newton's law or from fitting experimental response data with mathematical models. Having a model of the system does not guarantee that the real system will behave as the simulated system, because for many systems due to real-life effects such as friction, it is difficult to obtain simple models that can accurately capture the dynamic response of the system.

While all mechanical plants or processes to be controlled such as motors, heaters, and mixing tanks are continuous-time systems, the controller for such a system can be either an analog type or a digital type. **Analog controllers** are implemented using analog circuit components such as operational amplifiers (see Section 2.9), resistors, and capacitors, while digital controllers are implemented using computers and microcontrollers (see Chapters 4 and 6). **Digital controllers** have the feature that the gains of the controllers can easily be changed in software by changing the values of the parameters in the control law without the need to replace circuit components or to rewire circuits. Almost all modern control systems are implemented using digital controllers. When a digital controller is used, means must be provided to interface the digital controller with the continuous-time plant such as through an A/D or a D/A converter (see Chapter 5). Because a digital controller is an example of a discrete-time system, the controller effectively operates in closed-loop fashion only at the sampling instances, and the system is in open-loop fashion the rest of the time. However, with the use of modern processors that allow high sampling rates, the controller can be modeled as a continuous-time system without much loss of accuracy. Therefore, in the remainder of this chapter, we will only consider continuous-time models of plants and control laws.

Once a model is available for a control system, analysis can be performed in either state space form (or differential equation form) or in algebraic form (if the system is linear) using Laplace transform methods. For single-input, single-output (SISO) systems, it is usually easier to perform the analysis in the algebraic domain. The Laplace transform is an operator that converts differential equations into algebraic equations. All control textbooks have coverage of this topic; see for example reference [38], so its principles will not be covered here.

9.4 Control Basics

This section covers some basic control systems material. We start by discussing transfer functions and poles and zeros of transfer functions. The transfer function defines the input output relationship for each block in a control system, and thus we need to have a transfer function for each block in order to obtain a mathematical model of the entire system (see Appendix C which discusses transfer functions

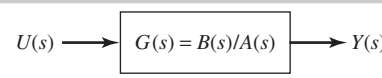
in the context of solving dynamic models in MATLAB). The **transfer function** of a linear, time-invariant system is defined as the ratio of the Laplace transform of the output to the Laplace transform of the input under the assumption of zero initial conditions. The transfer function is written as a ratio between two polynomials, $B(s)$ and $A(s)$, using the Laplace variable s shown in Equation (9.1):

$$G(s) = \frac{B(s)}{A(s)} = \frac{b_0 s^m + b_1 s^{m-1} + \cdots + b_m}{a_0 s^n + a_1 s^{n-1} + \cdots + a_n} \tag{9.1}$$

The roots of the numerator polynomial ($B(s)$) are called the **zeros** of the transfer function, and they have a direct effect on the shape of the response. The roots of the denominator polynomial ($A(s)$) are called the **poles** of the transfer function whose value determines whether the system is stable or not. All poles have to have a negative real-part in order for the output of the transfer function to be stable. A block diagram representation of a transfer function is shown in Figure 9.3, where $U(s)$ is the input and $Y(s)$ is the output.

Figure 9.3

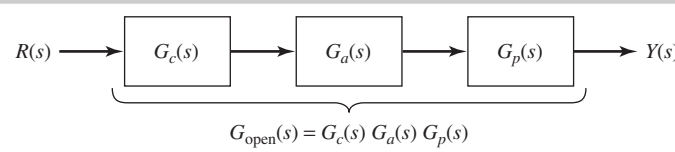
Block representation of a transfer function



If we have three blocks in series, one to represent a controller with a transfer function $G_c(s)$, another to represent the actuator with a transfer function $G_a(s)$, and a third that represents a plant with a transfer function $G_p(s)$ as seen in Figure 9.4, then the overall transfer function that represents the effect of the controller, actuator, and plant combined together is given by the product of the three transfer functions $G_c(s)$, $G_a(s)$ and $G_p(s)$.

Figure 9.4

Combined transfer function, $G_{open}(s)$

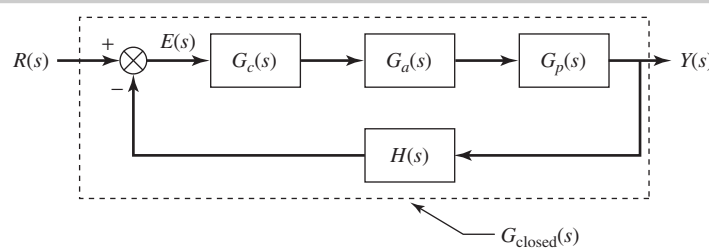


Now assume we added a sensor with a transfer function $H(s)$ to measure the output of the plant, and then we used the sensor signal to place the system under closed-loop control as shown in Figure 9.5. Using block diagram rules, the output of the system, $Y(s)$, is given by

$$Y(s) = G_p(s)G_a(s)G_c(s)E(s) = G_p(s)G_a(s)G_c(s)(R(s) - H(s)Y(s)) \tag{9.2}$$

Figure 9.5

Overall closed-loop transfer function



Where $E(s)$ is the error and is defined as $R(s) - H(s)Y(s)$. The **overall closed loop transfer function**, $G_{closed}(s)$, is defined as the ratio between the actual output of the system, $Y(s)$, and the reference input, $R(s)$. This ratio can be obtained from Equation (9.2) and is

$$G_{closed(s)} = \frac{Y(s)}{R(s)} = \frac{G_p(s)G_a(s)G_c(s)}{1 + H(s)G_p(s)G_a(s)G_c(s)} = \frac{G_{open(s)}}{1 + H(s)G_{open(s)}} \quad (9.3)$$

Note that $G_{closed}(s)$ is equivalent to the effect of all the components in the dashed block in Figure 9.5. Note also that the form of the closed loop transfer function. In the numerator, it has the product of all the blocks in the forward loop between $R(s)$ and $Y(s)$ *or* $G_{open}(s)$, while in the denominator, it has the form of 1 plus the product of all the blocks in the feedback loop between $R(s)$ and $Y(s)$, *or* $H(s)G_{open}(s)$.

The overall closed-loop transfer can be first, second, or a higher order, depending on the plant dynamics model and the type of controller used. Example 9.1 illustrates the determination of the closed-loop transfer function for a system represented in a block diagram.

Example 9.1 Overall Closed-Loop Transfer Function

Determine the overall closed-loop transfer function for the control system represented with the block diagram shown in Figure 9.6.

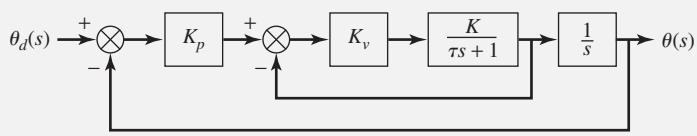


Figure 9.6

Solution:

The given block diagram has two loops, an inner and an outer loop. From Equation (9.3), the closed-loop transfer function for the inner loop is given by

$$G_{inner}(s) = \frac{K_v \dfrac{K}{\tau s + 1}}{1 + K_v \dfrac{K}{\tau s + 1}} = \frac{KK_v}{\tau s + KK_v + 1} \quad (1)$$

The block diagram can now be represented as shown in Figure 9.7 with only one loop.

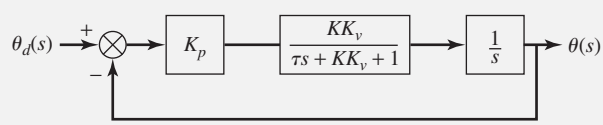


Figure 9.7

The overall transfer function is:

$$G_{overall}(s) = \frac{\theta(s)}{\theta_d(s)} = \frac{K_p \dfrac{KK_v}{\tau s + KK_v + 1}\dfrac{1}{s}}{1 + K_p \dfrac{KK_v}{\tau s + KK_v + 1}\dfrac{1}{s}} = \frac{K_p KK_v}{\tau s^2 + (KK_v + 1)s + K_p KK_v} \quad (2)$$

The transfer function is second order and represents the dynamics between the desired position $\theta_d(s)$ and the actual position $\theta(s)$.

9.5 PID Controller

The proportional, integral, derivative (PID) controller it is one of the most widely used controllers in industry. More than 90% of all industrial controllers are implemented using this popular control law [39]. In continuous time, the controller takes the form:

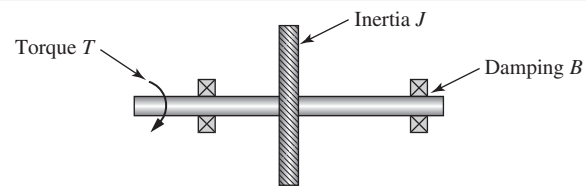
$$Y(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{d}{dt} e(t) \tag{9.4}$$

As seen in Equation (9.4), the output of the controller is proportional to the error signal (P-term), the integral of the error signal (I-term), and the derivative of the error signal (D-term) through the gains K_p, K_i, and K_d, respectively. The transfer function of the PID controller is given by Equation (9.5).

$$Gc(s) = \frac{Y(s)}{e(s)} = (K_p + K_{i/s} + K_d s) \tag{9.5}$$

In practice, variations of the above control law are also implemented, such as a **PI controller**, which has only the P and I terms, or a **PD controller**, which has only the P and D terms. To illustrate the effect of different control actions on the response of a mechanical system, we will consider the problem of speed and position control of a simple inertia as shown in Figure 9.8.

Figure 9.8
Simple inertia model



This model can represent the dynamics of an inertia load driven by a PM DC motor which has a small inductance (see Example 9.2). In this case the dynamics of the electrical parts of the motor can be neglected, and only the dynamics of the inertia element need to be considered. The equation of motion of this system is given by

$$T = J\ddot{\theta} + B\dot{\theta} \tag{9.6}$$

where θ is the angular position of the inertia, J is the inertia of the rotating parts, and B is the viscous damping coefficient. The transfer function between the torque input, $T(t)$, and speed, $\omega(t)$, is given by

$$\frac{\omega(s)}{T(s)} = \frac{1}{Js + B} = \frac{1/B}{J/Bs + 1} = \frac{K}{\tau s + 1} \tag{9.7}$$

Note in the above form, τ is the time constant for this first-order system and is given by the ratio of J to B. The **time constant** is defined as the time it takes to reach 63.2% of the final steady-state output for a step change in input, and the smaller the time constant, the faster the response of the system. When this first order system is subjected to a unit torque step input, the output speed is given by

$$\omega(s) = \frac{K}{\tau s + 1} \frac{1}{s} \tag{9.8}$$

In the time domain, the output is given by Equation (9.9):

$$\omega(t) = K(1 - e^{-t/\tau}) \tag{9.9}$$

A plot of this equation is shown in Figure 9.9 for a system with $J = 0.7$ kg m^2 and $B = 0.5$ N · m s/rad. These same parameters are used for all the simulations shown in this chapter.

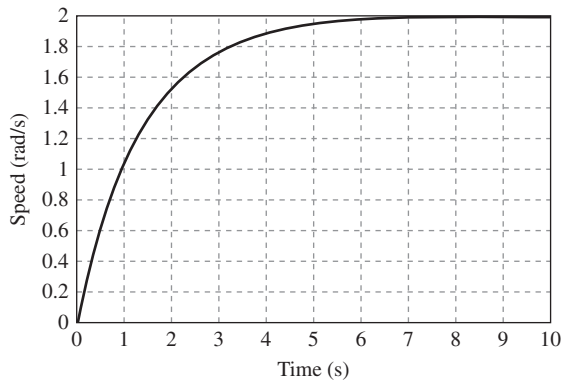


Figure 9.9

Open-loop unit step response of a simple inertia

Notice how the final steady value of this system is a function of the parameter K, and is equal to K (2 in this case) for a unit step input. From differentiating the expression for the speed and evaluating it at time zero, we see that the slope of the speed response curve at time zero is the ratio of K to τ. This fact can be used to experimentally determine the parameters K and τ of a first-order system from a step response plot.

9.5.1 Speed Control of an Inertia

Let us place the simple inertia system under a closed-loop proportional controller. The block diagram for this case is shown in Figure 9.10. The closed-loop transfer function for the overall system is $K_p/(Js + B + K_p)$ and is still a first-order system.

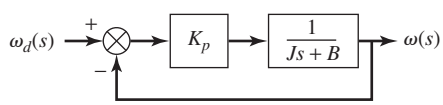


Figure 9.10

Simple inertia under closed-loop P-control

Under a unit step input, the output of this system is shown in Figure 9.11. As seen in the figure, the final steady-state value does not reach the reference input, and the system has a steady-state error.

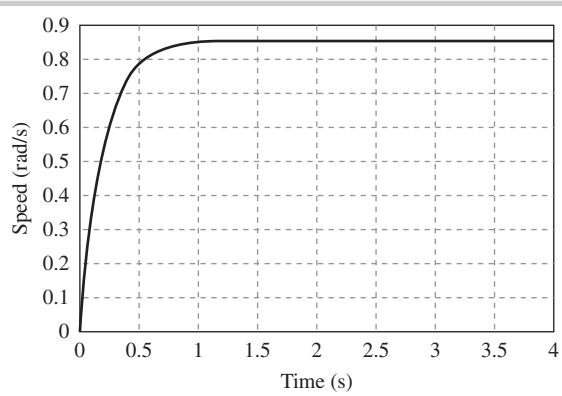


Figure 9.11

Speed response under closed-loop P-control ($K_p = 3$)

The value of this steady-state error can be obtained by first applying the final value theorem to determine the final output as time tends to infinity:

$$\lim_{t\to\infty} \omega(t) = \lim_{s\to 0} s\omega(s) = \lim_{s\to 0} s\frac{K_p}{Js + B + K_p}\frac{1}{s} = \frac{K_p}{B + K_p}$$

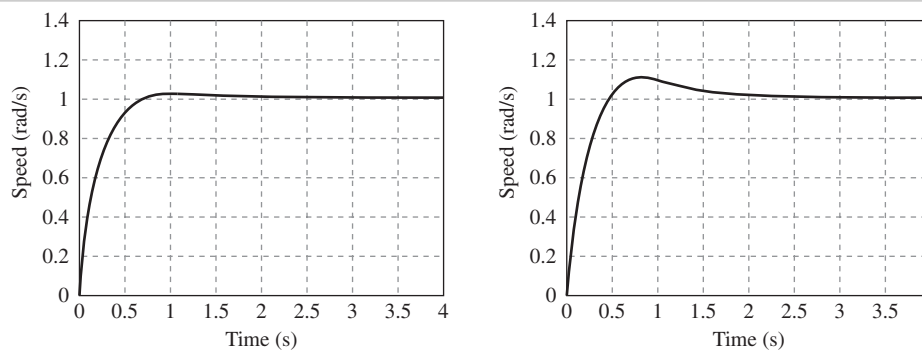
and then subtracting the final output from the unit input. As seen in Equation (9.10), the final steady-state error approaches zero only for very large gain K_p.

$$\lim_{t\to\infty} e(t) = 1 - \frac{K_p}{B + K_p} \neq 0 \tag{9.10}$$

To eliminate the steady-state error, we need to use a PI controller. The closed-loop transfer function for the overall system under closed-loop PI control is $(K_p s + K_i)/(Js^2 + Bs + K_p s + K_i)$ and is of second order. This means that response could exhibit oscillation depending on the value of the parameters of the system (see Appendix B). The speed response of the system under this case is shown in Figure 9.12 for two values of K_i, and shows no steady-state error.

Figure 9.12

Speed response under closed-loop PI-control



(a) $K_p = 3, K_i = 3$ (b) $K_p = 3, K_i = 6$

This also can be confirmed by determining the final steady-state error as shown in Equation (9.11).

$$\lim_{t\to\infty} \omega(t) = \lim_{s\to 0} s\omega(s) = \lim_{s\to 0} s\frac{K_p s + K_i}{Js^2 + Bs + K_p s + K_i}\frac{1}{s} = 1 \tag{9.11}$$

So

$$\lim_{t\to\infty} e(t) = 1 - 1 = 0$$

Example 9.2 illustrates the modeling of a PM DC motor under P-control, while Example 9.3 illustrates the selection of a PI controller gains for a speed control problem.

Example 9.2 P-Control of a PM DC Motor

Develop a block diagram model of a speed P-control system that uses a PM brush DC motor to drive the simple inertia system shown in Figure 9.8. An amplifier with a gain K_a volt/volt amplifies the voltage signal sent to the motor. The shaft speed is obtained from a tachometer with a sensitivity of k_{tach} volts/rpm.

Solution:

A block diagram of the components of this system is shown in Figure 9.13. The inductance of the PM brush DC motor is neglected here. The PM brush DC motor characteristics are given in Equation (8.5). The diagram has two loops: an inner loop to represent the PM DC motor characteristics; and an outer loop to represent the P-control action. The shaft speed measured by the tachometer is compared against the reference speed ω_{desired}, which should be in voltage units, to obtain the error voltage V_e. The factor $2\pi/60$ in the outer loop is a conversion factor from rad/s to rpm units. The error voltage is multiplied by the gain K_p (with units of volt/volt) to obtain the input voltage V_{in} that is sent to the amplifier. The amplifier output voltage V_m is sent to the motor. The back EMF voltage generated due to motor shaft rotation is subtracted from V_m and the net voltage is divided by the motor resistance R to get the armature current I_a. The armature current is multiplied by the motor torque constant K_T to get the torque T_m sent to the inertia system.

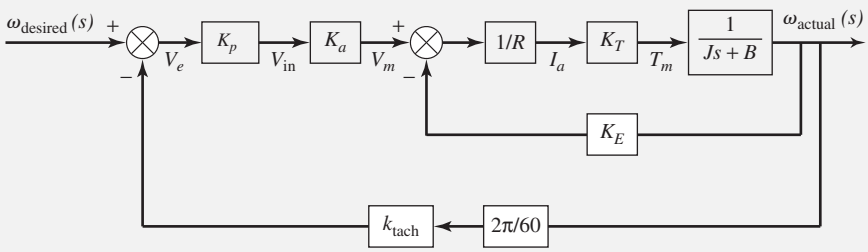


Figure 9.13

Note that the inner loop in the above diagram can be represented by the following transfer function:

$$\frac{\omega_{\text{actual}}(s)}{V_m(s)} = \frac{K_T}{RJs + RB + K_T K_E}$$

This transfer function has the same structure as that of the simple inertia transfer function given in Equation 9.7 but with different parameter values. Thus, one can see that using the model given by Equation 9.7 in the control analysis done in this chapter gives the same information as using a more detailed model such as the one above.

Example 9.3 Design of a PI Speed Controller

The model given by Equation (9.7) is placed under a PI speed controller. Determine the proportional and integral gains of the PI controller such that the controlled system has a critical damping ratio and a desired time constant of τ_d.

Solution:

The closed-loop transfer function of the model given by Equation (9.7) under a PI controller was used in Equation (9.11). The characteristic equation, which is second order, is

$$Js^2 + (K_p + B)s + K_i = 0 \tag{1}$$

The above characteristic equation can be equated to the characteristic equation of an underdamped second-order system that is given in Appendix B (see Equation (B.7)). This gives

$$s^2 + \frac{(K_p + B)}{J}s + \frac{K_i}{J} = s^2 + 2\zeta\omega_n s + \omega_n^2 \tag{2}$$

For critical damping, $\zeta = 1$. From Figure B.5, we know that the time constant and the natural frequency of a second-order system are related by

$$\tau = \frac{1}{\zeta \omega_n} \tag{3}$$

Using equations (2) and (3), the gains K_p and K_i are then obtained as

$$K_p = \frac{2J}{\tau_d} - B = B\left(2\frac{\tau}{\tau_d} - 1\right) \tag{4}$$

and

$$K_i = \frac{J}{\tau_d^2} \tag{5}$$

where $\tau = J/B$. For $J = 0.7$, $B = 0.5$, and $\tau_d = \tau/4$, K_p is 3.5 and K_i is 5.714. Figure 9.14 is the step response of the system using these gains.

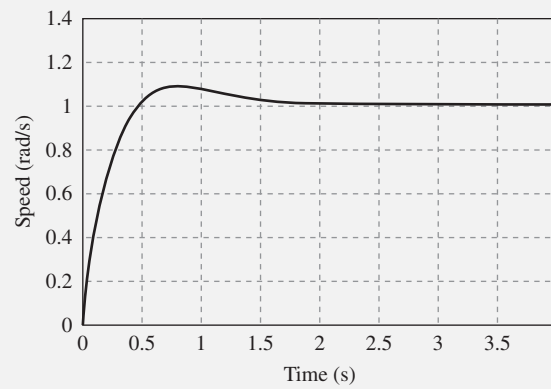


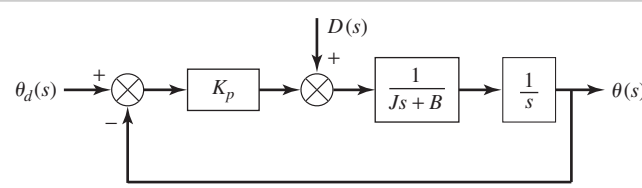
Figure 9.14

9.5.2 Position Control of an Inertia

Let us examine the response of the inertia system when the angular position of the inertia is controlled instead of the angular speed. A block diagram for the system under **P-control** is shown in Figure 9.15. The diagram shows an input disturbance that is also applied to the system. Note that since the angular position is the integral of the angular speed, the controlled plant is second order in this case.

Figure 9.15

Closed-loop P-control of inertia position



Using block diagram rules, we can determine an expression for the angular displacement $\theta(s)$ as a function of both the desired displacement $\theta_d(s)$ and an input disturbance $D(s)$. For a step reference input $\theta_d(s)$ and a constant disturbance $D(s)$, the output is given by

$$\theta(s) = \frac{K_p}{Js^2 + Bs + K_p}\theta_d(s) + \frac{1}{Js^2 + Bs + K_p}D(s) \tag{9.12}$$

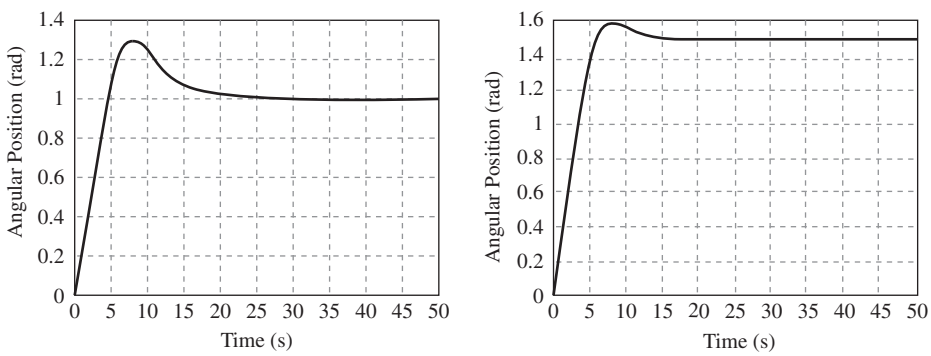
So the steady-state output is

$$\lim_{t\to\infty} \theta(t) = \lim_{s\to 0} s\theta(s) = \theta_d + \frac{1}{K_p}D \tag{9.13}$$

and the steady-state error is

$$\lim_{t\to\infty} e(t) = \theta_d - \theta(t) = -\frac{1}{K_p}D \tag{9.14}$$

Since disturbances are always present in a real system, there will always be a steady error if only P-action control was used. Similar behavior would be obtained if **PD control** was used. Figure 9.16 shows the position response under closed-loop P-control without and with constant disturbance ($K_p = 0.2$ and $D = 0.1$). Notice the large steady-state error when the disturbance is present.



(a) No Disturbance

(b) Constant Disturbance ($D = 0.1$)

Figure 9.16

Position response under closed-loop P-control without and with constant disturbance ($K_p = 0.2$)

Now let us examine the response of the system under **PI control**. The block diagram for this case is shown in Figure 9.17.

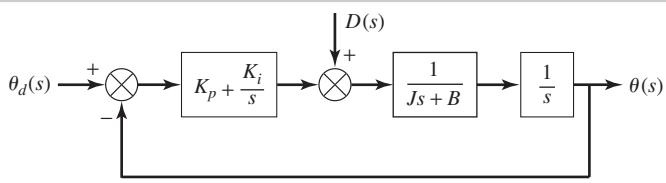


Figure 9.17

Closed-loop PI-control of inertia position

The transfer function between the actual displacement and the desired displacement is given by

$$\frac{\theta(s)}{\theta_d(s)} = \frac{K_p s + K_i}{Js^3 + Bs^2 + K_p s + K_i} \tag{9.15}$$

and the transfer function between the actual output and the disturbance is given by

$$\frac{\theta(s)}{D(s)} = \frac{s}{Js^3 + Bs^2 + K_p s + K_i} \tag{9.16}$$

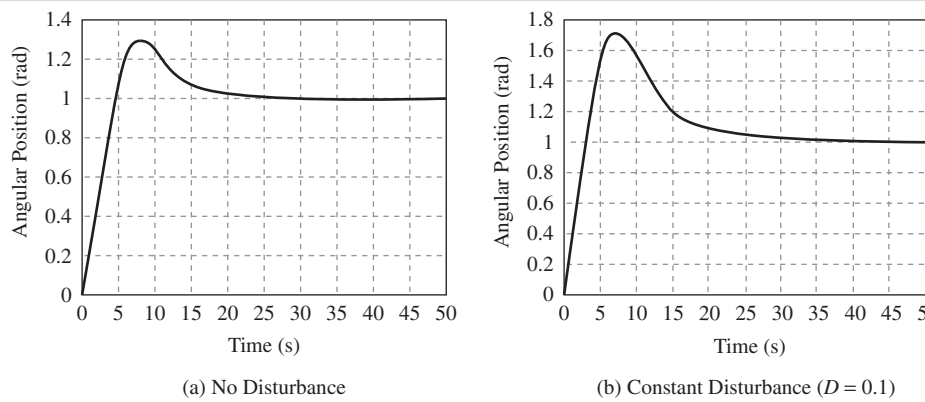
Similar to the P-action case, we can determine the final steady-state output for the case of step reference input and a constant disturbance. The final steady-state output is θ_d as seen in Equation (9.17).

$$\lim_{t\to\infty} \theta(t) = \lim_{s\to 0} s\theta(s) = \theta_d + 0 = \theta_d \tag{9.17}$$

So, unlike P-action control, the addition of the I-action produces a response with zero steady-state error even with the presence of disturbances. Thus, for position control, PI or PID control should give a response with no steady-state error. Figure 9.18 shows the position response under closed-loop PI-Control ($K_p = 0.2$, $K_i = 0.02$) without and with input disturbance present.

Figure 9.18

Position response under closed-loop PI control without and with disturbance



(a) No Disturbance

(b) Constant Disturbance ($D = 0.1$)

One important issue in the design of a feedback controller is the **stability** of the controller system. Ruth and Hurwitz [38] have come up with a quick method to determine the stability of a control system based on the values of the coefficients of the closed-loop characteristic equation of the system. Table 9.1 lists these stability conditions for first-, second-, and third-order systems.

Table 9.1

Ruth-Hurwitz stability conditions

System Order	Characteristic Equation	Stability Conditions
First	$a_1 s + a_0 = 0$	Stable if and only if a_1 and a_0 have the same sign
Second	$a_2 s^2 + a_1 s + a_0 = 0$	Stable if and only if a_2, a_1, and a_0 all have the same sign
Third	$a_3 s^3 + a_2 s^2 + a_1 s + a_0 = 0$	For $a_3 > 0$, stable if and only if a_2, a_1, and a_0 all have the same sign and $a_2\, a_1 > a_3\, a_0$

As an example, let us apply these stability conditions for the characteristic equation of the transfer function of Equation (9.15). Since the inertia and viscous damping coefficients are always positive for this third-order system, the system is stable if both K_p and K_i are positive and if $K_p/K_i > J/B$. This is the case for the parameters used to obtain the plots of Figure 9.18.

9.6 Digital Implementation of a PID Controller

Due to the discrete nature of digital control, when a PID controller is implemented on a PC or a microcontroller, the controller is approximated by

$$y(kT) = K_p e(kT) + K_i T \sum_{j=0}^{k-1} e(jT) + K_d((e(kT) - e((k-1)T))/T \quad (9.18)$$

or alternatively

$$u_i(kT) = u_i((k-1)T) + K_i T e((k-1)T)$$
$$y(kT) = K_p e(kT) + u_i(kT) + K_d((e(kT) - e((k-1)T))/T \quad (9.19)$$

where T is the sampling interval, u_i is the I-action control output, and k ($k = 0, 1, 2, \ldots$) is an index that represents the number of the instance at which control is done. Notice how the integral for the I-action term is now replaced by a summation and the derivative for the D-action is now replaced by a difference equation. This summation expression comes from approximating the area under the error vs. time plot. While there are several ways to do the approximation (such as backward approximation, and trapezoidal), Equation (9.18) is based on the **forward rectangular approximation** scheme. This is illustrated in Figure 9.19, for example, where at $k = 3$ the sum of all the errors to that point is given by the sum of the three shaded areas.

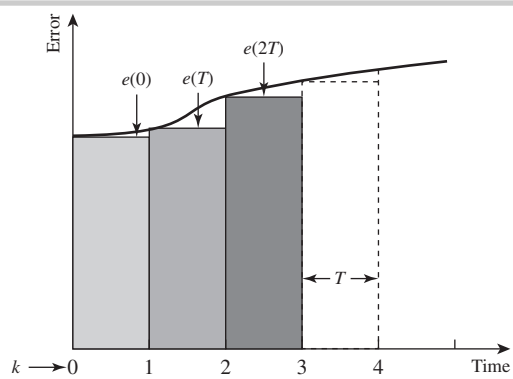


Figure 9.19
Forward rectangular approximation

In Section 6.4.2, we had discussed a control software structure for implementation of a feedback control system. The next chapter discusses the details of implementation of this control software structure where a digital PI controller was implemented in two different platforms, one using an MCU and the other using a PC.

9.7 Nonlinearities

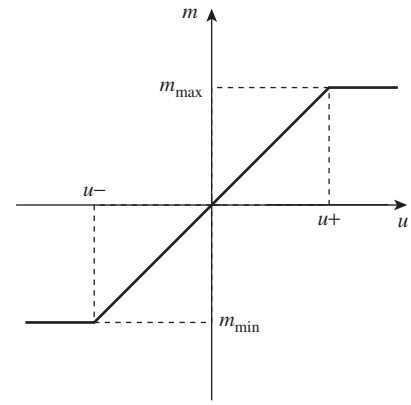
Real systems have nonlinearities such as saturation and Coulomb friction. These nonlinearities cause a deviation from the ideal linear system behavior discussed before. This section will discuss the effect of these nonlinearities.

9.7.1 Saturation

In real systems, the output from the controller is limited. This is the case since the amplifiers that amplify the control output that is sent from the computer or the

Figure 9.20

Illustration of the saturation nonlinearity



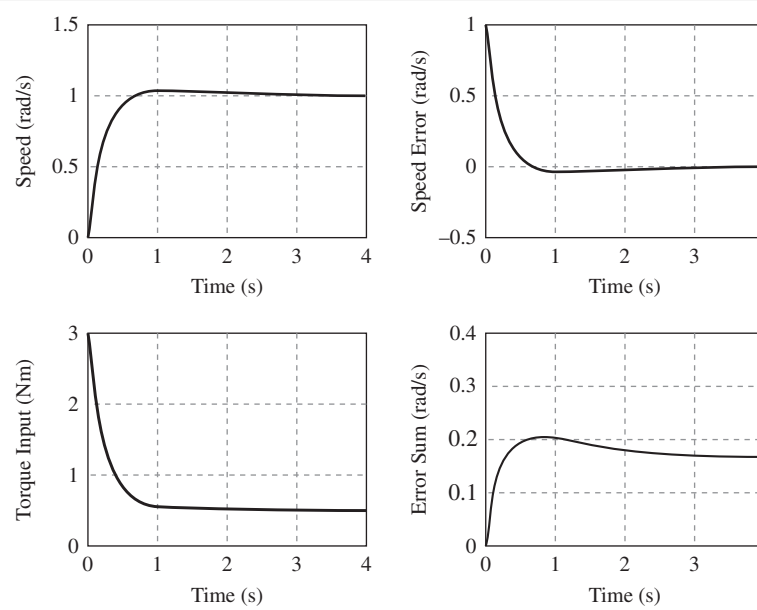
microcontroller to the actuators are power limited. The inability to exceed a particular output level is called saturation, and a plot of the **saturation nonlinearity** is shown in Figure 9.20, where u is the controller output and m is the amplifier output. When the controller output exceeds $u+$ (*or is smaller than* $u-$), the output saturates at the maximum (minimum) value of m_{max} (or m_{min}). In some systems such as a heating system, m_{min} is zero, since a heater can not apply a negative heat.

The saturation of the controller output causes the PID controller to overshoot and to delay the response of the system. To illustrate this point, consider Figure 9.21 which shows a PI simulation ($K_p = 3$ and $K_i = 3$) of the simple inertia system considered earlier with no saturation limits placed on the controller output. Notice how the speed overshoot is very limited, and the speed error goes to zero just before $t = 3.4$ s. However, the controller output is initially high and stays above 1.0 N · m for approximately the first 0.4 s. The Simulink model (see Appendix C) that is used to obtain this plot is shown in Figure 9.22. The model uses the continuous time *PID Controller* block in Simulink.

Now the PI simulation was repeated but with a limit of +/− 1 N · m placed on the controller output. This simulation is obtained by specifying the *lower saturation*

Figure 9.21

PI simulation with no controller output limits



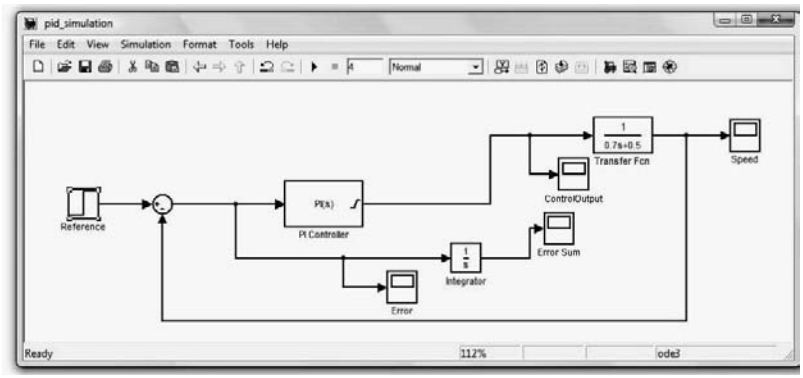


Figure 9.22

Simulink model for simulating a PI controller of a first-order system

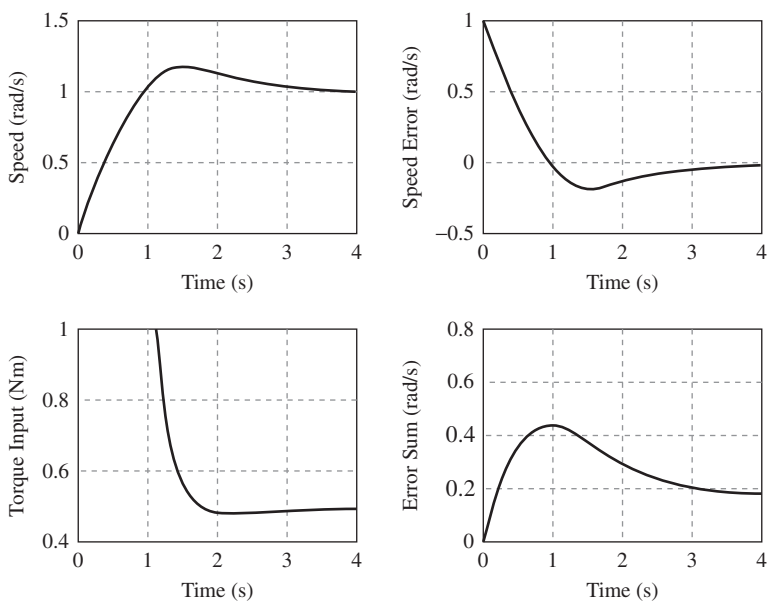


Figure 9.23

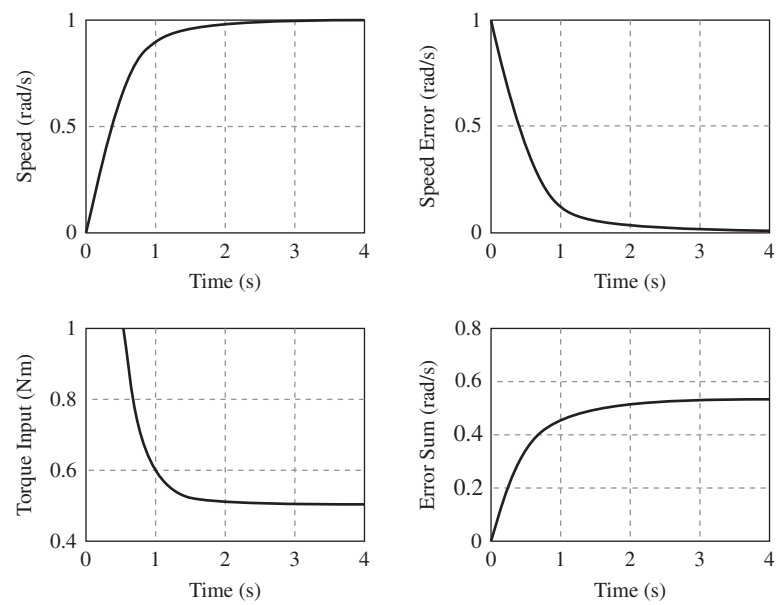
PI simulation with a limit of +/− 1 N · m placed on the controller output

limit and the *upper saturation limit* in the *PID advanced* section of the block parameters for the *PID controller* block. The simulation results are shown in Figure 9.23. The controller saturation is shown in the plot of the torque input to the system where the torque stays at the limit value for approximately the first 1.1 s. Notice now the significant overshoot in the speed response plot. This overshoot is a direct result of the I-action and is explained below. Due to P-action alone, the controller output is close to 3 N · m at the beginning of the simulation, assuming zero initial conditions. Having I-action in this time interval is not useful, since the additional control output from the I-action will not be utilized due to saturation of the control output. Furthermore, due to the summing nature of the I-action, the I-action term keeps increasing in value until the error switches sign and will supply a nonzero input to the system even if the error is zero. Thus, for systems that have saturation, it would have been better to shut off completely the I-action while the contribution from the P-action alone (or from the P and D-action if a PID controller is implemented) exceeds the controller limit. This behavior of the PI controller is called **reset windup** or **integrator buildup** and can occur with any controller with integrator action and saturation.

With the use of digital controllers, it is possible to implement the PID controller in software so as to avoid the above-mentioned problem. Such implementation is called a PID controller with anti-windup or **no reset windup**. The turning off or adjustment of the I-action contribution helps to prevent the reset windup of the system. In Simulink, no reset windup simulation is obtained by selecting an anti-windup method in the *PID advanced* section of the block parameters for the *PID control* block. Figure 9.24 shows the simulation results with the *clamping* method is chosen as the anti-windup method. In this method, Simulink stops integration action when the sum of the PID block components exceeds the output limits and the integrator output and block input have the same sign. The I-action integration is resumed when the sum of the block components exceeds the output limits and the integrator output and PID block input have opposite sign. Here the speed response is slower than the response shown in Figure 9.23, but there is no overshoot.

Figure 9.24

Simulation of PI controller with no reset windup feature



9.7.2 Nonlinear Friction

Mechanical systems subjected to dry or Coulomb friction exhibit a nonlinear response behavior. This point is illustrated in Figure 9.25, which shows the open-loop speed step response of a small PM brush DC motor subjected to different input voltages. At a 3.0 V input, the final speed steady-state value is less than 1 V. At double the input voltage value or 6 V, the final steady-state value is about 3 V, which is more than three times the response at 3.0 V input. This behavior is characteristic of **dry or Coulomb friction** where the output speed is not proportional to the input voltage. In addition, the input voltage (or torque) has to exceed a certain value before motion occurs. For example, in this particular system, the motor does not rotate unless the input voltage exceeds 2.25 V.

A linear model for such a system (such as Equation 9.7) will only reproduce the actual behavior at input conditions similar to those used in obtaining the model parameters. At other input values, the model prediction would deviate from real behavior. This point is illustrated in the next chapter in modeling the paper dispensing system (see Section 10.3.6). Fortunately in many cases, with closed-loop PI or PID control action, the nonlinear behavior becomes a disturbance to the

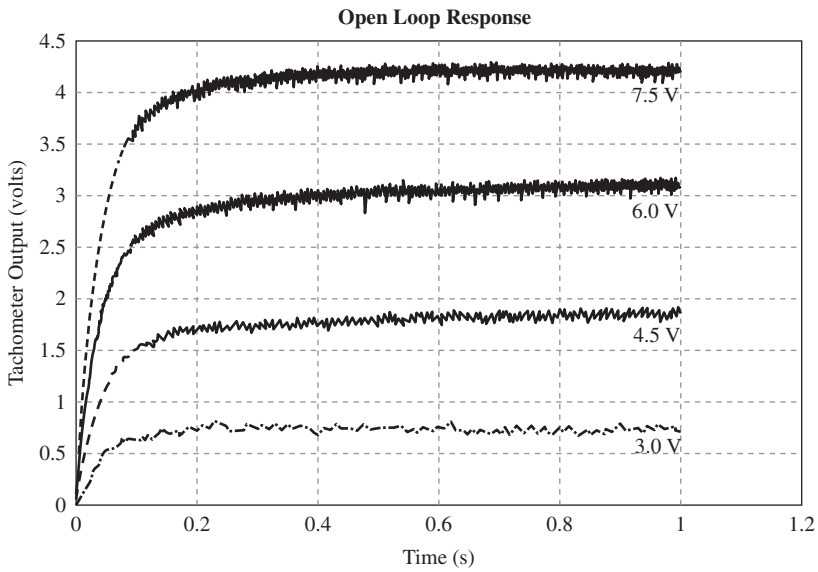


Figure 9.25

Open-loop step response of a small DC motor system with a tachometer

system, and the feedback control system can produce a response with zero steady-state error for such a system.

9.8 Other Control Schemes

In addition to the PID controller, there are many other control schemes that are used in practice. Some of these include the ON-OFF controller, and the state feedback controller. Each of these control schemes will be briefly discussed in this section.

9.8.1 On-Off Controller

The on-off controller is a simple, but nonlinear controller in which the controller is either on or off. A heating/cooling system controlled by a thermostat is an example of an on-off controller. In Chapter 6, we illustrated the simulation of such a controller in discussing the implementation of a digital thermostat state-transition diagram in software using different computing platforms. An on-off controller can be easily implemented in a PC or MCU using a combination of a digital output line and a relay or a transistor to turn on/off the control voltage to the actuator or heater.

Because the on-off controller is nonlinear (the output is not proportional to the error but is fixed at either one of two values), the closed loop transfer function of the overall system cannot be simply obtained as was done for the PID controller. However, we can simulate the behavior of the controller using MATLAB. Figure 9.26 shows a simulation of a heated plate in which the output is either 0 or 12 V (the model of this system is discussed in Section 10.4.5).

The on-off controller was implemented as a Simulink MATLAB function block with a sample time of 1 second. The left part of Figure 9.26 shows the temperature response of the plate for a 10° step change in temperature as well as the output from the on-off controller. As seen in the plot, when the plate temperature was below the desired temperature, the controller was fully on. After the plate reaches the desired temperature at $t = 146$ s, the on-off controller alternates between being fully on and off. Since the controller cannot supply any cooling, the off interval is longer than the on interval, as seen in Figure 9.26(b).

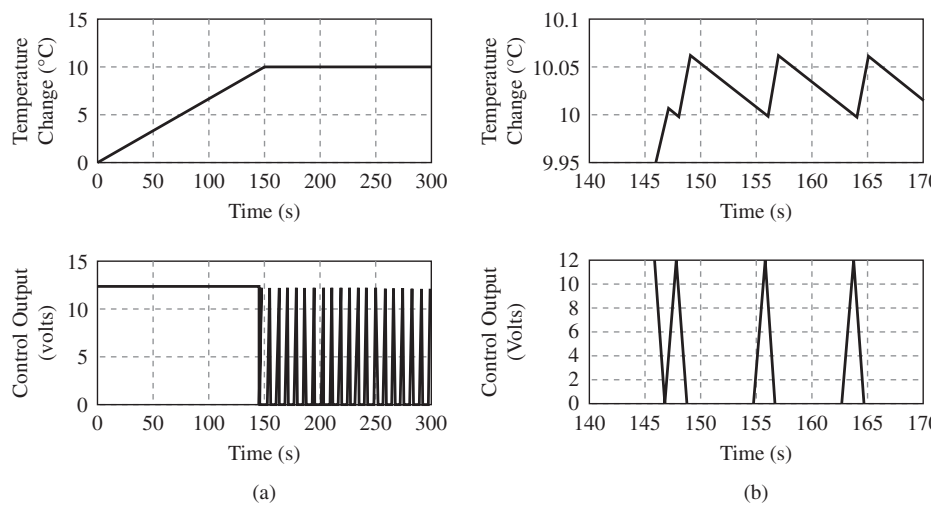


Figure 9.26

Simulink simulation of an on-off controller for the heater system considered in Section 10.4. (a) Response and controller output and (b) detailed view of response and controller output

9.8.2 State Feedback Controller

One limitation of the PID controller is that the location of the closed-loop poles of the system cannot be arbitrarily selected but are a function of the control gains used. If we want to exactly specify the location of the closed-loop control poles (and hence the dynamic response behavior), then we need to use a state feedback controller.

For a linear system represented in the form (see Appendix C)

$$\dot{x} = Ax + Bu \tag{9.20}$$

the state feedback controller takes the form

$$u = -Kx \tag{9.21}$$

where K is the $1 \times n$ state feedback gains matrix. Substituting the controller expression into Equation (9.20), a system under state feedback control has a dynamics given by

$$\dot{x} = (A - BK)x \tag{9.22}$$

The gain matrix K is determined by matching the poles of the system matrix A-BK to a user specified set of pole locations. In MATLAB, this operation is performed using the *PLACE* function. For arbitrary placement of the closed-loop poles, the given system needs to be controllable [40].

The controller form given by Equation (9.21) performs a regulation of the system around the origin (i.e., if the system is subjected to disturbances, it will bring the state vector back to the origin). If we need the state feedback controller to **track a particular reference signal *r***, then the state feedback controller is given in the form

$$u = -Kx + \tilde{N}r \tag{9.23}$$

where $\tilde{N}$ is a $1 \times m$ gain matrix to produce zero steady-state error for a reference r. The determination of the $\tilde{N}$ matrix is discsussed in many control texts, see for example [40]. For a single input, single output system, $\tilde{N}$ is given by

$$\tilde{N} = -1/(C(A - BK)^{-1}B) \tag{9.24}$$

Example 9.4 illustrates the design of a state feedback controller for a non-rigid gear drive system.

Example 9.4 State Feedback Controller for Non-Rigid Gear Drive System

Design and simulate in MATLAB a state feedback controller to control the load link displacement in the gear drive system considered in Example 8.2 and reproduced in Figure 9.27. Let the torque output of the motor be the input to the system, and take into consideration the compliance of the input and output shafts. Use the following parameter values: $N = 10$, $k_1 = 1200$ N/m, $k_2 = 1900$ N/m, $b_1 = 0.015$ Nm s/rad, $b_2 = 0.030$ Nm s/rad, $I_1 = 0.002$ kg m^2, and $I_2 = 0.925$ kg m^2.

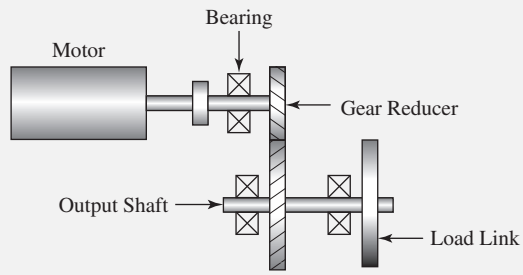
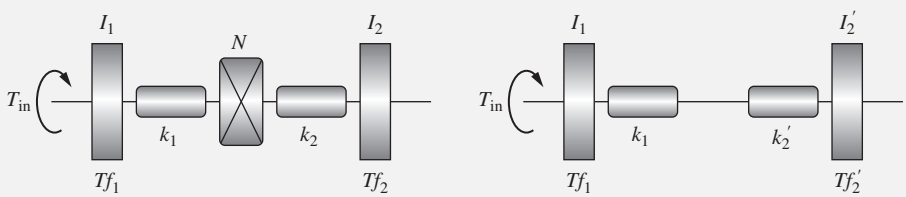


Figure 9.27

Solution:

If the shaft compliance is taken into account, then the motor inertia and the load inertia are connected through compliant members and our model needs to reflect this fact. The system is modeled as shown in Figure 9.28(a). The rotor is connected in series with a spring k_1 that represents the elasticity of the shaft connecting the rotor to the gear. Similarly, spring k_2 represents the elasticity of the shaft connecting the gear to the link. We assume the gears to be rigid since the gear stiffness is normally much higher than the shaft stiffness. The model in Figure 9.28(a) can be represented by an equivalent system based on the input shaft as shown in Figure 9.28(b). In this representation, the gear reduction is eliminated and the parameters that represent the output shaft stiffness k_2, the output inertia I_2, and the friction torque Tf_2 are modified to reflect the effect of the gear reduction (i.e., $k_2' = k_2/N^2$, $I_2' = I_2/N^2$, and $Tf_2' = Tf_2/N$). The springs k_1 and k_2' are in series and can be combined to have an effective stiffness of k. Let θ_1 be the motor angular position, and let θ_2 be the angular position of the output link measured in the input shaft coordinate system



(a) Model with Shaft Compliance (b) Model Incorporating Gear Effects

Figure 9.28

The equations of motion for the system are then obtained as

$$T_{in} = k(\theta_1 - \theta_2) + b_1\dot{\theta}_1 + I_1\ddot{\theta}_1 \quad (1)$$

$$k(\theta_1 - \theta_2) - b_2'\dot{\theta}_2 = I_2'\ddot{\theta}_2 \quad (2)$$

where $1/k = 1/k_1 + 1/k'_2$, $k'_2 = k_2/N^2$, $I'_2 = I_2/N^2$, and $b'_2 = b_2/N^2$ where N is the gear ratio.

If we let x_1 to be the input shaft (or motor) angular displacement, x_2 to be the input shaft angular speed, x_3 to be output link angular displacement measured in the input shaft coordinate system, and x_4 be the output link angular speed measured in the input shaft coordinate system, then the above equations can be represented in state space form with:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -k/I_1 & -b_1/I_1 & k/I_1 & 0 \\ 0 & 0 & 0 & 1 \\ k/I'_2 & 0 & -k/I'_2 & -b_2/I'_2 \end{bmatrix},\ B = \begin{bmatrix} 0 \\ 1/I_1 \\ 0 \\ 0 \end{bmatrix},\ C = [0\ 0\ 1\ 0], \text{ and } D = [0] \quad (3)$$

For the given parameters, the A and B matrices are given in MATLAB as

$$A = \begin{matrix} \text{1.0e + 003*} \\ \begin{bmatrix} 0 & 0.0010 & 0 & 0 \\ -9.3519 & -0.0075 & 9.3519 & 0 \\ 0 & 0 & 0 & 0.0010 \\ 2.0220 & 0 & -2.0220 & -0.0000 \end{bmatrix} \end{matrix} \quad \text{and} \quad B = \begin{bmatrix} 0 \\ 500 \\ 0 \\ 0 \end{bmatrix}$$

If we let the desired closed-loop poles of the system be clp = [−3 −4 −5 −6], then the state feedback gain matrix K is determined from the MATLAB command:

```
K = place(A, B, clp)
```

This gives the K vector as:

K = [−22.5111 0.0209 22.5115 −0.0355]

From equation (9.24), the $\tilde{N}$ parameter is 3.5608e-004. The state feedback simulation of the above system model for a unit step response of x_3 is shown in Figure 9.29. The model is simulated as the following system

```
sys1 = ss(A-B * K, B, C, D);
```

with the elements of the input vector multiplied by the parameter $\tilde{N}$. The plot also shows the response for two other sets of closed-loop pole locations. The state feedback controller achieved a zero steady-state error and as expected, the response speed improves as the poles are placed farther away from the imaginary axis.

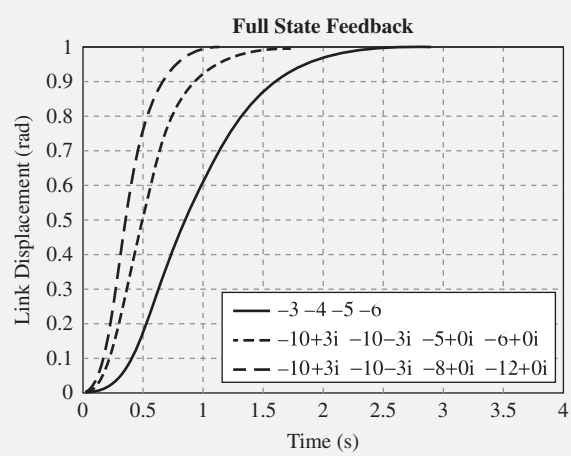


Figure 9.29

Note that a state feedback controller requires availability of all the states of the system. In reality, all states may not be available. For example in many positioning systems, only the position signal is available, and the velocity signal is not measured. The unavailable states can be created in software using an observer [40], but this topic is beyond the scope of this textbook. State feedback control can also produce large actuator output if the poles are not selected carefully.

Even if the position and velocity signals are available, for many positioning applications, such as the system considered in Example 9.4, we have the choice of placing the position/velocity sensors on the motor shaft or on the output link shaft but not on both shafts to reduce the cost and complexity of the feedback system. To illustrate these choices, consider the system of Example 9.4 with a state feedback controller of the form

$$T_{\text{in}} = R\,K_p - K_p\,x_1 - K_d\,x_2 \tag{9.25}$$

where x_1 is the motor shaft angular displacement and x_2 is the motor shaft angular speed. The unit step response of the link position for different combinations of K_p and K_d gains is shown in Figure 9.30(a). The figure shows that the motor shaft–based state feedback controller behaves adequately, and the response behavior is dependent on the K_d gain. Let us now replace the controller in Equation (9.25), by one that is based on the output link shaft. The state feedback controller in this case is

$$T_{\text{in}} = R\,K_p - K_p\,x_3 - K_d\,x_4 \tag{9.26}$$

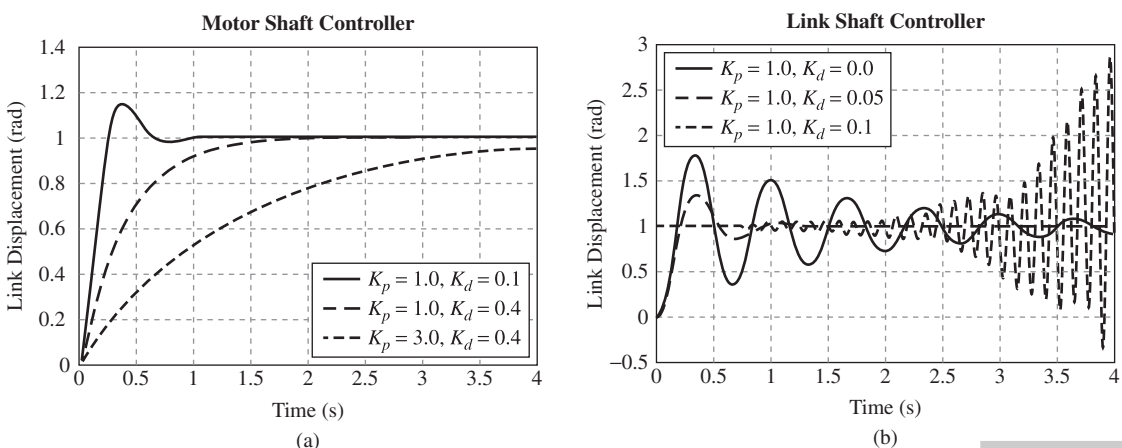


Figure 9.30

Response of the system in Example 9.4 to (a) state feedback controller based on the input (motor) shaft and (b) a state feedback controller based on the output (link) shaft

where x_3 is the output link angular displacement measured in the input shaft coordinate system and x_4 is the output link angular velocity measured in the input shaft coordinate system. The response of the system under such a controller is shown in Figure 9.30(b). Notice here how the system becomes unstable with increasing values of K_d. The second control configuration is an example of a **non-collocated** actuator-sensor system [41]. The instability arises from the compliance of the drive elements between the actuator and the sensor. Thus to avoid instability problems, the controller should be based on the input shaft at the expense of less accurate positioning of the link since a motor shaft controller does not compensate for any hysteresis or compliance in the gear train.

9.9 Chapter Summary

This chapter gave a brief overview of feedback control systems. A feedback control system is one in which the input to the system is a function of *both* the actual output and the reference input, unlike an open-loop system in which the input is only a function of the reference input. The chapter focused primarily on the PID control algorithm, which is one of the most widely used feedback control laws. Through analysis, it was shown that a P or PD controller would not achieve a zero steady-state error in controlling a first-order system under a step input. However, a PI or PID controller would achieve a zero-steady-state for the same system with or without constant disturbances acting on the system. For a second-order system, a P-controller would achieve a zero steady-state error for step input provided that there are no disturbances, while the PI or PID controller would achieve no steady-state error with or without the presence of constant disturbances. The effect of saturation nonlinearity on the behavior of feedback control system with integrator action was also considered. A simulation of the behavior of a PID controller with no reset windup was presented in this chapter. Other control schemes such as the on-off controller and the state feedback controller were also discussed in this chapter.

Questions

9.1 List two limitations of open-loop control.

9.2 What assumption is made in obtaining a transfer function?

9.3 What do the poles and zeros of a transfer function mean?

9.4 Does a P-action closed-loop control of a first-order system achieve a zero steady-state error?

9.5 What condition assures stability of a closed-loop control system?

9.6 What causes reset-windup problems when using a PID controller?

9.7 List two nonlinear effects that are encountered in control of real systems.

9.8 What limits the application of the state feedback controller in some cases?

Problems

P9.1 The controller transfer function and the plant transfer function are given below. Determine the overall closed transfer function of this system. What is the final steady-state value for a unit step input applied to the closed-loop system?

$$G_c(s) = \frac{s+3}{s+6}$$

$$G_p(s) = \frac{5}{s^2+s+5}$$

P9.2 Determine the closed-loop transfer function of the system shown in Figure P9.2.

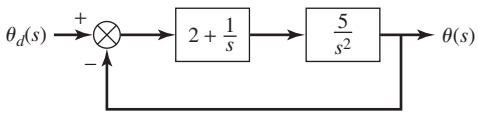


Figure P9.2

P9.3 For the system shown in Figure P9.2, determine the transfer function between $\theta(s)$ and a disturbance $D(s)$ applied to the system between the controller and the plant blocks. What is the final steady error if the disturbance is constant and has a magnitude of 2?

P9.4 For the block diagram shown in Figure 9.6, determine the transfer function between $\theta(s)$ and a disturbance $D(s)$ applied to the diagram between the K_v and the $K/\tau s+1$ blocks. What is the final steady output if the disturbance is constant?

P9.5 If the P-controller in Figure 9.15 was replaced with a PD-controller with a transfer function of $K_p + K_d s$, determine the closed-loop transfer function between the angular displacement $\theta(s)$ and the desired displacement $\theta_d(s)$ as well as

between the angular displacement $\theta(s)$ and the input disturbance $D(s)$.

P9.6 A first-order system with the transfer function $G(s) = 1/(0.5\ s + 1)$ was placed under a closed-loop P-control. Determine the value of the control gain K_p so that the closed-loop control system time constant is 0.1 seconds. What is the value of the steady-state error for a unit-step input using this gain?

P9.7 The system in Problem 9.6 was placed under a closed-loop PI control. Determine if the system will have an overshoot for a step input:

a. $K_p = 2$ and $K_i = 1$

b. $K_p = 1$ and $K_i = 3$

P9.8 For the system given in Problem 9.7, determine the K_p and K_i gains so that the closed-loop system has a natural frequency of 5 rad/s and a damping ratio of 1.

P9.9 For the inertia system represented by Equation 9.6 and using the velocity and position as state variables, determine:

a. State space model matrices (A, B, C, and D) for the system assuming that the position (x_1) is used as the output.

b. The state feedback gains for desired closed-loop poles of $-5+3$i and $-5\ -3$i. Use the same B and J values as those used in Figure 9.9.

Laboratory/Programming Exercises

L/P9.1 Using the following transfer function $G(s) = 1/(0.5\ s + 0.1)$, model a PI controller for this system in Simulink and perform the following:

a. Obtain the unit step response for $K_p = 1.0$ and $K_i = 0$.

b. Repeat the step response but use $K_p = 3$ and $K_i = 0.5$.

c. Repeat the step response but use $K_p = 1$ and $K_i = 1$.

d. Repeat the step responses in parts a through c, but consider the effect of saturation on system response by adding a saturation block after the controller block with output limits of $+/-$ 0.5.

L/P9.2 Design and implement in MATLAB a state feedback controller for controlling the position of the system given by Equation (9.6). Use the model parameters given in Example 9.3. Try at least two different locations for the closed-loop system poles. Plot the response of the system for each of these cases.

L/P9.3 Implement a closed-loop PI controller in a PIC-microcontroller for a first-order system such as a motor-tachometer system. Use the A/D converter on the board to read the output of the system and the PWM feature to send the control output to the system. Connect the microcontroller to the PC to display to the user information about the performance of the system.

L/P9.4 Use Simulink to model the Coulomb friction nonlinearity. Use the data shown in Figure 9.25 to obtain the parameters of the model. Test your model with different inputs and compare the model results to the actual data in Figure 9.25.

L/P9.5 Implement in VBE the logic for a PID controller with no reset windup. Test your code by simulating the speed control of an inertia similar to the one done in this chapter.

CHAPTER 10

Mechatronics Projects

CHAPTER OBJECTIVES:

When you have finished this chapter, you should be able to:

- Apply state-transition diagrams concepts to the operation and control of different mechatronic systems
- Apply circuit design for the construction of circuits to interface PIC microcontrollers with physical systems
- Develop software for control of mechatronic systems
- Develop software for the interface between a PC and a microcontroller system
- Apply modeling techniques to develop a dynamic model of a mechatronic system
- Apply MATLAB to simulate the response of mechatronic systems
- Explain the integration of the different components of a mechatronic system such as sensors, actuators, amplifiers, interface circuits, and control software that were covered in the book

10.1 Introduction

This chapter discusses several experimental systems that are suitable for extended or final project topics. These systems include a) a stepper-motor driven rotary table, b) a toilet-paper dispensing system that uses a roller driven by a position-controlled DC motor, and c) a temperature-controlled heating apparatus that uses a heating coil, a copper plate, and a temperature sensor. All three systems combine various mechatronics elements. The discussed projects objective is to link the topics covered in the previous chapters into an integrated unit. All of the suggested projects include software design issues, hardware interfacing, data-acquisition, timing, and control software. The second and third projects also include dynamic modeling. A list of the main components needed to fabricate each of these systems is provided.

10.2 Stepper-Motor Driven Rotary Table

Stepper motors are widely available and offer a low-cost actuation system that can operate in open-loop fashion without the need for feedback sensors. Stepper motors require digital signals for actuation, and these can be conveniently supplied from a PC or microcontroller. This project uses a PIC MCU as the controller.

10.2.1 Project Objectives

This project focuses on the open-loop control of a stepper-motor–driven rotary table. The project objectives are to illustrate:

- State-transition diagram for a discrete-event control system
- Circuit for interfacing the microcontroller (PIC16F690) and the stepper motor system
- The use of PWM signals for driving the stepper motor
- The development of a C program for the control code

10.2.2 Setup Description

The setup consists of a four-phase stepper motor with its rotation axis oriented vertically and mounted on a small aluminum support base, as shown in Figure 10.1(a). An optical CD with a notch is mounted on the stepper-motor shaft, as shown in Figure 10.1(b). The notch presence on the disk is detected by a photo interrupter optical sensor. A PIC16F690 microcontroller is used as the controller, and it transmits the step and pulse information to either a stepper-motor driver chip or a commercial stepper-motor driver. The PIC16F690 microcontroller is mounted on a Microchip low pin-count development board (see Figure 4.11).

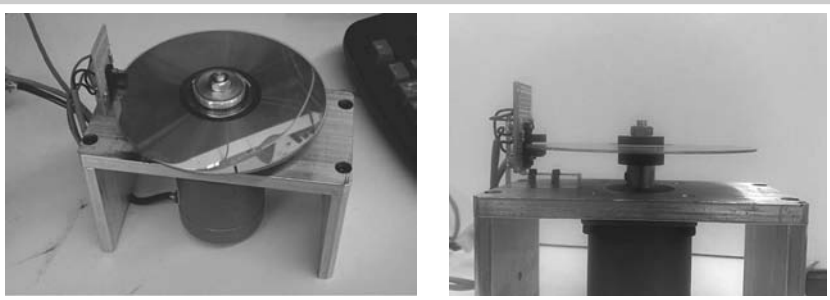


Figure 10.1

(a) Rotary table system and (b) mounting details of the CD

(Jouaneh, University of Rhode Island)

10.2.3 Interface Circuit

The PIC16F690 can be interfaced to the stepper motor using either a stepper-motor interface chip (such as the EDE1200 in Figure 8.36) or a commercial stepper-motor driver. Figure 10.2 shows an interface circuit using the **PDO2035**

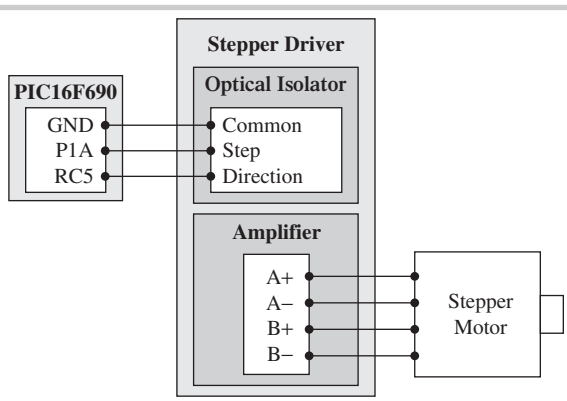


Figure 10.2

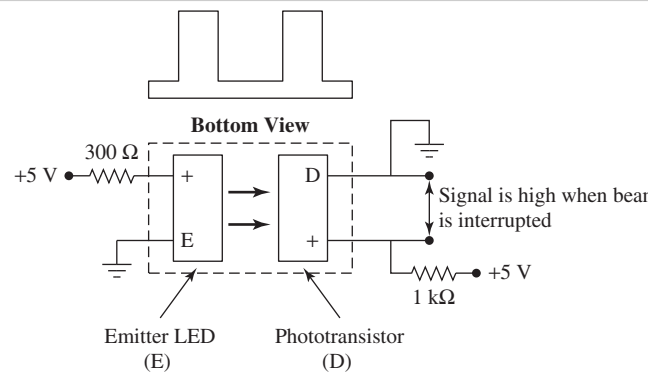
Interface circuit of PIC16F690 and stepper-motor driver

stepper-motor driver from Applied Motion Products, which was discussed in Section 8.4.2. The stepper motor used in this project is a four-phase, six-lead motor and was connected using a series connection (see Figure 8.39(a)). The MCU sends pulse and step signals to the driver, and the driver in turn sends the appropriate signals to the phases of the stepper motor. The direction signal is supplied from the RC5 port on the PIC16F690, while the pulses are supplied from the P1A PWM port. Full- and half-stepping modes as well as the current setting per phase are set using dip switches on the driver.

The circuit for processing the signal from the **photo interrupter** is shown in Figure 10.3. As discussed in Section 4.4, a photo interrupter is a combination of an LED and a phototransistor. When the light beam sent by the diode is interrupted by an object, the phototransistor stops conducting, and the output of the sensor is pulled up to the supply voltage.

Figure 10.3

Interface circuit for photo interrupter sensor



10.2.4 Operation Commands

The motion of the table should be controlled by three commands. An explanation of the desired function of each of these commands is given here.

CW: This command should cause the table to rotate clockwise (as seen from above the disk) and should cause the table to automatically stop when it reaches the optical sensor. If the table was originally at the home position, this command should cause the table to make one complete revolution.

CCW: This command should cause the table to rotate counterclockwise (as seen from above the disk) and should cause the table to automatically stop when it reaches the optical sensor. If the table was originally at the home position, this command should cause the table to make one complete revolution.

Stop: This command should stop the table if it was moving, but the program should still be running.

At startup, the program should rotate the disk until the notch on the disk is aligned with the optical sensor (home position). If the disk is already aligned with the sensor, then there is no need to home the table.

Since the low pin-count development board has only one built-in switch (SW1) input, we can implement the three commands using the rotary pot (see Section 7.3.1) on the development board. The rotary pot is connected to a 10-bit analog channel RA0. We map the output of the analog port as shown in Figure 10.4. The nearly three-quarter revolution of the rotary pot is split into four

Stop 0–255	CW 256–511	CCW 512–767	Stop 768–1023

Figure 10.4

Mapping interface between commands and A/D output

zones: *Stop*, *CW*, *CCW*, and *Stop*. The use of two separate *Stop* zones allows the user to easily change between the different commands. To replicate button input in a GUI, the *CW*, *CCW*, and *Stop* commands are only active when the built-in SW1 switch on the board is pressed. This prevents the command from being active all the time.

The desired operating commands can be translated into seven states as shown in the **state-transition diagram** in Figure 10.5. At startup, the program starts in the *Start* state. If the disk happens to be at the home position, then the state transitions to the *AtHome* state. If the disk is not at the home position, then the disk is set to rotate clockwise until it hits the home position. In the *AtHome* state, the table can transition to either the *Initial CW Motion* or *Initial CCW Motion* state, depending on the user input. These initial motion states are used for a short time while the disk clears the home sensor zone. No monitoring of the home sensor is done in either state to prevent a transition based on false triggering. When the short time interval elapses, the table is in the *CW Motion* or *CCW Motion* state. Monitoring is done in each of these states for a home sensor signal, a *Stop* command, or a reversal of the motion direction command, and the corresponding transition is taken. In the *Stopped* state, the table waits for a *CW* or *CCW* command to start moving again. While the table is stopped in both the *AtHome* state and the *Stopped* state, two states are used here to distinguish the fact that in the *AtHome* state the table is at the home position, which is not the case with the *Stopped* state.

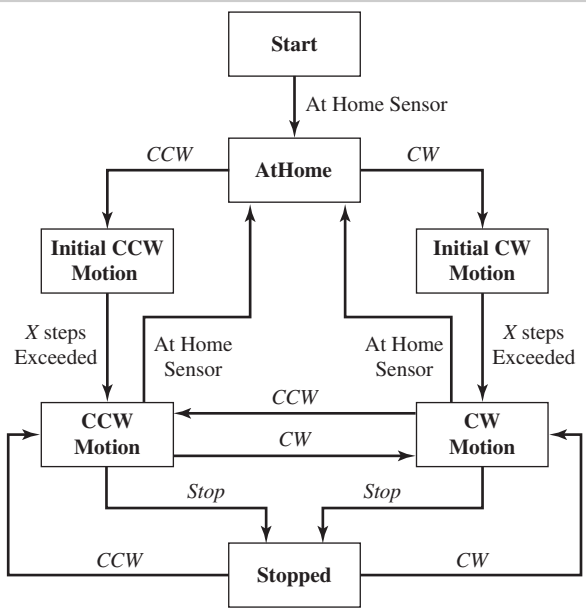


Figure 10.5

State-transition diagram for the operation of the stepper-driven rotary table

10.2.5 Microcontroller Code

The program for controlling the stage motion is implemented on a PIC16F690 microcontroller that was mounted on the low-pin-count development board. The PIC-C compiler (see Section 4.6) was used for this project. The variable definitions are shown in Figure 10.6, while the code listing for the *main()*, *GetCommand()*, *and MoveTable()* functions is shown in Figure 10.7. The code listing for the

Figure 10.6

Variable definitions for code to control the rotary stage

```
////////////////////////////////////////////////////////////////////////////////
///          RotaryTable.c
///
/// This program implements the state diagram for the stepper-motor driven
/// rotary table. The solution uses PIC16F690 using the Low pin count board
/// Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////
#include <16F690.h>                    //Include file for the particular chip used
#DEVICE ADC = 10                       //10-bit A/D return value
#use delay(internal = 8M)              //Use Internal 8 MHz- clock
#fuses INTRC_IO, NOMCLR, NOWDT, NOPROTECT, NOBROWNOUT

#define Start      1                   //Define the states
#define AtHome     2
#define InitialCWMotion  3
#define InitialCCWMotion  4
#define CWMotion     5
#define CCWMotion    6
#define Stopped    7

#define HomeSensor PIN_A1              //Define input line for Home sensor (Use A1)
#define DirectionLine PIN_C5           //Define output line for direction signal
#define CW  1                          //CW rotation direction constant
#define CCW 2                          //CCW rotation direction constant
#define STOP 3                         //Stop constant

unsigned int32 Time;                   //Variable to record time using Timer1
unsigned int16 LastCount;              //Internal variable used by GetTimeNow()
float TimerRes;                        //Resolution of Timer1

int8 EntryStartState = 0;              //Start state entry flag
int8 EntryInitialCWMotionState = 0;    //InitialCWotion state entry flag
int8 EntryInitialCCWMotionState = 0;   //InitialCCWotion state entry flag
int8 EntryCWMotionState = 0;           //CWMotion state entry flag
int8 EntryCCWMotionState = 0;          //CCWMotion state entry flag
int8 EntryStoppedState = 0;            //Stopped state entry flag

int32 StartTime;                       //Variable used for interval timing
int8 State, NextState;                 //State and NextState of transition diagram

                                       //Declaration of functions used in program
void Table_Task(void);                 //Rotary table state transition diagram function
void MoveTable(int8);                  //Function to send pulses and direction signals
                                       //to stepper motor driver
int8 GetCommand(void);
int32 GetTimeNow(void);                //Returns time in multiple of timer resolution
void SetUpTimer(void);                 //Function to setup Timer 1
```

Table_Task() function is shown in Figures 10.8 and 10.9. Note how the timer, the A/D converter, and the PWM functions are set in the *main()* routine. The code for setting and accessing the timer (Timer1 in this case) is very similar to the one shown in Section 6.3.4, so it is not repeated here. The timer is set to operate at a frequency of 1 MHz, so the *GetTimeNow()* function returns time in units that are multiples of the timer resolution (1 μs). Note how the A/D converter is set to use $F_{OSC}/16$ as the timer source for the A/D conversion. This gives a T_{AD} value of 2.0 μs (see Section 4.6.2). The PWM signal generated from channel P1A is used as the step signal for the stepper-motor driver. Using the internal 8 MHz clock, a prescale factor of 16 for timer 2, and a register period of 255, gives a PWM frequency of

```
void main(void)                          //main function
{
  SetUpTimer();                          //Setup timer 1
  setup_adc(ADC_CLOCK_DIV_16);           //Setup A/D
  setup_adc_ports(sAN0);                 //Select channel RA0 for A/D
  set_adc_channel(0);
  setup_timer_2(T2_DIV_BY_16,255,2);     //Set-up timer2 for PWM.
  setup_ccp1(CCP_PWM);                   ///At 8 MHz clock => 488.28 Hz PWM freq.
  set_pwm1_duty(0);
  NextState = AtHome;

  while ( 2 > 1)                         // Start infinite loop
  {
  Table_Task();
  } }

int8 GetCommand(void)                    //Read user commands from rotary pot
{
int8 inputv;
inputv = (int8) (read_adc()/255.0);      //Read Motion command from pot
                                         //0-255 -> Stop, 256-511-> CW, 512-767-> CCW, 768-1023 ->Stop
if (input(PIN_A3) == 0)                  //Check if switch SW1 is pressed
  {
  if ((inputv == 0) || (inputv == 3))    //Stop command
   {
   CommandV = 0;
   return (3);
   }
  else
   {
   if (inputv == 1)                      //CW command
     CommandV = 0x08;                    //Variable to lit LED C3 if CW motion
   else if (inputv == 2)                 //CCW command
     CommandV = 0;
     return(inputv);
   }
  }
else
  {
  return(0);                             // Input is valid only when SW1 is pressed
  }
 }

void MoveTable(int8 value)               //Send step and direction information to stepper driver
{
switch (value)
  {
  case CW:
   output_low(DirectionLine);            //Set direction bit for CW motion
   set_pwm1_duty(128);                   //send pulses to stepper driver
   break;

  case CCW:
   output_high(DirectionLine);           //Set direction bit for CW motion
   set_pwm1_duty(128);                   //send pulses to stepper driver
   break;

  case STOP:
   set_pwm1_duty(0);                     //Shut of pulses when table is stopped
   break;
  }
}
```

Figure 10.7

Code listing for *main()*, *GetCommand()*, and *MoveTable()* functions

Figure 10.8

Code listing for *TableTask()*

```
void Table_Task(void)
{
 GetTimeNow();                                  //Keep reading the timer to avoid overflow
 Command = GetCommand();
 output_c(State | CommandV);                    //Send state and command value to
                                                // to 4 LEDS attached to port C. C0-C2 ->State,
                                                   C3->CommandV
 State = NextState;                              //Update State variable
 switch (State)                                 //Go to active state
  {
   case Start:
   if (input(HomeSensor) == 0)                  //Is Disk at home position?
     {
     NextState = AtHome;
     MoveTable(STOP);
     EntryStartState = 0;                       //Reset Entry flag on transition to another state
     }
   else
     {
     if (EntryStartState == 0)
       {
       MoveTable(CW);
       EntryStartState = 1;                     //Set Entry Flag on first entry to state
       }
     }
   break;

  case AtHome:
      if (Command == CW)                        //Check if user selected CW motion
     NextState = InitialCWMotion;
   else if (Command == CCW)                     //Check if user selected CCW motion
     NextState = InitialCCWMotion;
   break;

  case InitialCWMotion:
   if (EntryInitialCWMotionState == 0)
    {
     StartTime = GetTimeNow();                  // Record the start time
     MoveTable(CW);
     EntryInitialCWMotionState = 1;             //Set Entry Flag on first entry to state
    }
   else if ((GetTimeNow() - StartTime) >= (1*100000))   //Check if 0.1 second has elapsed
    {
     NextState = CWMotion;
     EntryInitialCWMotionState = 0;             //Reset Entry flag on transition to another state
   }
    break;

case InitialCCWMotion:
   if (EntryInitialCCWMotionState == 0)
    {
     StartTime = GetTimeNow();                  //Record the start time
     MoveTable(CCW);
     EntryInitialCCWMotionState = 1;            //Set Entry Flag on first entry to state
    }
   else if ((GetTimeNow() - StartTime) >= (1*100000))   //Check if 0.1 second has elapsed
    {
     NextState = CCWMotion;
     EntryInitialCCWMotionState = 0;            //Reset Entry flag on transition to another state
    }
    break;
```

```
case CWMotion:
 if (EntryCWMotionState == 0)
  {
  MoveTable(CW);
  EntryCWMotionState = 1;                    //Set Entry Flag on first entry to state
  }
 else if (input(HomeSensor) == 0)            //Is Disk at home position?
  {
  NextState = AtHome;
  MoveTable(STOP);
  EntryCWMotionState = 0;                    //Reset Entry flag on transition to another state
  }
 else if (Command == CCW)                    //Check if user selected CCW motion
  {
  NextState = CCWMotion;
  EntryCWMotionState = 0;                    //Reset Entry flag on transition to another state
  }
 else if (Command == STOP)
  {
  NextState = Stopped;
  EntryCWMotionState = 0;                    //Reset Entry flag on transition to another state
  }
 break;
 case CCWMotion:
  if (EntryCCWMotionState == 0)
  {
  MoveTable(CCW);
  EntryCCWMotionState = 1;                   //Set Entry Flag on first entry to state
  }
 else if (input(HomeSensor) == 0             //Is Disk at home position?
  {
  NextState = AtHome;
  MoveTable(STOP);
  EntryCCWMotionState = 0;                   //Reset Entry flag on transition to another state
  }
 else if (Command == CW)                     //Check if user selected CW motion
  {
  NextState = CWMotion;
  EntryCCWMotionState = 0;                   //Reset Entry flag on transition to another state
  }
 else if (Command == STOP)
  {
  NextState = Stopped;
  EntryCCWMotionState = 0;                   //Reset Entry flag on transition to another state
  }
 break;
 case Stopped:
  if (EntryStoppedState == 0)
  {
  MoveTable(STOP);
  EntryStoppedState = 1;                     //Set Entry Flag on first entry to state
  }
 else if (Command == CW)                     //Check if user selected CW motion
  {
  NextState = CWMotion;
  EntryStoppedState = 0;                     //Reset Entry flag on transition to another state
  }
 else if (Command == CCW)                    //Check if user selected CCW motion
  {
  NextState = CCWMotion;
  EntryStoppedState = 0;                     //Reset Entry flag on transition to another state
  }
 break;   } }
```

Figure 10.9

Continuation of code listing for *TableTask()*

488.28 Hz (see Equation (4.3)). The stepper motor used in this project has a specification of 200 pulses per revolution, so this gives a disk rotation speed of 2.44 rev/s. The user can change the rotation speed by changing the parameters of the *setup_timer_2()* function.

The function *GetCommand()* reads the user commands. As discussed previously, the user uses a combination of rotating the pot (connected to RA0 channel) and pressing the SW1 switch (RA3 channel) to issue a *CW*, *CCW*, or *Stop* command. The A/D value obtained from reading channel RA0 is converted to a digital value (0 to 3) by dividing by 256 and converting the result to an integer form. The variable *CommandV* is used to set pin RC3, which indicates to the user if the input setting is CCW or CW.

The *MoveTable()* function controls the motion of the table by controlling the step signal. If the table is rotating either CW or CCW, the function also sets the value of the direction bit (RC5) to cause the proper motion. If the table is commanded to stop, then the PWM duty cycle is set to 0.

The *TableTask()* function implements the state-transition diagram that is shown in Figure 10.5. The coding for this function follows the material discussed in Section 6.6. Note the code division in some of the states into entry section, action section, and test and exit section, as was discussed in Section 6.6.

10.2.6 Results

The material presented in this section shows how an MCU can be used as a standalone controller for a discrete-event system. The code presented for this system was divided into modular pieces. This is done to give flexibility in case of hardware changes, so the user has only to change some of the code. For example, if the user input was changed to use a PC or a terminal to transmit commands to the MCU instead of using the rotary pot and switch SW1, then only the *GetCommand()* code needs to be changed. The reader should also keep in mind that developing a state-transition diagram for the operation of a physical system is a very important step. One needs to make sure that the state-transition diagram operates correctly on paper before coding it.

10.2.7 List of Parts Needed

Table 10.1 shows a list of the main components needed to fabricate this setup. If the low pin-count board is not available, any other development board can be used (such as The PICDEM PIC18 Explorer Demonstration Board from Microchip

Table 10.1 Main components for the stepper-motor–driven rotary table setup

Component	Manufacturer/Part #	Comments
Stepper motor	Superior Electric SLO-SYN Synchronous/Stepping motor Model # M062-LS09	Any four-phase stepper motor can work
Sensor	Photointerrupter Fairchild H21B1	
Microcontroller	Microchip low pin-count development board with PIC16F690 MCU	Any other PIC development board will work. An example is Microchip PIC18 Explorer Board with the PIC18F8722 MCU
Stepper-motor driver	Applied Motion Products PDO 2035 step-motor driver	
Disk	Any CD computer disk	The notch is created by sawing a small slit in the disk

Technology). Note that, while the availability of PIC development board will eliminate the wiring needed for the LEDS and the pot, this project still can be done just using a bare PIC16F690 or another PIC MCU.

10.3 A Paper-Dispensing System That Uses a Roller Driven By a Position-Controlled DC Motor

10.3.1 Project Objectives

This project focuses on the control of a paper-dispensing system using a PC as the control medium. The project objectives are to illustrate:

- Control of a system that involves discrete-event and feedback control activities
- Modeling and control of a positioning system
- Development of a PC GUI using VBE
- PC interfacing and data acquisition
- Use of the Performance Counter for time keeping
- Use of trapezoidal motion profile
- Simulation of a feedback controller

10.3.2 Setup Description

The toilet paper roll is pulled between two spring-loaded rollers, one of which is driven by a geared permanent-magnet brush DC motor with an incremental encoder, as shown in Figure 10.10. The gear ratio is 5.9:1, the encoder has 512 lines per revolution, and it operates in quadrature mode (see Section 7.3.3). A solenoid-operated arm acts as a stopper when a certain length of paper needs to be removed (not demonstrated here). The control signal is sent to a servo amplifier (see Figure 8.19) using a 12-bit D/A converter. The motor position is obtained from a 24-bit hardware counter that is connected to the incremental encoder.

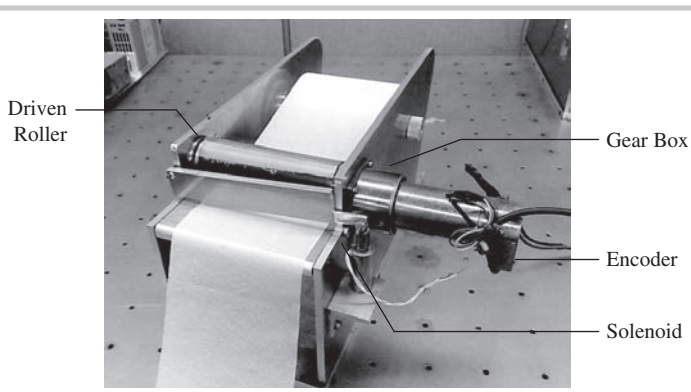


Figure 10.10

Paper-dispensing setup
(Jouaneh, University of Rhode Island)

A block diagram of the components of the control system for this setup is shown in Figure 10.11.

The signal flow and units of the relevant quantities in this system are shown in Figure 10.12. The figure assumes a simple gain for the amplifier and a linear model for motor and gearbox dynamics. The roller diameter is 1.55 inches.

Figure 10.11

Block diagram of system components

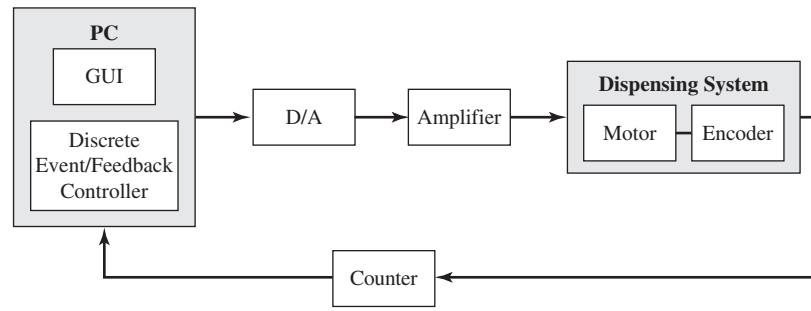
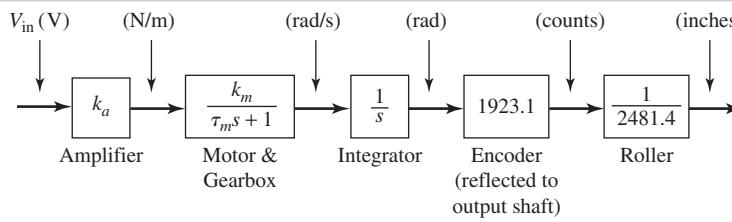


Figure 10.12

Signal flow and units of relevant quantities in paper-dispensing setup



10.3.3 User Interface

A graphical user interface (GUI) should be developed to control the operation of the paper-dispensing machine. The user specifies unwinding 'jobs' that the machine needs to perform. Each job is specified by two entries: the number of sheets of paper to be dispensed, and the speed of unwinding (in units of sheets per second). The GUI should use a form to list all of the pending jobs and should have the ability to delete a particular job after it has been entered into the system. For flexibility, the control system should allow the user to enter new jobs while a job is being executed. The system also should have several control buttons, including *START/RESUME*, *ABORT*, and *SAVE DATA*. The user interface also should display information about the current/pending/executed jobs. This information should include:

- The number of jobs waiting to be processed
- The total number of sheets waiting to be dispensed
- The number of jobs completed since the control system started
- The total number of sheets dispensed since the control system started
- The number and data of the currently executing job
- The number of seconds that has elapsed in executing the current job
- The execution time of the current job

An example of a **GUI design in VBE** (see Section 6.12.2) that meets these requirements is shown in Figure 10.13. A screenshot of the GUI while the program is operating is shown in Figure 10.14. The user selects the job data (number of sheets and speed) by clicking on the appropriate radio buttons. The job information is displayed using a four-column *ListView* control. A job is added to the list by clicking on the *Add Job* button, and a listed job is removed from the list by highlighting the job number and then clicking on the *Delete Job* button. The control system starts by clicking on the *Enable Control* Button. A job is

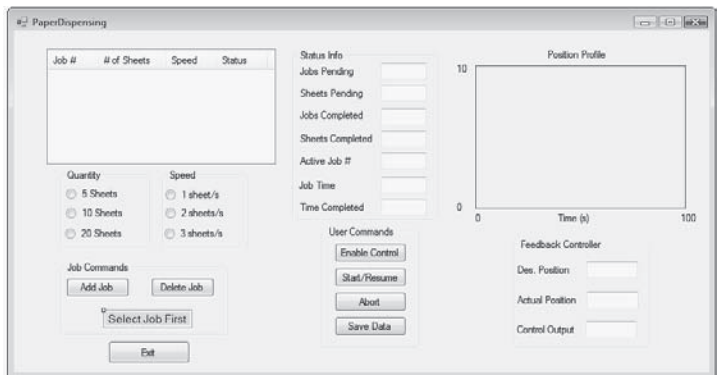


Figure 10.13

GUI design in VBE for paper-dispensing system

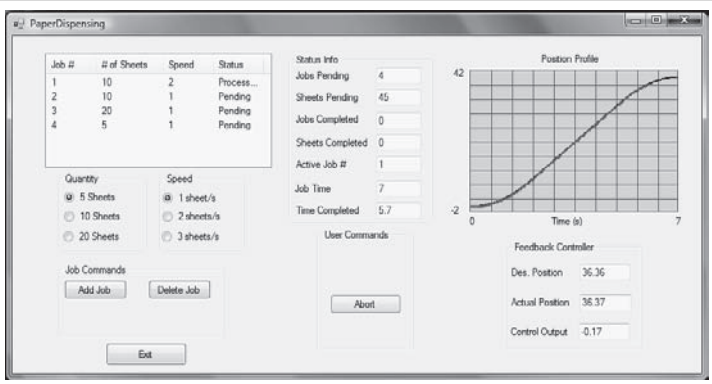


Figure 10.14

GUI for paper-dispensing system

processed by clicking on the *Start/Resume* button. While a job is being executed, its status in the job list is displayed as *Processing*. When the job completes execution, the job information is removed from the list, and the status info is updated. A currently executed job is aborted by clicking on the *Abort* button. This causes the job information to be removed from the system, and its data not to be included in the status information.

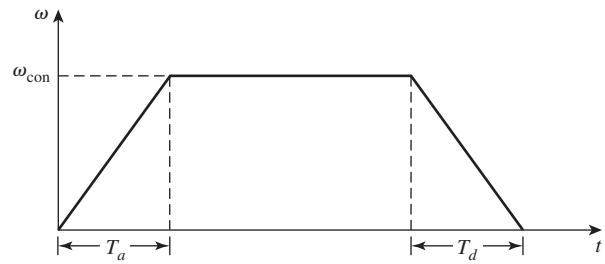
A panel control is used to graphically display the current position of the roller versus time. Both the desired and actual (or simulated) position of the roller are displayed.

10.3.4 Motion Profile

To make the roller motion smooth, a trapezoidal velocity trajectory profile should be used for planning the motion of the roller for each job. A typical trapezoidal velocity profile is shown in Figure 10.15. T_a and T_d are the acceleration and deceleration zone time intervals, respectively, and ω_{con} is the speed during the constant speed portion of the profile (corresponding to the dispensing speed). Since we are doing position control in this project, one of the things that needs to be done is to derive an expression for the displacement of the roller corresponding to this profile. This is done by integrating this profile. Once we have an expression for the angular displacement, we can use it to specify values for the desired rotation of the roller in the feedback control system.

Figure 10.15

Trapezoidal velocity profile for planning the motion of the drive roller for each job



10.3.5 Control Software

The control system for the paper-dispensing system was implemented on a PC with the code developed using VBE 2010 (see Appendix A). The code implements a **cooperative control-mode program** (see Section 6.5.2) in which the *ControlTask* is called repeatedly inside an infinite do-loop. The program enters the infinite do-loop when the user clicks on the *Enable Control* command button. The VBE *DoEvents()* method is called inside the infinite do-loop to handle any pending user commands. The variables used in the project are listed in Figure 10.16, while the code listing for the *ControlTask* is shown in Figures 10.17 and 10.18. The ***ControlTask*** has two states: an *Initial* state and an *Execution* state. The *Initial* state

Figure 10.16

Variable definitions for the paper-dispensing program

```
Dim pen1 As New System.Drawing.Pen(Color.Blue)          'Pen used in drawing graphics
Dim Sheets As Integer                                   'Number of sheets in a job
Dim Speed As Integer                                    'Job Speed in sheets/sec
Dim JobsPending As Integer                              'Number of jobs waiting to be executed
Dim SheetsPending As Integer                            'Number of sheets waiting to be dispensed
Dim JobsDone As Integer                                 'Number of jobs completed
Dim SheetsDone As Integer                               'Number of sheets dispensed
Dim ActiveJobSheets As Integer                          'Number of Sheets in current job
Dim ActiveJobSpeed As Integer                           'Desired speed for current job
Dim ActiveJobTime As Single                             'Execution time for current job
Dim ActiveJobNumber As Integer                          'Job number for current job
Dim ActiveJobElapsedTime As Single                      'Time since active job started
Dim ListInfoItem As ListViewItem                        'Listview 'view' property needs to be set to 'detail'in the design window
Dim itemnum As Integer                                  'Index for jobs on the list
Dim State, NextState As Integer                         'Variables for state transition diagram
Dim StartTime1 As Double                                'Start Time Value
Dim StartTime2 As Double                                'Start Time Value
Dim Command As String                                   'User command
Dim DesPos(10000) As Double                             'Array of desired position values
Dim ActPos(10000) As Double                             'Array of actual position values
Dim Tupdate As Double = 0.01                            'Trajectory update interval in sec
Dim Tsamp As Double = 0.001                             'Sampling Time in seconds
Dim NumAccel As Short                                   'Number of points in accel. phase
Dim NumConst As Short                                   'Number of points in constant speed phase
Dim Simulation As Boolean = True                        'Flag to indicate simulation or real operation
Dim SheetsToInches As Single = 3.65                     'Number of inches per sheet
Dim index As Short = 0                                  'Variable for stepping through the trajectory profile
Dim Vk As Double = 0.0                                  'Simulated model speed
Dim Xk As Double = 0.0                                  'Simulated model position
Dim ControlOutput As Double                             'Output from PI Controller
Dim tmr1 As New PerformanceTimer                        'Variable of type PerformanceTimer
Dim CounterResetValue As Long                           'Value of encoder when reset
```

Figure 10.17

VBE code listing for *ControlTask()*

```
Sub ControlTask()
Static xdes As Double                                   'Desired roller position
Static xact As Double                                   'Actual roller position
Static State2EntryFlag As Boolean = False               'Entry flag for state #2
Static OpenLoop As Boolean = True                       'Flag for open/closed loop operation
Static OpenLoopVoltage As Single = 7.0                  'Open loop step input voltage
State = NextState                                       'Update state variable
Select Case State                                       'Go to current state
  Case 1 'Initial State
    If Command = "Start" Then                           'Execute the following code if the user pressed the 'Start' command
      Command = ""                                      'Reset Command
      If JobsList.Items.Count <> 0 Then                 'Check if there are jobs on the list
        cmdAbort.Visible = True                         'Make Abort command button visible
        cmdStart.Visible = False                        'Hide Start command button
        ExecuteJobStartSequence()                       'Update displays related to the new job
        GenerateTraj()                                  'Generate desired trajectory data
        draw_traj()                                     'Draw desired trajectory in panel
        ActiveJobTime = (2*NumAccel + NumConst)*Tupdate 'Compute job execution time
        textJobTime.Text = ActiveJobTime                'Display job execution time
        StartTime1 = ReadTimeNow()                      'Record start time for doing control
        StartTime2 = ReadTimeNow()                      'Record start time for updating trajectory
        NextState = 2                                   'Transition to state 2
      Else
        textActiveJobNum.Text = " "                     'Clear active job display if start button was pressed with no jobs on list
        textJobTime.Text = " "
        Panel1.CreateGraphics.Clear(Color.Aqua)
      End If
    End If
  Case 2 'Execution State
    If State2EntryFlag = False Then
      If Simulation Then
        xact = 0                                        'Start at zero position in simulation mode
        If OpenLoop Then
          RollerModel(OpenLoopVoltage)
        End If
      Else
        ZeroCounter()                                   'Zero encoder output reading in real mode
        xact = ReadPosition()
        If OpenLoop Then
          SendToMotor(OpenLoopVoltage)
        End If
      End If
      ActPos(0) = xact                                  'Store the initial position in an array
      State2EntryFlag = True                            'Set entry flag to true
    End If
    If (ReadTimeNow() - StartTime1) >= Tsamp Then       'Check if it is time to do control
      xdes = DesPos(index)                              'Get desired position from created profile
      If Simulation Then
        If OpenLoop = False Then
          xact = RollerModel(ControlOutput)             'Set actual position to model output in simulation mode
        Else
          xact = RollerModel(OpenLoopVoltage)
        End If
      Else
        xact = ReadPosition()                           'Set actual position to encoder supplied data in real mode
      End If
      ControlOutput = PIControl(xact, xdes)             'Call PI control routine
        If Simulation = False Then
          If OpenLoop = False Then
            SendToMotor(ControlOutput)                  'Send control output to motor in real mode
          End If
        End If
        StartTime1 = ReadTimeNow()                      'Record start time for doing control
      End If
```

Figure 10.18

Continuation of code listing for *ControlTask()*

```
  If (ReadTimeNow() - StartTime2) >= Tupdate Then      'Check if it is time to update the trjectory
    index = index + 1                                  'Increment counter for trajectory
    ActPos(index) = xact                               'Store current position
    StartTime2 = ReadTimeNow()                         'Record start time for trajectory update
    draw_currentpoint(index)                           'Update current position in panel window
  End If

  If Command = "Abort" Then                            'Check if user pressed 'Abort' command
    Command = ""                                       'Reset Command
    ExecuteAbortSequence()                             'Update displays to reflect aborted job
    cmdAbort.Visible = False
    cmdStart.Visible = True
    State2EntryFlag = False                            'Reset entry flag to its initial value
    NextState = 1                                      'Transition to state 1
  End If
  If (index >= (2*NumAccel + NumConst)) Then           'Check if profile is completed
    ExecuteJobEndSequence()                            'Update textboxes to reflect profile completion
    State2EntryFlag = False                            'Reset entry falg to initial value
    cmdSaveData.Visible = True                         'Make visible save data button
    cmdStart.Visible = True                            'Make visible start button
    NextState = 1                                      'Transition to state 1
  End If

End Select
```

(state 1) waits for the user to click on the *Start/Resume* command button. Once this button is clicked, and there is pending job on the list, the program goes through a series of activities before transitioning to the *Execution* state (state 2). These activities include updating displays related to the current job, generating the desired trajectory data from the given job information, and drawing the desired trajectory in the control panel. The **code listing for generating the desired trajectory** data is shown in Figure 10.19. Note that the trajectory generation routine can generate a profile with either a fixed-acceleration time interval (T_a) or a variable-acceleration time interval, depending on the setting of the *FixedAccelerationRate* flag. The latter case keeps the acceleration rate constant for any desired constant speed. The trajectory generation routine handles also the case where there is no constant speed interval, which happens when the desired displacement is short. In this case, the resulting velocity profile is triangular and not trapezoidal.

In the entry part of the *Execution* state, the program records the initial position of the roller. In the active part of the state, the program performs a PI closed-loop control (see Section 9.6) of the roller position every *Tsamp* interval, where the current and desired roller displacements are supplied to the PI control function to compute the control output that is sent to the motor through the D/A converter (see Section 5.4). The desired roller displacement is updated every *Tupdate* interval, where the *Tupdate* interval (10 ms) is normally larger than the *Tsamp* interval. The *Performance Counter* (see Section 6.3.3) is used to obtain timing information since the built-in timer in VBE has a coarse resolution of about 15 ms, and we used a *Tsamp* value of 1 ms. The actual position profile is updated in the drawing panel every *Tupdate* interval.

The *ControlTask* jumps back to the *Initial* state when either the travel time for the current job time has been completed or the *Abort* command was issued by the user. In normal completion, the user can click the *Save* command button

```
Private Sub GenerateTraj()
  Dim Accel As Single = 1                                  'Acceleration in inches/sec
  Dim FixedAccelerationRate As Boolean = False             'Flag to indicate fixed acceleration rate mode
  Dim SheetLength As Single                                'Total length of sheets (in inches) for active job
  Dim Vconst As Single                                     'Constant speed value in in/sec for active job
  Dim Ta As Single = 2.0                                   'Accelertion time in sec
  Dim Tconst As Single                                     'Constant speed time interval in sec
  Dim Ttotal As Single                                     'Total time for profile
  Dim DistAcc As Single                                    'Distance travelled during acceleration/deceleration phases
  Dim m As Short
  SheetLength = ActiveJobSheets*SheetsToInches             'Convert sheets data to inches
  Vconst = ActiveJobSpeed*SheetsToInches                   'Convert speed data to inches/sec

                                                           'Determine the profile time parameters
  If FixedAccelerationRate = True Then
    Ta = Vconst / Accel
  End If
  DistAcc = 0.5*Ta*Vconst                                  'Distance travelled during acceleration phase
  If (2*DistAcc) >= SheetLength Then                       'Check if there is no constant speed phase
    DistAcc = 0.5*SheetLength                              'For triangular vel. profile, set accel. distance to half of travel distnace
    Tconst = 0.0                                           'Constant speed time interval is zero for a triangular velocity profile
    Vconst = (2*DistAcc) / Ta                              'Compute peak velocity
    Ttotal = 2*Ta                                          'Compute profile time
  Else
    Tconst = (SheetLength - 2*DistAcc) / Vconst            'Compute constant travel time for a trapezoidal velocity profile
    Ttotal = 2*Ta + Tconst                                 'Compute profile time
  End If
  NumAccel = Int(Ta / Tupdate)                             'Convert time interval to a count
  NumConst = Int(Tconst / Tupdate)

                                                           'Generate points during acceleration phase
  For i As Short = 1 To NumAccel
    DesPos(i) = 0.5*(i*Tupdate)*(i / NumAccel)*Vconst
  Next
                                                           'Generate points during constant speed phase
  For j As Short = (NumAccel + 1) To (NumAccel + NumConst)
    DesPos(j) = DesPos(NumAccel) + Vconst*(j - NumAccel)*Tupdate
  Next
                                                           'Generate points during deceleration phase
  For k As Short = (NumAccel + NumConst + 1) To (2*NumAccel + NumConst)
    m = k - (NumAccel + NumConst)
    DesPos(k) = DesPos(NumAccel + NumConst) + 0.5*m*Tupdate*(Vconst + Vconst*((NumAccel - m) / NumAccel))
  Next
```

Figure 10.19

VBE code for generating the desired trajectory

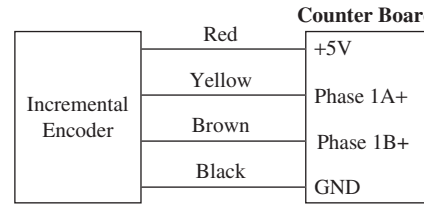
to save in a file the actual (or simulated) and desired roller displacement data versus time.

The feedback information from the **incremental encoder** that is attached to the motor is read by a 24-bit hardware counter that is part of the Measurement Computing PCI-QUAD04 four-channel quadrature encoder board. The connection diagram for the encoder and the **counter board** are shown in Figure 10.20. Note that the 24-bit counter will overflow once every 16.77 million counts. This is more than enough to completely dispense a 1000-sheet roll of paper.

The code for the *Add Job* and *Delete Job* commands that manage the job list are shown in Figure 10.21. Note that the variables *Speed* and *Sheets* that appear in the

Figure 10.20

Connection diagram between incremental encoder and counter board




```
Private Sub cmdAddJob_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles cmdAddJob.Click
    LabelSelectJob.Visible = False                          'Hide warning label if it was displayed
    itemnum += 1                                            'Increment counter for number of jobs
    ListInfoItem = JobsList.Items.Add(itemnum)              'Add job number to 1st coloum in job list
    ListInfoItem.SubItems.Add(Sheets)                       'Add number of sheets to 2nd column in job list
    ListInfoItem.SubItems.Add(Speed)                        'Add job speed to 3rd column in job list
    ListInfoItem.SubItems.Add("Pending")                    'Add job status to 4th column in job list
    JobsPending += 1                                        'Increment the number of pending jobs
    textJobsPending.Text = JobsPending                      'Update display in jobs pending text box
    SheetsPending = SheetsPending + Sheets                  'Update the number of sheets pending
    textSheetsPending.Text = SheetsPending                  'Update display in sheets pending text box
End Sub

Private Sub cmdDeleteJob_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles cmdDeleteJob.Click
    If JobsList.SelectedItems.Count = 1 Then                'Check if only one job is selected for deletion at a time
      If JobsList.SelectedItems(0).SubItems(3).Text <> "Processing" Then 'Delete only pending jobs
        LabelSelectJob.Visible = False                      'Hide warning label
        SheetsPending = SheetsPending - JobsList.SelectedItems(0).SubItems(1).Text 'Update sheets pending data
        JobsList.Items.Remove(JobsList.SelectedItems(0))    'Delete selected job data
        JobsPending -= 1                                    'Decrement jobs pending counter
        textJobsPending.Text = JobsPending                  'Update jobs pending display text box
        textSheetsPending.Text = SheetsPending              'Update sheets pending text box
      End If
    Else
      LabelSelectJob.Visible = True                         'Display label if no jobs were selected or more than one job is selected
    End If
End Sub
```

Figure 10.21

VBE code listing for the *Add Job* and *Delete Job* commands

AddJob function are updated in the code which handles the event associated with selecting the speed or sheet number radio buttons, respectively.

10.3.6 Modeling and Simulation of System

In this system, the angular displacement of the driven roller is used to control the quantity of the paper to be dispensed. Due to the use of a geared DC motor, any torque applied to the driven roller by the tension in the paper has a minimal impact on the motor shaft. The torque due to friction in the motor bearings and gears has a more important effect. To identify the dynamics of the system, a series of step-input voltages are applied to the motor, and the angular displacement of the motor is recorded using the PC software discussed in the previous section. To obtain this data, the *OpenLoop* flag was set to true (see Figure 10.17). A plot of the open-loop step-input position response is shown in Figure 10.22 for three different input voltages sent by the D/A to the amplifier. The figure shows that the system is nonlinear, since the final position reached for each input voltage is not proportional to the input voltage. This nonlinearity is caused primarily by nonlinear friction, which is typical of many positioning systems.

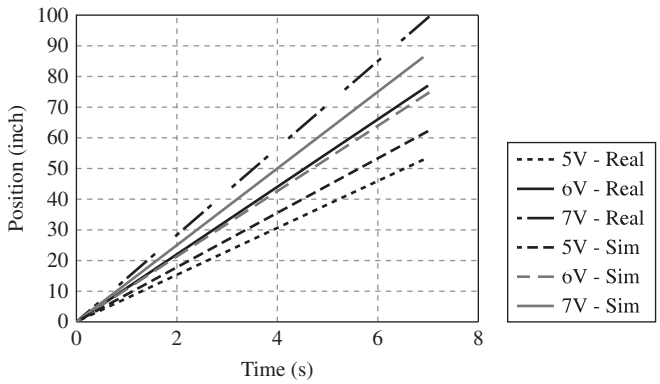


Figure 10.22

Measured and simulated open-loop step position response of driven roller to step-input voltages

The position data is digitally differentiated to obtain the speed response. A plot of the open-loop position and velocity step responses are shown in Figure 10.23.

Using the speed data for the 6 V case, the parameters of a linear first-order model can be identified as discussed in Section 9.5. The model, which relates the output speed of the driven roller (inches/s) to the input voltage applied to the amplifier, is given by Equation (10.1), and it will be utilized in the design of a feedback controller to control the roller motion in the next section.

$$\frac{\nu(s)}{V_{\text{in}}(s)} = \frac{k_s}{\tau_s s + 1} = \frac{1.833}{0.04s + 1} \tag{10.1}$$

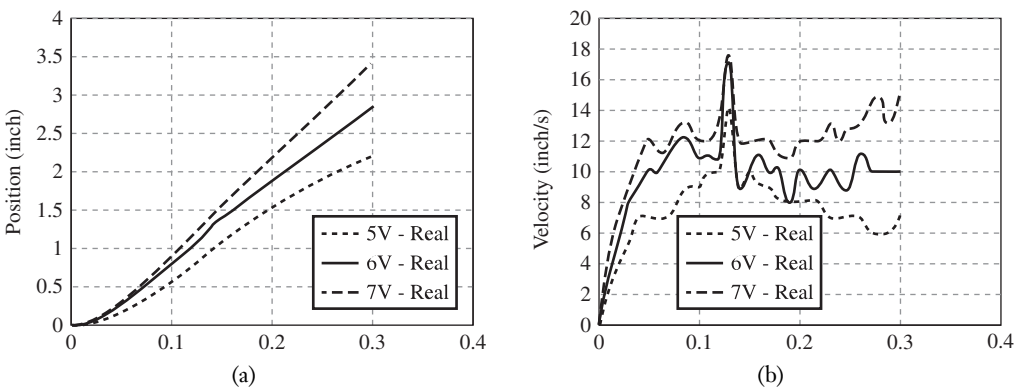


Figure 10.23

(a) Measured open-loop step position response of driven roller and (b) measured (differentiated) open-loop step velocity response of driven roller to step-input voltages

In the model, k_s is the open-loop gain of the system, and τ_s is the time constant of the system. The open-loop position step response of the dynamic model given by Equation (10.1) was simulated, and the simulated response is plotted in Figure 10.22. The data is obtained by setting the *OpenLoop* and *Simulation* flags to 'true' in the software. Note the close correspondence between the actual and simulated response for the 6 V case (which is the case on which the model is based) but the divergence for the 5 and 7 V case.

10.3.7 Feedback Controller Simulation in MATLAB

A model of a PI closed-loop controller was created in Simulink to simulate the response of the system. The model is shown in Figure 10.24. The motor-gear dynamics are represented by the linear model given in Equation (10.1).

Figure 10.24

Simulink model for roller-position control system

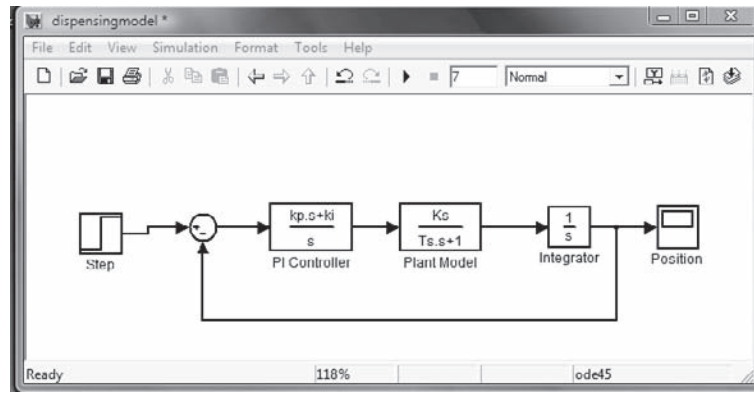
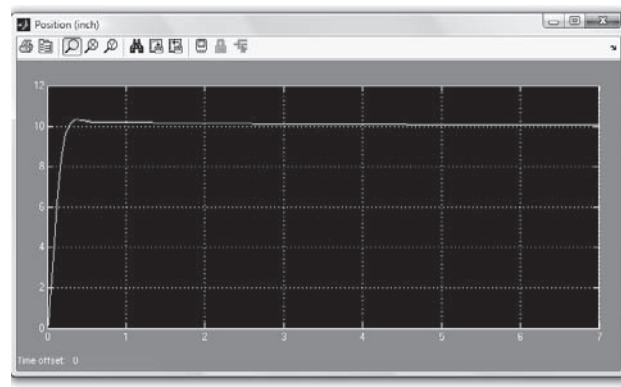


Figure 10.25

Simulated closed-loop step position response in MATLAB



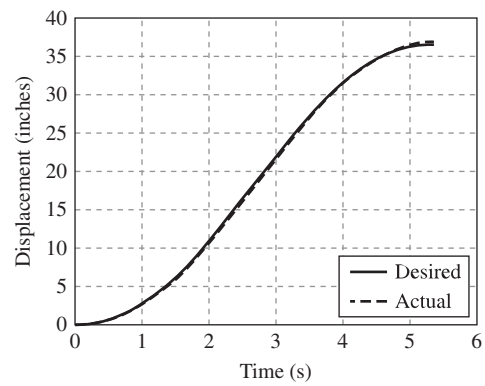
The Simulink model was used to determine control gains that give a satisfactory performance. Figure 10.25 shows the closed-loop position step response obtained in MATLAB for K_p and K_i gains of 5 and 1, respectively. As expected, Figure 10.25 shows that the PI controller produces a response with zero steady-state error.

10.3.8 Results

Figure 10.26 shows the desired and actual position profiles for a 10-sheet job at 3 sheets/sec dispensing speed. The data was obtained while the machine was running in closed-loop control using the PI gains determined in the previous section. The data is plotted from the data file that is written when the user clicks on the

Figure 10.26

Desired and actual displacement profiles for a 10-sheet job at 3 sheets/sec



Save command after running a job. Figure 10.26 shows that a closed-loop control system achieves a good job of controlling the motion of the driven roller in spite of the nonlinearities present in the system.

This project can be run in either real or simulation mode by simply changing the value of the *Simulation* flag inside the code. The simulation mode allows the user to run the software with no need for any hardware or data-acquisition system. In the **simulation mode**, the actual hardware (motor, encoder, and drive amplifier) is replaced by a dynamic model of the system in the software. Instead of reading the actual roller position from the counter, the position is obtained from the model output. Similarly, instead of using the D/A to send the control output to the amplifier, the control output is sent to the model. No other changes are needed in the code for the simulation mode except for the two mentioned. The model is numerically integrated using the Euler method, and Figure 10.27 shows the code listing that implements a simulated model of the system.

```
Private Function RollerModel(ByVal ControlInput As Double) As Double
    Dim Inertia As Double = 0.02182          'Ks = 1.833, tau = 0.04
    Dim Bviscous As Double = 0.5456          'Bviscous = 1/Ks, Inertia = tau x Bviscous
    Dim DeltaT As Double                     'Integration time interval
    Dim AccelV As Double                     'Computed acceleration
    Dim Xk1, Vk1 As Double                   'Computed position and velocity

    DeltaT = Tsamp                           'Tsamp is set to 0.001 s
    AccelV = (1 / Inertia)*(ControlInput - Bviscous*Vk)
    Vk1 = AccelV*DeltaT + Vk
    Xk1 = Xk + Vk*DeltaT + 0.5*AccelV*DeltaT*DeltaT
                                             'Update the initial values
    Vk = Vk1
    Xk = Xk1
    Return (Xk1)                             'Return position in inches
End Function
```

Figure 10.27

VBE code listing for simulating the motor/encoder system

It should be noted that the PI feedback controller implemented in this project used the **Performance Counter** (see Section 6.3.3) for timing the execution of the controller. The Performance Counter, which has a sub-microsecond resolution, is called repeatedly through the use of the *ReadTimeNow()* function (see code in Figure 10.17) to determine if one *Tsamp* interval has elapsed since the last time the controller was called. If that is the case, then the controller is run, and the process repeats. Since the feedback controller is implemented as part of a control task operating in cooperative control mode on a PC platform, there is no guarantee that the feedback controller will be called exactly every *Tsamp* interval. While there is only one task in this project (and thus there is no other task to compete for the computational resources), the application is running on a PC platform where many processes are active at the same time and sharing the computational resources (see Section 6.8). Thus, there is the possibility that another process could be executing when it is time to execute the feedback controller. This results in a delay or an effective *Tsamp* interval larger than the nominal *Tsamp* interval. The average delay however is very small, especially on high-speed processors (> 2 GHz clock speed) and reasonable sampling rates (few kHz).

This project illustrates the integration of a discrete-event task (dispensing of paper) with a feedback-control task (controlling the displacement of the driver roller), and should serve as an illustration for the control of many other mechatronic systems that involve a combination of these types of tasks.

10.3.9 List of Parts Needed

A list of the main parts (excluding support frame) needed for this project are given in Table 10.2. Note that while fabricating a paper-dispensing system gives the user the experience of controlling a 'real' piece of machinery, most of the tasks required in this project can be achieved by just using a motor with an incremental encoder without the need to fabricate the complete setup. A simulated roller diameter can be specified in software to dispense 'simulated' paper lengths.

Table 10.2 Main components needed for the paper-dispensing project

Component	Manufacturer/Part #	Comments
DC-geared motor with incremental encoder	Pitman GM9236C 534-R2\Motor	
Data-acquisition card	Measurement Computing PCIM-DAS1602/16	Any data-acquisition card will work, provided that the manufacturer provides a library of VBE functions for accessing the hardware
Counter board	Measurement Computing PCI-QUAD04 four-channel quadrature encoder board	
Motor amplifier	12A8 PWM amplifier from Advanced Motion Controls	

10.4 A Temperature-Controlled Heating System That Uses a Heating Coil, a Copper Plate, and a Temperature Sensor

10.4.1 Project Objectives

This project focuses on the implementation of a feedback controller for a dynamic system in a microcontroller and the use of the PC as a GUI for the control system operation. The project objectives are to illustrate:

- Implementation of a feedback controller in a PIC MCU
- Development of a GUI for the control system operation
- Communication between a PC and a PIC MCU
- Use of a BackgroundWorker thread for a lengthy task
- Illustration of the use of interrupts to schedule control tasks
- Determination of feedback-control gains
- Simulation of a control system in MATLAB
- Interfacing of sensors and actuators to a microcontroller

10.4.2 Setup Description

The experimental hardware (see Figure 10.28) consists of a small rectangular (50.8 mm $\times$ 38.1 mm $\times$ 12.7 mm) copper plate heated by a 10-W flexible

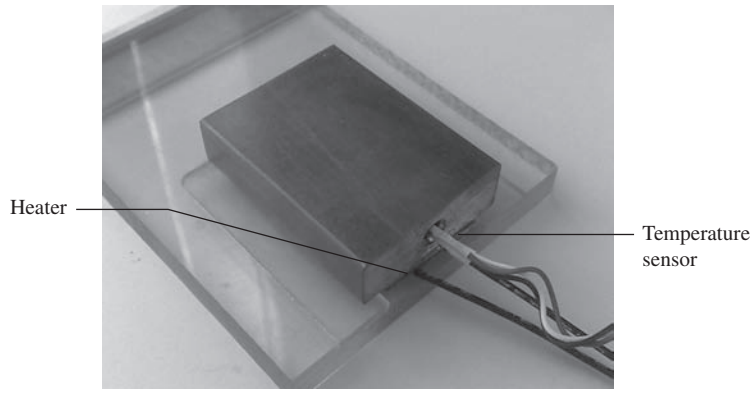


Figure 10.28

Plate and heater experimental setup
(Jouaneh, University of Rhode Island)

silicone-rubber heat strip that is glued to the bottom of the plate. The plate is mounted horizontally on a 76.2 mm × 102 mm polycarbonate base that acts an insulator. A small hole is drilled into one side of the plate, and a thermo-transistor temperature sensor (LM35C plastic package from National Semiconductor) is inserted into the plate to read to read the temperature of the plate. The temperature sensor has a sensitivity of 10 mV/°C, and a measurement range of −40 to 110°C.

While we have a choice of several microcontrollers to use for this project, we have chosen the **PIC18F8722** MCU in this project. The primary reason for using the PIC18F8722 is that it has a large RAM capacity (3936), which allows the user to store data without the need to use an external RAM chip. If another MCU was used, additional RAM can be added by using a chip such as the RAMTRON FM24C256 chip and by using the I^2C interface (see Section 5.9) to read and write to the external RAM chip. Instead of using a stand-alone MCU, we have used a commercially available development board (PICDEM™ PIC18 Explorer Demonstration Board from Microchip Technology, Inc. in Figure 10.29). The development board has a built MAX-232 chip (see Section 4.7.8) and an RS-232 connector (which simplifies the interfacing of the MCU to a PC using a standard RS-232 DB9 cable). Other features of this development board are the availability of eight built-in LEDs and a two-line character display LCD (not demonstrated here).

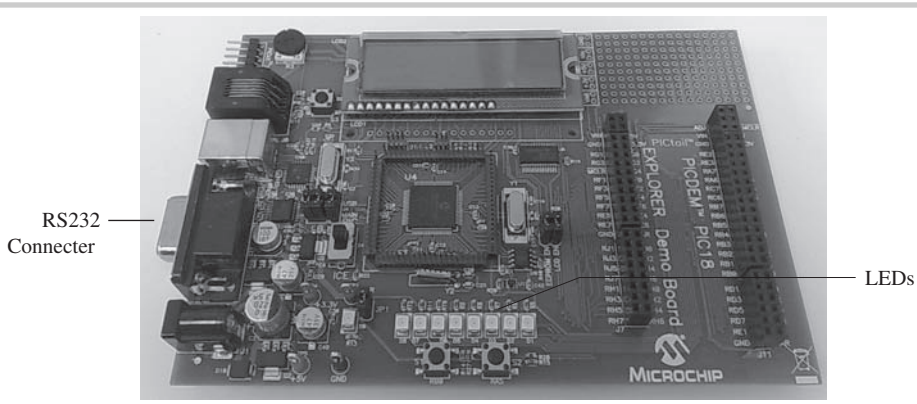


Figure 10.29

PICDEM™ PIC18 Explorer Demonstration Board
(Jouaneh, University of Rhode Island)

A block diagram of the components of the system is shown in Figure 10.30. The PIC18F8722 MCU implements a feedback controller to control the plate temperature. A PC acts as a GUI interface for this control system and uses the RS-232 serial line to communicate with the MCU. The control input to the heater is supplied

Figure 10.30

A block diagram of the control system components

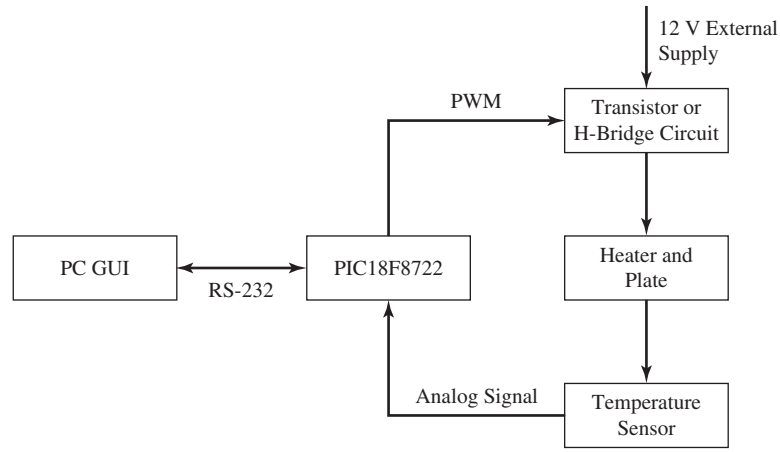
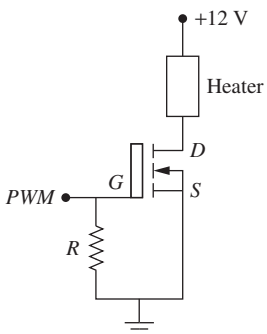


Figure 10.31

Interface circuit between MCU and heater



from the PWM output of the microcontroller through either a transistor or an H-Bridge driver. The temperature is measured using the 10-bit A/D converter on the microcontroller. With a voltage reference of 5.0 V for the A/D, the temperature measurement resolution is 0.488°C. The heat output rate q from the heater is directly proportional to the heater voltage v as $q = Kv$, where $K = 10/12$ W/V.

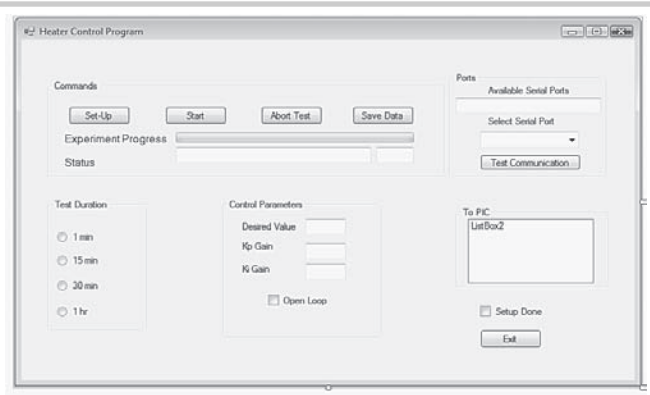
The interface circuit between the microcontroller and the heater is developed using the material covered in Chapter 3. Figure 10.31 shows such a circuit using the IRFZ14 power transistor. The heater element is connected to the drain line. The PWM output of the MCU is connected to the gate input of the IRFZ14 transistor (see Section 3.5) to modulate the 12 volt external supply to the heater. The IRFZ14 has a 10 A maximum drain-current rating and a power rating of 43 W, which is more than sufficient to drive the 10 W heater.

10.4.3 VBE PC User Interface

A GUI for the PC portion of the control program was developed in VBE. The GUI design is shown in Figure 10.32. The GUI allows the user to enter the control test parameters as well as the ability to start and abort the control program. The user interface allows the user to select one of four test duration times (1 minute, 15 minutes, 30 minutes, and 1 hour) plus the capability to enter the control gains (K_p and K_i) as well as the desired temperature (or open-loop input voltage). The program also allows the user to run the system in open-loop fashion if the user checks the *Open Loop* check box. In this case, the entry in the *Desired Value* textbox is interpreted as the open-loop voltage (0 to 12 V) to be sent to the heater.

Figure 10.32

GUI design for heater control



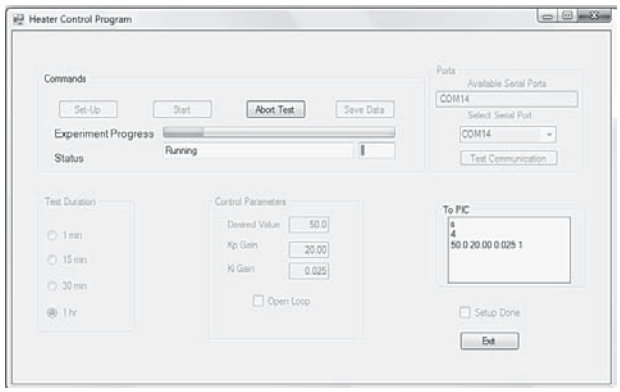


Figure 10.33

A screen shot of the control program in operation

A screenshot of the GUI in operation is shown in Figure 10.33. The user first clicks on the *Set-Up* command to set the parameters for the control system. These include the duration time and the feedback-control parameters. Once the experimental parameters are selected, the user checks the *Setup Done* check box. This disables the *Set-Up* selections and enables the *Start* command, which starts the control system upon pressing. The control progress is indicated by a progress bar, but the user can abort the control system by pressing the *Abort Test* command. When the control is completed, the *Save Data* command is enabled, which allows the user to store the collected data into a file upon pressing it. The collected data then can be imported into plotting software (such as Excel).

When the user clicks on the *Start* button, the program calls a control task (see Section 6.4) that controls the interaction between the PC and the MCU. The state transition diagram for the control task is shown in Figure 10.34. The PC acts as the master that initiates all communication between the two devices. Since the PC and the MCU are running independently, a handshaking mechanism is employed in the transfer of data between the two programs to ensure that the data is transmitted properly. No new data is sent from the PC to the MCU unless the PC receives an acknowledgment from the MCU on the previous data transfer. For example, in the *Start* state, after it sends a start character to the MCU, the PC waits to receive an acknowledgment from the MCU before it transitions to the *Second* state.

Note that for transmitting data after the control experiment is performed, the PC GUI uses a ***BackgroundWorker* thread** (see Section 6.9) to signal the communication. The code for this portion of the program is shown in Figure 10.35. The call to setup the *BackgroundWorker* thread is performed just before entering the fifth state. Note that, since the PC knows how many data points need to be sent back from the MCU, it will exit the *BackgroundWorker* thread once the required number of data points has been received.

10.4.4 Microcontroller Code

The code structure implemented in the MCU is shown in Figure 10.36. The structure consists of a state-transition diagram to sequence the control activities and communication with the PC and an **interrupt service routine** that executes the feedback controller. The state-transition diagram is shown in Figure 10.37 and has five states. The MCU begins in the *Start* state waiting for a start signal ('s' character) from the PC. Upon receiving this character, it sends it back to the PC as an acknowledgment. It then transitions to the *Second* state and waits to receive the

Figure 10.34

State-transition diagram for PC GUI

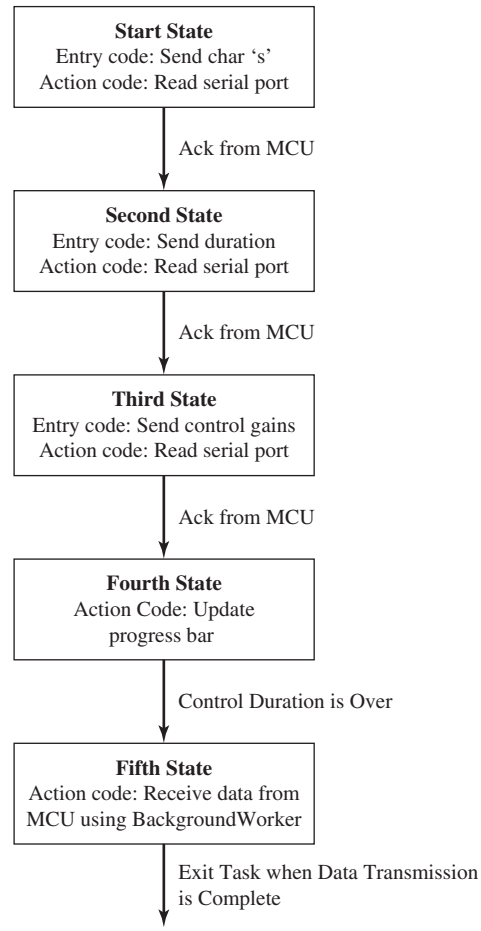


Figure 10.35

BackgroundWorker code in VBE for transmitting experiment data from MCU to PC

```
Private Sub BackgroundWorker1_DoWork(ByVal sender As System.Object, ByVal e As System.ComponentModel.DoWorkEventArgs)
Handles BackgroundWorker1.DoWork
    i = 0                                   ' Counter for number of data elements
    Dim vstr As String                      ' String variable

    While (i <= ((TestDuration / Tsamp)))   'Loop until the specified number of data points has been received by the PC

      SendtoSerialPort("d")                 'Send char 'd' to MCU to inform MCU to send data
      vstr = SerialPort1.ReadLine()         'Read data from serial port

      xdata(i, 1) = i                       'Store recieved data in an array
      xdata(i, 2) = Val(vstr)

      i = i + 1                             'Increment counter for number of data elements received
      If ((i Mod 200) = 0) Then             'Send number of data elements received to GUI
        Me.SetText2(Str(i))
      End If
      If BackgroundWorker1.CancellationPending = True Then
        SendtoSerialPort("a")               'Abort control job if requested by user

      End If
    End While

  End Sub
```

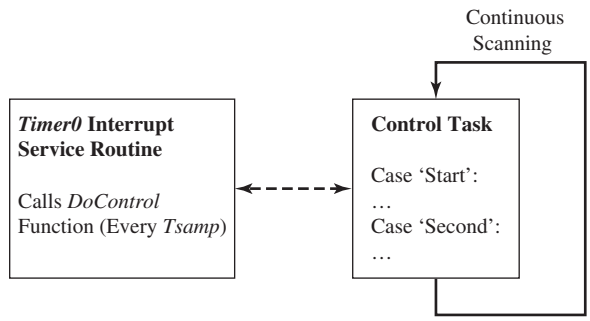


Figure 10.36
Code structure in MCU

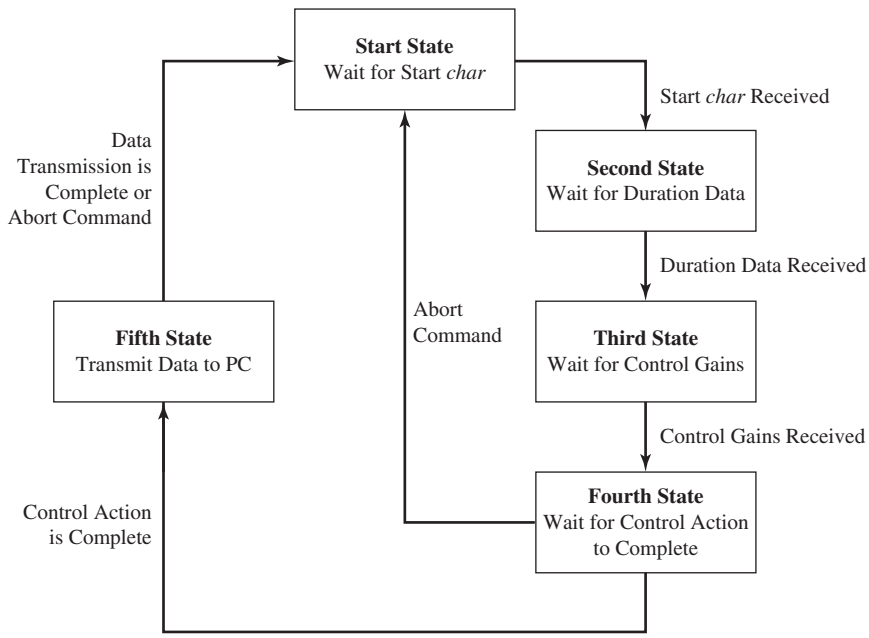


Figure 10.37
State-transition diagram for code in MCU

experiment duration time data. In a similar fashion, once that data is received, an acknowledgment is sent back to the PC, and the MCU transitions to the *Third* state, where it waits to receive the desired temperature and the control gains (or the open-loop control voltage). Just before the state diagram enters the *Fourth* State, it sets and enables a *Timer0* overflow interrupt (see Section 4.8). The state diagram waits in this state until the required number of control samples has been completed. At that point, it transitions to the *Fifth* state and waits for commands from the PC to transmit the measured temperature data to the PC. When all of the data has been transmitted or an *Abort* command was received from the PC, the program disables interrupts and moves back to the *Start* state to wait for another start signal to begin the process again.

The execution rate of the interrupt service routine is controlled by the setting parameters for *Timer0*. Using a prescsale factor of 1:1 and a 10-MHz F_{OSC} frequency, the 16-bit *Timer0* will overflow at the rate of 38.14 times per second. The ISR will decrement a counter that is set to the desired number of interrupts per sampling interval. For a *Tsamp* of one second, it is set to 38. For a *Tsamp* of two seconds, it is set to 76. The two-second *Tsamp* interval is used for the one hour-long control test so that the collected data (1800 points of *int16*) can be stored within the

available RAM space on the chip. Note that, due to the use of an integer number of interrupts per sampling interval, the actual *Tsamp* interval is slightly shorter than one or two seconds by less than 0.4%.

At each *Tsamp* interval, the ISR calls the *DoControl* function, which in turn calls the *PIControl* function to compute the control output. The code listing for the *DoControl* and *PIControl* functions are shown in Figure 10.38. The measured temperature is converted to volts, and the control output from the PI controller is converted to duty cycle. Note that if an open-loop mode was specified, the *DoControl* function simply would store the measured temperature, skip computing and sending a control output signal.

Figure 10.38

DoControl and *PIControl* functions implemented on MCU

```
////////////////////////////////////////////////////////////////////////////////////////////////
///   Compiler: PCWH from CCS, Inc. (Version 4.103)
////////////////////////////////////////////////////////////////////////////////////////////////
void DoControl(void)                          //Routine that is called inside Timer0 ISR
{
float error,controlout;
int16 addata;

addata = read_adc();                          //Read actual temperature
datastorage[j] = addata;                      //Store temperature in an array
j = j + 1;

if (ClosedLoop == 1)                          //closed loop
  {
   error = desired - addata*0.0048828;        //A/D to volts units
   controlout = PIControl(error);             //Call pidcontrol routine to compute control output
   duty = (controlout/12.0)*1023;             //Convert control output to duty cycle
   if( duty >= 0 )                            //Ignore negative control output
   {
     if (duty >= 1023)                        //Check if duty cycle exceeds limit
       {
       duty = 1023;
       }
    set_pwm4_duty(duty);                      //Send PWM signal to transistor or H-bridge
   }
  }
  counter = counter + 1;                      //Increment counter that keeps count of the
                                              // number of control cycles
}

float PIControl(float error)
{
float out;
out = kp*error + ki*sumerror*Tsamp;           //P and I terms of PI controller. Current error is not used in
                                              //Computation of I-term (see Equation 9.19)
sumerror = sumerror + error;                  //Update sum of errors expression
return (out);
}
```

10.4.5 Modeling and Simulation of Physical System

Using the Conservation of Heat Energy principle and Newton's law of cooling [38], a basic model of the copper plate (excluding radiation effects) is

$$RC\frac{dT}{dt} = T_a - T + Rq \tag{10.2}$$

where T = plate temperature, T_a = ambient temperature, q = heater output (W), C = thermal capacitance, and R = convective resistance.

The solution is (assuming that $T(0) = T_a$)

$$T(t) = T_a + Rq(1 - e^{-t/RC}) \tag{10.3}$$

The parameters R and C can be determined experimentally from analyzing the open-loop temperature response of the plate to a given heat input. For example, using the data in Figure 10.39, R is 8.00°C/W, and the time constant τ (or RC) is 1100 s. Figure 10.39 also shows the solution of the model. The model shows a close agreement with the data.

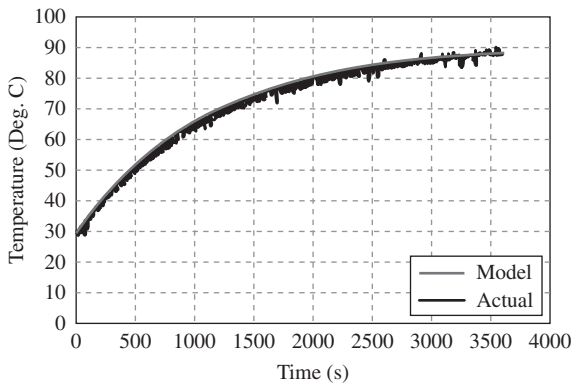


Figure 10.39

Open-loop response of the plate/heater system with q = 7.5 W (9 V)

Letting $\Delta T = T - T_a$, the plate model transfer function with the heater voltage V ($q = KV$) as the input is

$$\frac{\Delta T(s)}{V(s)} = \frac{RK}{RCs + 1} \tag{10.4}$$

Using PI control action, the closed-loop transfer function (see Section 9.5) is

$$\frac{\Delta T(s)}{\Delta T_R(s)} = \frac{RK(K_P s + K_I)}{RCs^2 + (RKK_P + 1)s + RKK_I} \tag{10.5}$$

where ΔT_R is the desired value of ΔT.

For a damping ratio of $\zeta = 1$ and a desired closed-loop time constant τ_d, the PI gains can be calculated as

$$K_P = \frac{2\tau/\tau_d - 1}{RK} \tag{10.6}$$

and

$$K_i = \frac{(RKK_P + 1)^2}{4\tau RK} \tag{10.7}$$

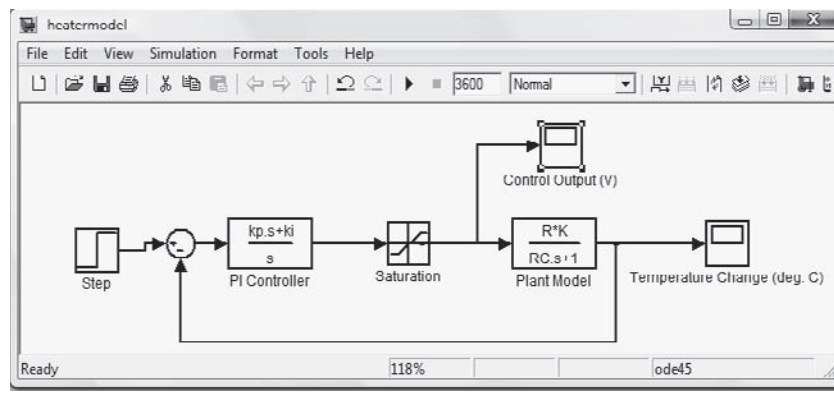
Note that, in implementing the PI controller in the PIC MCU, the control gains obtained from Equations (10.6) and (10.7) need to be multiplied by a factor of 100, because the temperature sensor has an output sensitivity of 0.01 V/°.

10.4.6 Controller Simulation in MATLAB

A Simulink model of the heater control system is shown in Figure 10.40. The model incorporates a saturation limit of 12 V to simulate the maximum voltage that can be sent to the heater for 100% duty cycle.

Figure 10.40

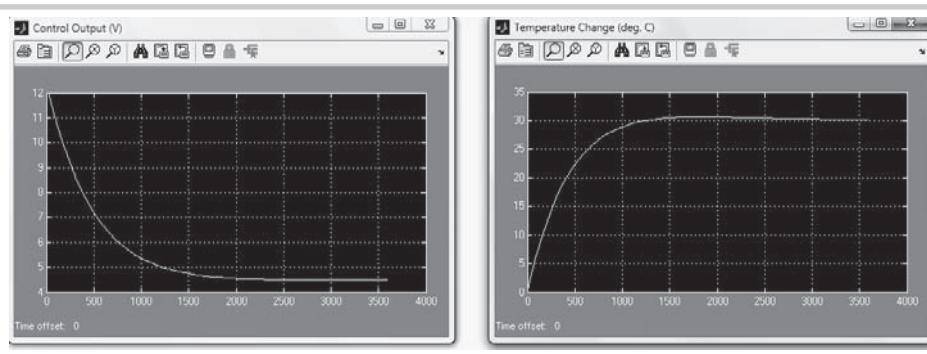
Heater model in MATLAB



A plot of the simulated closed-loop temperature response of the system using the gains $K_p = 0.4141$ and $K_i = 4.824 \times 10^{-4}$ for a 30° step change in temperature is shown in Figure 10.41. Since the heater voltage is limited to 12 V, if τ_d is selected too small, the heater will saturate. The Simulink model allows us to investigate how small τ_d could be made without causing saturation. It was found that τ_d close to 585 s was the smallest possible value.

Figure 10.41

Simulated response of heater control system in MATLAB: (a) control output (volts) and (b) temperature change (°C) with the horizontal axis indicating time in seconds



(a) (b)

10.4.7 Results

A plot of the actual closed-loop control system performance for a one-hour test with a desired temperature of 50°C and using the same gains as used in the MATLAB simulation is shown in Figure 10.42. The result shows a good agreement between the simulated and the real system behavior.

The measured temperature data is noisy. One source of noise is the coarse resolution of the temperature sensor. The measurement resolution can be improved by using an external supply voltage to act as a reference for the A/D. For example, if we have a connected a 2.5 V signal to the V_{ref+} line on the MCU, then the temperature measurement resolution would be 0.244°C instead of 0.488°C. It should

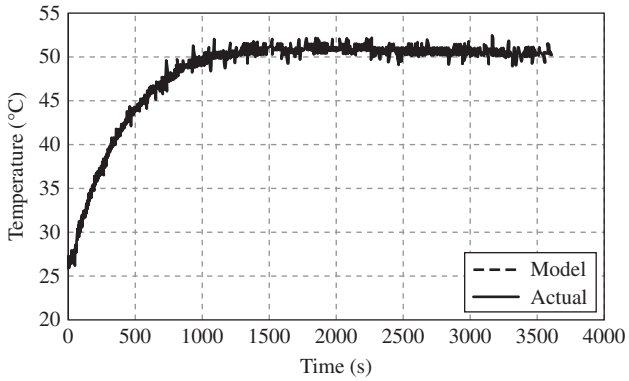


Figure 10.42

Experimental and simulated data for the plate setup

be noted that the heated plate system could be easily replaced by another system (such as a small motor-tachometer system). The main difference would be that the time constant of the motor-tachometer system would be much smaller than that of this heater system (tens of ms versus several minutes), which requires the use of a much smaller sampling interval (1 ms versus 1 second) and correspondingly shorter control times.

While this project demonstrates the use of the PC as a GUI for a control job implemented on an MCU, this project also could be implemented without the use of the VBE-developed GUI that is discussed here. The user can simply use a terminal program (such as *HyperTerminal* or *PuTTY*) to communicate with the MCU. Data stored in the MCU will then be sent to the terminal program instead of being written to a file as done here.

10.4.8 List of Parts Needed

A list of the main components needed to implement the heater control system is shown in Table 10.3.

Table 10.3

Main components for heater control system

Component	Manufacturer/Part #	Comments
Heater	McMaster-Carr #7945T52 DC Volt Flexible Silicone-Rubber Heat Strip Adhesive Backed, 1″ × 2″, 10 W	This heater has a 10 W output power value. A heater with a different power output will decrease/increase the time constant of the system
Temperature Sensor	National Semiconductor LM35C	
PIC 18 Development Board	Microchip PICDEM™ PIC18 Explorer Demonstration Board	Any PIC MCU with enough RAM capacity can be used
Transistor	IRFZ14 transistor	Any transistor with a power rating of 10 W or higher and a current limit of 1 A can be used

10.5 Chapter Summary

This chapter illustrated the integration of several of the topics covered in this book in the form of extended projects. The first project illustrated the use of a PIC MCU to perform open-loop control of the motion of a stepper-driven rotary stage that uses a photo interrupter as a homing sensor. A state-transition diagram was created to

handle the commands that were specified for the operation of the stage, and the control code was coded in C-language using the PIC-C compiler. The second project considered the closed-loop position control of a custom-built DC motor-driven machine that is used for dispensing paper. A VBE GUI was created to handle the user interface for the machine. The dynamics of the motor were identified using step-response tests, and a closed-loop PI controller was designed and simulated in MATLAB. The third project considered the temperature control of a small copper plate heated by a flexible heater. The dynamics of the heated plate were identified from open-loop step-response tests, and the developed model was used to design a PI controller. The PI-controller was implemented on a PIC-microcontroller with connection to a PC program created using VBE that acts as the user interface.

BIBLIOGRAPHY

[1] F. Harashima, M. Tomizuka, and T. Fukuda. "Mechatronics—What Is It, Why and How." *IEEE/ASME Trans. on Mechatronics*, Vol. 1, No. 1, 1996, pp. 1–4.

[2] D. Auslander, J. Ridgely, and J. Ringgenberg. *Control Software for Mechanical Systems: Object-Oriented Design in a Real-Time World.* Prentice Hall PTR, Upper Saddle River, NJ, 2002.

[3] R. Boylestad. *Introductory Circuit Analysis.* 3rd Edition, Charles E. Merrill Publishing Company, Columbus, OH, 1977.

[4] J. Irwin and R. Nelms. *Basic Engineering Circuit Analysis*. 9th Edition, Wiley, Hoboken, NJ, 2008.

[5] G. Rizzoni. *Principles and Applications of Electrical Engineering*. 5th Edition, McGraw-Hill, New York, NY, 2007.

[6] R. Coughlin and F. Driscoll. *Operational Amplifiers and Linear Integrated Circuits*. 6th Edition, Prentice Hall, 2000.

[7] P. Horowitz and W. Hill. *The Art of Electronics.* Cambridge University Press, Cambridge, UK, 1980.

[8] C. Roth. *Fundamentals of Logic Design*. 3rd Edition, West Publishing Co., St. Paul, MN, 1985.

[9] M. Rafiquzzaman. *Fundamentals of Digital Logic and Microcomputer Design*. 5th Edition, Wiley-Interscience, 2005.

[10] H-W. Huang and L. Chartrand. *PIC Microcontroller: An introduction to Software and Hardware Interfacing*. Delmar Learning, Clifton Park, NY, 2005.

[11] T. Wilmshurst. *Designing Embedded Systems with PIC Microcontrollers: Principles and Applications*. Newnes, Oxford, UK, 2007.

[12] B. Kernighan and D. Ritchie. *The* C *Programming Language*. 2nd Edition, Prentice Hall, 1988.

[13] K. Astrom and B. Wittenmark. *Computer-Controlled Systems: Theory and Design*. 3nd Edition, Prentice Hall, 1997.

[14] S. Smith. *The Scientist & Engineer's Guide to Digital Signal Processing.* California Technical Publishing, San Diego, CA, 1997.

[15] J. Seams. "R/2R Ladder Networks." Application Note IRC/AFD006, International Resistive Company, Inc., 1988.

[16] J. Axelson. *USB Complete: The Developer's Guide*. 4th Edition, Lakeview Research, Madison, WI, 2009.

[17] E. Hall. *Internet Core Protocols: The Definitive Guide*. O'Reilly Media, Cambridge, MA, 2000.

[18] C. Kozierok. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference.* No Starch Press, San Francisco, CA, 2005.

[19] S. Lyshevski. "Mechatronic curriculum—retrospect and prospect." *Mechatronics*, Vol. 12, pp. 195–205, 2002.

[20] P. Laplante. *Real-Time Systems Design and Analysis*. 3rd Edition, Wiley, 2004.

[21] D. Auslander, A. Huang, and M. Lemkin. "A Design and Implementation Methodology for Real Time Control of Mechanical Systems." *Mechatronics*, Vol. 5, No. 7, pp. 811–832, 1995.

[22] W. Cedeño and P. Laplante. "An Overview of Real-time Operating Systems." *Journal of the Association for Laboratory Automation*, Vol. 12, No. 1, pp. 40–45, 2007.

[23] E. Lamie. *Real-Time Embedded Multithreading Using ThreadX and MIPS.* Newnes, 2009.

[24] S. Smith. *MATLAB: Advanced GUI Development*. Dog Ear Publishing, Indianapolis, IN, 2006.

[25] E. Petroutsos. *Mastering Visual Basic 2010*, Wiley, 2010.

[26] A. Boehm, *Murach's Visual Basic 2010*. Mike Murach & Associates, Fresno, CA, 2010.

[27] T. Beckwith, R. Marangoni, and J. Lienhard V. *Mechanical Measurements.* 6th Edition, Prentice Hall, 2007.

[28] *The Measurement, Instrumentation and Sensors Handbook*. J. Webster, Editor-in-Chief, CRC Press, Boca Raton, FL, 1999.

[29] J. Holman. *Experimental Methods for Engineers.* 7th edition, McGraw-Hill, 2001.

[30] S. Rao. *Mechanical Vibrations*. 3rd Edition, Addison-Wesley, Reading, MA, 1995.

[31] A. Hughes. *Electric Motors and Drives: Fundamentals, Types and Applications.* 3rd Edition, Newnes, 2006.

[32] I. Gottlieb. *Electric Motors and Control Techniques*. 2nd Edition, TAB Books, McGraw-Hill, 2004.

[33] T. Kenjo. *Electric Motors and their Controls: An Introduction*. Oxford University Press, Oxford, UK, 1991.

[34] C. de Silva. *Sensors and Actuators: Control System Instrumentation*. CRC Press, 2007.

[35] W. Brown. "Brushless DC Motor Control Made Easy." Application Note AN857, Microchip Technology, Inc., 2002.

[36] L. Elevich. "3-Phase BLDC Motor Control with Hall Sensors Using 56800/E Digital Signal Controllers." Application Note AN1916, Rev. 2.0, Freescale Semiconductor, 2005.

[37] K. Ogata. *Modern Control Engineering*. 4th Edition, Prentice Hall, 2002.

[38] W. Palm III. *System Dynamics*. McGraw-Hill, 2005.

[39] *The Control Handbook*. W. Levine, Editor, Chapter 10, CRC Press, 1996.

[40] G. Franklin, J. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*. 4th Edition, Prentice Hall, 2002.

[41] A. Preumont. *Vibration Control of Active Structures*. 2nd Edition, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[42] W. Palm III. *Introduction to MATLAB for Engineers*. 3rd Edition, McGraw-Hill, New York, NY, 2011.

[43] A. Gilat. *MATLAB: An Introduction with Applications*. 3rd Edition, Wiley, 2008.

CHAPTER 2

P2.5 $I_1 = 1.25$ mA, $I_2 = I_3 = 0.625$ mA

P2.6 $I_1 = 2.31$ mA, $I_2 = 1.54$ mA, $I_3 = 0.77$ mA, $I_4 = I_5 = 0.385$ mA

P2.8 (a) $R_{TH} = 17\ \Omega$, $V_{TH} = 10$ V

(b) $R_{TH} = 3.33\ \Omega$, $V_{TH} = 3.33$ V

P2.10 0.376

P2.11 $R = 429\ \Omega$, $X = 343\ \Omega$

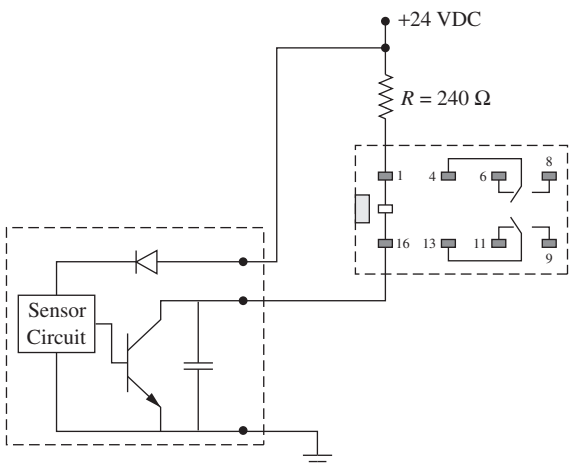
P2.14 Circuit will not operate as proposed

CHAPTER 3

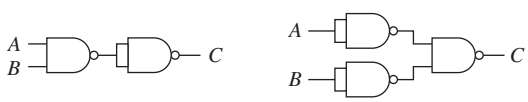
P3.4 $I_C = 2800\ \mu$A, $V_{CE} = 1$ V

P3.5 $V_1 = 4.14$ V, $V_2 = 3.84$ V

P3.7



P3.9



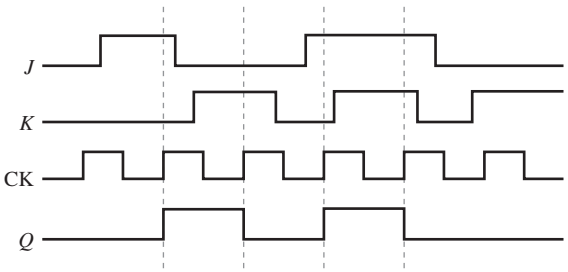
(a) (b)

P3.10 (a) $Q = B + \overline{A} \cdot C$

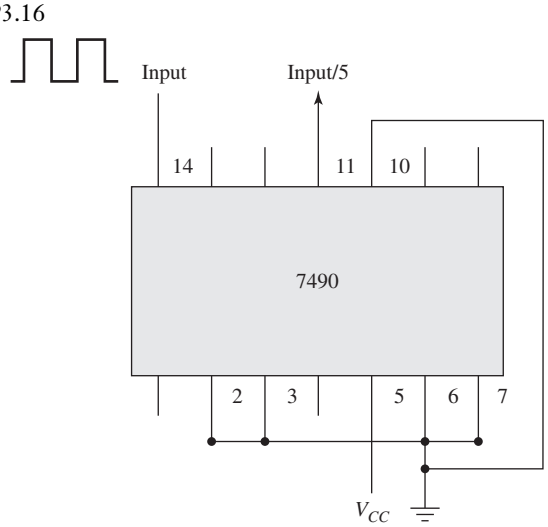
(b) $Q = A \cdot (B + C)$

P3.12 $Q = A \cdot B + A \cdot C + B \cdot C$

P3.15



P3.16



CHAPTER 4

P4.2 (a) 0xFF (b) 0xD5 (c) 0x87

P4.3 (a) 1111 (b) 01101100 (c) 0111

P4.4 (a) $0.078125 = 1 \times 2^{-4} \times 1.25$

(b) $-0.5 = -1 \times 2^{-1} \times 1$

(c) $10.5 = 1 \times 2^3 \times 1.3125$

P4.8 Maximum counting interval is 0.0524 seconds

P4.9 $PR2 = 124$, $t2pres = 4$, $Value = 125$

P4.11 Use a prescale factor of 8 which gives an overflow period of 1.024 ms which is 2.4% higher than the desired interrupt period

P4.15 (a) Complement: 0xD5, Negate: 0xD6

(b) Complement: 0x82, Negate: 0x83

CHAPTER 5

P5.2 (a) 204 (b) 511 (c) 1023

P5.3 (a) 307 (b) 511 (c) 921

P5.4 (a) 7.6294 to 7.6370 °C (b) +/− 3.8147 e-3 °C

P5.5 0.96875 V_R

P5.6 5.208 seconds

CHAPTER 6

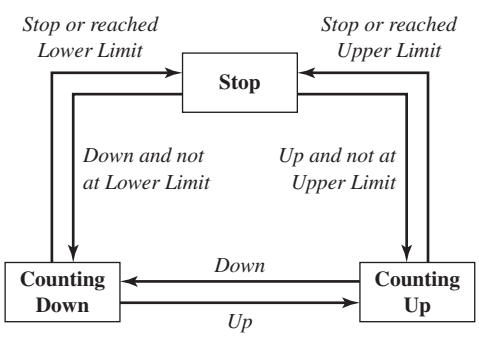
P6.1 Timer resolution = 1 μsec, Maximum counting interval = 65.536 msec

P6.3 (partial answer)

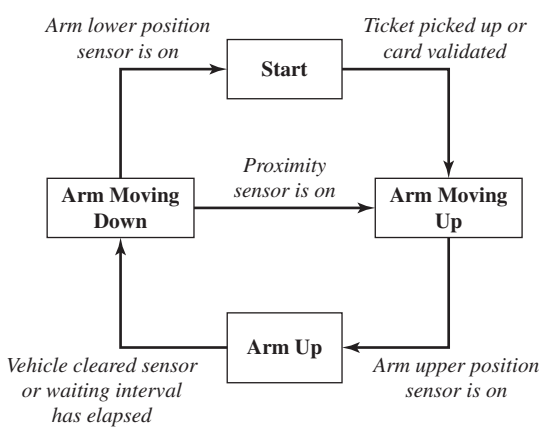
(a) Absolute (b) Absolute

(c) Interval (d) Interval

P6.5



P6.7



CHAPTER 7

P7.2 197.3 degrees, 195.2 degrees

P7.3 0.008 degrees

P7.6 20.8 microstrain, 1.66 kN

P7.7 255 °C

P7.9 (a) −1.63 g (b) 2.88 g

P7.10 166 Ω

CHAPTER 8

P8.4 2400 rpm, 0.190 hp

P8.5 374 lb-in

P8.7 0.125 inch/sec

P8.11 30.3 A

CHAPTER 9

P9.2 $G_{cl} = \dfrac{\theta(s)}{\theta_d(s)} = \dfrac{10s + 5}{s^3 + 10s + 5}$

P9.3 $G_{cl} = \dfrac{\theta(s)}{D(s)} = \dfrac{5s}{s^3 + 10s + 5}$

Steady state error = 0

P9.5 $\dfrac{\theta(s)}{\theta_d(s)} = \dfrac{K_P + K_d s}{Js^2 + (B + K_d)s + K_P}$

$\dfrac{\theta(s)}{D(s)} = \dfrac{1}{Js^2 + (B + K_d)s + K_P}$

P9.6 $K_P = 4$ Steady state error = 0.2

P9.9 (a) $A = \begin{bmatrix} 0 & 1 \\ 0 & -B/J \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 1/J \end{bmatrix}$, $C = [1 \quad 0]$, $D = [0]$, $B = 0.5$, $J = 0.7$

(b) $K = [23.8 \quad 6.50]$

APPENDIX A

Visual Basic Express

A.1 Introduction

Visual Basic (VB) is a high-level programming language that is used by many programmers worldwide to produce professional code for a multitude of applications. In addition, Visual Basic provides an easy way of creating powerful window-based user interfaces. Microsoft introduced the Visual Basic programming language in the early 1990s, starting with VB1 as the first version. Five updated versions were introduced in the 1990s with VB6 being the latest in that series before the VB.NET was introduced in 2002. Visual Basic Express (VBE) is a simplified version of the VB.NET edition of Visual Basic, and it was first released in late 2005 as VB 2005 Express. This was followed by VB 2008 Express in 2008 and VB 2010 Express in 2010. VBE was created specifically to meet the needs of students, hobbyists, and novice programmers who do not need all of the intricacies and details of the .NET version. Also VBE is available to be downloaded free of charge.

VBE is designed around the .NET Framework, which provides tools for security, deployment, memory management, and versioning. An important component of the .NET Framework is the Base Class Library, which has a large number of programming components that programmers can combine with their code to produce applications. The .NET Framework enables interoperability between objects, so an object created in a VBE can be used in another .NET Framework compatible language.

This appendix gives a general overview of VBE with code examples that use VB 2010 Express. It is not intended to give an in-depth coverage of the language and its development environment, but to simply highlight some of the basic concepts. Readers without a previous knowledge of Visual Basic are encouraged to read any of the many books that cover the language in more detail.

To develop code in VBE, first you need to create a project that consists of set files grouped together. In VBE, you can create different types of projects—the most common being Windows Forms application and Console application. A Windows Forms application will create a program that has a Windows user interface similar to that used in programs such as MS Word. On the other hand, a Console application creates an MS-DOS type interface, similar to the interface that programs had in the pre-Windows age. Console applications are easier to create, since they are not event driven (as are Windows applications).

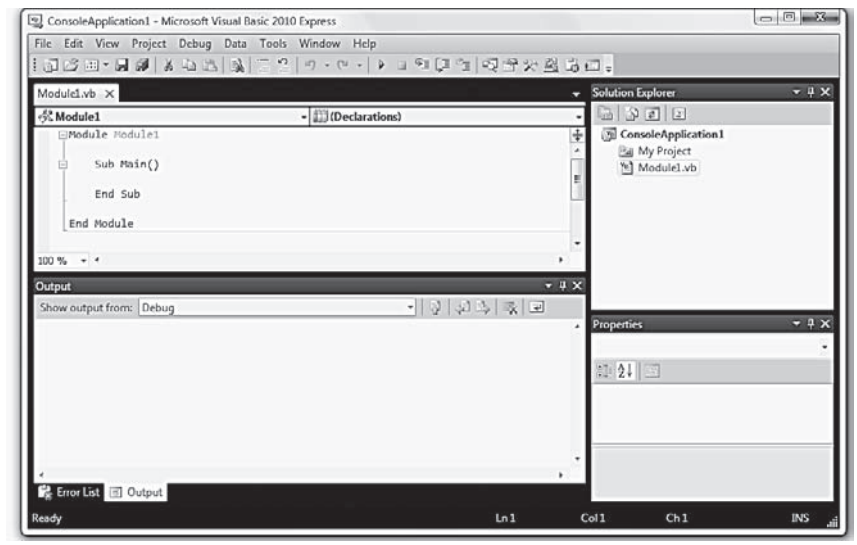
A.2 Console Application

We will first demonstrate how to create a simple console application before we talk about Windows applications. Our simple console application will ask the user to type his/her name and then it will display a greeting message. To start on this application, call up VBE, then select **New Project** from the **File** drop-menu bar.

Select **Console Application** from the **New Project** window that shows up, and then click **OK**. On the screen, you will see a display similar to that shown in Figure A.1. For this console application, VBE has created a Visual Basic file called *module1.vb*. Module type files can only contain code that does not control a window. This module has one subroutine called *Main()*, which we will edit to write the code for the greeting message. Notice on the right hand of the screen the *Solution Explorer* window, which shows the files that make up this project. In addition to *module1.vb*, this project has a *My Project* folder which contains information about this application. The *Solution Explorer* window can be activated from the **View** menu.

Figure A.1

Console application in VBE



The edited *Main()* routine is shown in Figure A.2. It has three lines of code consisting of the *Console.Write()* and *Console.ReadLine()* functions. To compile and run this console application, select **Build ConsoleApplication1** from the **Debug** menu. Then select **Start Debugging** from the **Debug** menu.

Figure A.2

Code listing for consol application

```
Sub Main ()
        Console.Write ("Type your name ")
        Console.Write ("Hello " + Console.ReadLine ())
        Console.ReadLine ()
End Sub
```

The first *Console.Write()* statement causes the text enclosed in parentheses to be displayed in the console when this application is run. When the user types a name and hits the *Enter* key, the message "Hello," followed by the typed name, is displayed. Notice that the *Console.ReadLine()* statement waits for the user to hit the *Enter* key before the greeting message is printed. The third line in the code was added to make the console window stay open until the user hit the *Enter* key the second time. (Try to run this application with the third line removed and see what happens.)

A.3 Windows Forms Applications

Unlike console applications, Windows Forms applications are event driven. This means the operation of a Windows program is not preset but depends on which events happen as the program is executed. For example, events are created by clicking on a command box, moving the mouse, or activating a form. To develop a Windows Forms application, the user places controls on a Windows Form, and then writes code to manage events from these controls in another file. To illustrate a Windows Forms application, we will create a simple application that displays a message in a textbox when a user clicks on a button. To start on this application, call up VBE, then select **New Project** from the **File** drop-menu bar. Select **Windows Forms Application** from the **New Project** window that shows up and then click **OK**. On the screen, you will see a display similar to that shown in Figure A.3.

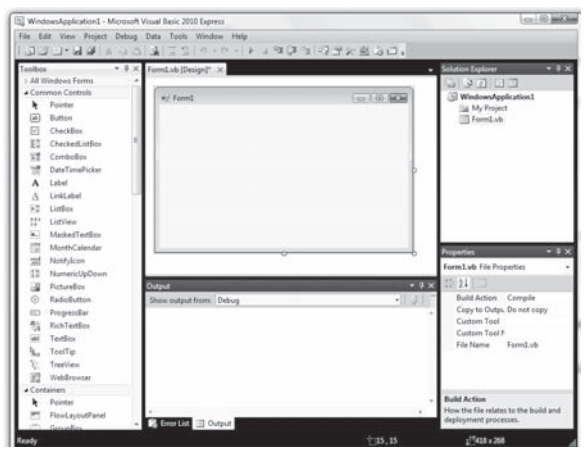


Figure A.3

Design form in a Windows Forms application

VBE has created a blank window form labeled *Form1.vb[Design]*. VBE calls this form the Design form since in this form the user designs the program interface. Next we will add two controls to this form. In the **Toolbox** window shown to the left of the form, we click on the **Button** control and drag and place it on the form. We repeat this process to add a **TextBox** control. After these operations, our form looks similar to that shown in Figure A.4. If you build and run this application, nothing will happen when you click on the button control, since we have not added any code to process this clicking event. What VBE has done so far is to set up a framework for the application to handle events, but the programmer has to provide the details of what needs to be done with these events.

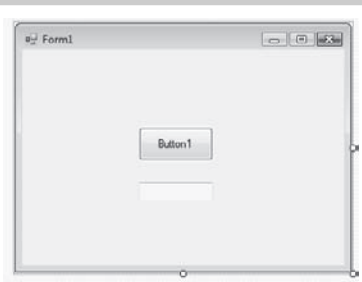


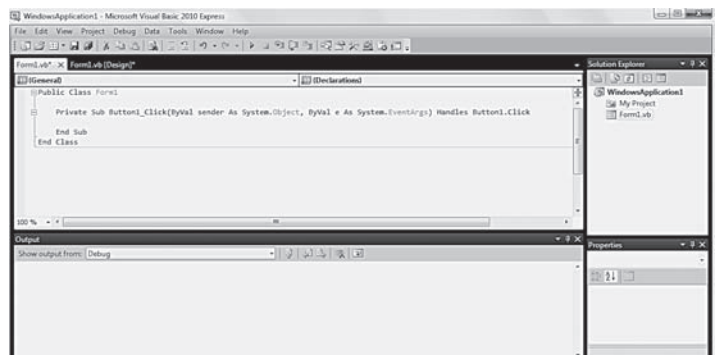
Figure A.4

Windows form with two controls

Now close the application, and go back to the development environment. Double click on the button control. We notice that the *Form1.vb[Design]* has disappeared, and a window named *Form1.vb* has shown up, as shown in Figure A.5.

Figure A.5

Code window for *Form1.vb*



VBE is an object-oriented language. As such, a component (like a Windows form, for instance) is an object (referred to as a class in code), and the code belonging to a particular form is grouped in the class belonging to that object. In our example, the class has a single procedure called *Button1_Click*. This procedure is of the **Private type**, which means that it can be accessed only from code within this class. The *Button1_Click* routine is the code that handles the event associated with clicking on the button (*Button1.Click* event). This routine has two arguments: *sender* and *e*. *Sender* is a variable that indicates which object is associated with this routine, which in this case is *Button1*, while *e* has the details of the mouse click event that caused the routine to be called.

Figure A.6

Code for *Button1_click*

```
TextBox1.Text = "Hello"
```

The button control has many other **events** (such as the mouse hovering over the control, called *MouseHover* event or the mouse leaving the button, called *MouseLeave* event). We will add one line of code to the *Button1_click* routine, as shown in Figure A.6, so that it displays a message in the textbox when one clicks the button. Now build and run this application. Observe what happens when you click on the button.

Let us now change the label on the control button to one that is more descriptive. The button label is one of many **properties** of the button object. Other properties include color, size, location, and text font among others. Go back to the *Form1.vb [Design]* window, click on the *Button1* object, then replace '*Button1*' in the text property window with '*Run*'. Notice that the button label is now '*Run*'. Experiment with changing other properties such as the location of the button and the font.

In developing applications with many controls, one should make use of the 'visible' and 'enabled' properties of these controls to help the application user in navigating through the available controls. Setting the '**visible**' property of a control to false makes the control not show up at run-time, while setting the '**enabled**' property to false cause the control to show up at run-time, but it is inactive.

Notice that any control element in VBE or any object for that matter has **properties**, **methods**, and **events**. We have already talked about properties and events. **Methods** are actions that the object or control can perform in the form of sub-procedures and functions. For example, for the button control, the methods include *Focus* (which sets input focus to the control), and *GetType* (which gets the type of the current instance of the object).

VBE has a very useful feature that helps in debugging and tracing console and Windows applications. It is available under the **Debug** menu. Instead of using the

Start Debugging command under the **Debug** menu to run the application in the IDE, one can use the ***Step Into*** command. With the *Step Into* command, VBE executes one line of code at a time, stopping at the start of the next executable statement. With the code stopped at each statement, one can check program flow and use the mouse to hover over the variables in each statement to see their values. To continue program execution, the user simply needs to press F8 or the *Step Into* command again.

If the user wants to make the program stop at particular line instead of stepping through all of the lines in the code, then one can use the ***Toggle Breakpoint*** command in the **Debug** menu to mark lines where the code should be stopped. When the program is run using the *Start Debugging* command, the program will stop at the beginning of each of these marked lines. Each breakpoint can be removed by simply clicking on the toggle breakpoint mark on that line, or all toggle breakpoints can be removed by clicking on the ***Delete All Breakpoints*** command.

A.4 Files and Directory Structure

Before we discuss some details of the VBE programming language, let us have a look at the files and directory structure (see Figure A.7 for illustration) that are created with a project in VBE. When a project is saved in VBE, a folder is created to store the files related to the project. The folder has three subfolders labeled *bin*, *My Project*, and *obj*. The folder also has some other files, including a VB project file (with extension *.vbproj) and a file with extension **.sln* (or solution file) that also has the same name as the parent folder.

To edit the project, you simply click on the solution file. The ***bin*** folder holds binary files, while the ***obj*** folder holds compiled files. The *bin* folder is where the application looks for the files it needs when running. The files that are needed are copied from the *obj* folder to the *bin* folder before the application is run. Both the *bin* and the *obj* folders have two folders each, labeled ***Debug*** and ***Release***, corresponding to the debug or release configuration of the code, respectively. The debug configuration stores extra data that enables the application to interface with the debugger when the application is run. The release configuration does not store this extra data; therefore, it is smaller and runs faster than the debug version but does not offer linking to a debugger. Before the application is built using the *Build* command, the release folder will be empty. The *My Project* folder holds the settings and resource files that are used by the application (such as external libraries).

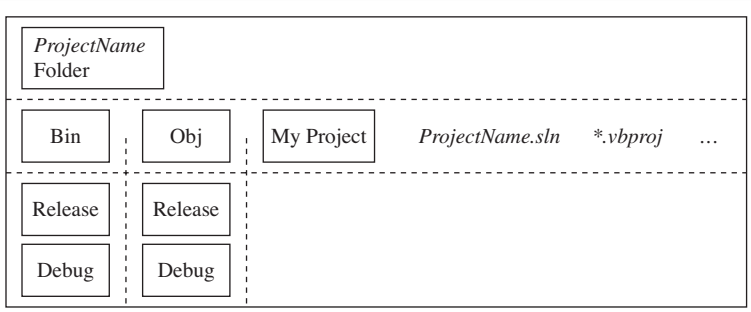


Figure A.7

Folder and directory structure for the Windows project in VBE

Table A.1 lists some of the common file type extensions and their usage that are present when a project is created.

Table A.1 Common file type extensions in VBE

File Extension	File Usage
**.Designer.vb*	The file extension for the designer view of a form.
**.exe*	The executable file. The application can be run by clicking on this file.
**.myapp*	Application configuration file.
**.pdb*	A program database (PDB) file that holds debugging and state information. It is created when the application is built in *Debug* mode.
**.resx*	A file to store .Net application resources.
**.settings*	A file for user-settings information.
**.sln*	The solution file. A solution can have one or more projects.
**.vb*	A file for storing the code associated with a Form, class, module, assembly information, or user settings.
**.vbproj*	Visual basic project file. Clicking on this file causes the project to be displayed in the VBE integrated development environment.
**.vhost.exe*	A hosting process file that is used by Visual Studio 2010. The file should not run directly.
**.xml*	A file that holds data in extensible markup language format.

A.5 VARIABLES

As with any programming language, the user should be aware of the different types of variables that can be used in the code. A variable is a name or a character that can store a value or any information that is used in the program. VBE supports many variable types ranging in size from *Byte* (1 byte) to *Decimal* (16 bytes). Some VBE variables are designed to store integer data types (such as *Integer*), others can store floating-point data types (such as *Single* and *Double*), and some can store any collection of numeric and non-numeric characters (such as *String*). Table A.2 lists some of the common variables in VBE, their respective size, and the range of values

Table A.2 Common variables in VBE

Variable Type	Size	Range of Values
Byte	1 byte	Integer numbers from 0 to 255 (unsigned).
Char	2 bytes	Integer numbers from 0 to 65535 (unsigned).
Short	2 bytes	Integer numbers from −32768 to 32767.
Integer	4 bytes	Integer numbers from ~ −2.14e9 to ~2.14e9.
Long	8 bytes	Integer numbers from ~ −9.2e18 to ~9.2e18.
Single	4 bytes	Single-precision floating-point numbers from ~ +/ −1.4e–45 to ~ +/−3.4e38.
Double	8 bytes	Double-precision floating-point numbers from ~ +/−4.94e-324 to ~ +/−1.79e308.
String	Varies	Any collection of characters.
Decimal	16 bytes	Floating-point numbers 0 to ~ +/−7.9e28 with no decimal, or 0 to ~ +/−7.92 with 28 places to the right of the decimal.
*Object**	4–8 bytes	Any type can be stored. Object is the default type if type is not explicitly declared.

***Depends on platform 32-bit or 64-bit.**

that they can store. Notice that *Short*, *Integer*, and *Long* types can be restricted to handle only unsigned integers if they are replaced by *UShort*, *UInteger*, and *ULong* respectively. Similarly, the *Byte* type can be made to handle signed integers if replaced by *SByte*.

Any collection of characters can be assigned to a **string** variable by enclosing the characters between double quotes and using the equal sign operator for assignment, such as

```
StringVar="Num123"
```

where *StringVar* is a variable of type string. VBE has many built-in functions that operate on strings. These include the *Len* function to find the length of a string (i.e., the number of characters in the string), the *Mid* function to get a substring from a given spring, and the *Format* function to format a string.

To use a variable in VBE, first you need to declare the variable through a **declaration statement** (for example, *Dim var1 As Integer*). The declaration statement should precede any location in the program where that variable is used. The declaration statement defines what kind of data that variable should hold (such as *Integer* for *var1* above). In addition, the location of the declaration statement defines the scope of the variable. The **scope** refers to the area in the program in which the variable can be accessed. There are three different scope levels. The most restrictive is the **block-level** scope, which is the case when the variable is declared inside a defined block of code (such as a for-loop or a do-while statement). Such a variable is only defined within the block of code in which it is declared, and cannot be accessed outside the block. The second level of scope is the **procedure-level scope**, which applies to variables declared inside a function or sub-procedure but outside of a block of code. These variables are known only within that procedure in all of the statements that follow their declaration statements. The third level of access is the **file-level scope**, which applies to a variable declared outside any function or sub-procedure. In this case, the variable is accessible by all of the procedures inside the file in which it is declared.

Furthermore, by including certain qualifiers (such as *public* or *private*) in the declaration statement, one can modify the access level for that variable. For example, a *public* variable (default setting) declared outside any function or sub-procedure is accessible from other files or modules in the project, while a *private* variable is only accessed from code within the file in which is declared. Note that, for cases where a variable needs to be accessed in several files including a form file, the variable should not be declared in the form file but in the other files (such as a module file). Moreover, the declaration statement location defines the **lifetime** of the variable, which defines how long the variable should continue to exist. A variable declared outside of any function or sub-procedure continues to exist as long as the program is running. On the other hand, a variable declared inside a function or a sub-procedure exists only when that procedure is running. The use of the '**static**' qualifier in a declaration statement causes the variable to have an infinite lifetime, regardless of where it is declared. When the data that needs to be stored remains constant during the program execution, VBE allows the use of the *Const* data type to store data that does not change.

In addition to regular variables, VBE supports the use of **single** and **multi-dimensional arrays**. For example, to declare an array of integers, we write (*Dim array1(10) As Integer*). Because the first element of an array is the zero element (i.e., *array1(0)*), the previous statement has declared 11 array elements. Similarly, the statement (*Dim array2(10,1) As Integer*) declares a two-dimensional array named *array2* that has 22 elements (11 rows by 2 columns).

A.6 Operators

VBE supports different groups of operators. These include mathematical, logical, and relational. The **mathematical operators** include addition (+), subtraction (−), multiplication (*), floating-point result division (/), integer result division (\), and exponentiation (^).

Mathematical functions (such as sine, cosine, and logarithm) are also available but are called as a component of the *Math* class (through the use of the dot operator). For example, the sine function is called as *Math.Sin()* and the natural logarithm function is called as *Math.Log()*.

Logical operators compare Boolean expressions and return a value of true or false (called Boolean). These operators include *And*, *Or*, *AndAlso*, *OrElse*, *Xor*, *and Not*. The first five operators are binary in the sense that they operate on two expressions, while the last operator is unary because it operates on only one expression. Table A.3 gives the truth table for these logical operators. Note that the *AndAlso* and *OrElse* operators have the same truth table as the *And* and *Or* operators, respectively, but these operators are **short circuiting**. This means that the compiled code can bypass the evaluation of the second expression depending on the result of the first expression. This is done to improve performance. For example, the second expression for the *AndAlso* operator is not evaluated if the first expression is false, since in this case, the first expression being false determines the result of the operation (which is false). Similarly, the second expression in the *OrElse* operation is not evaluated if the first expression evaluates to true.

Table A.3

Truth table for VBE logical operators

X	Y	X *And* Y	X *Or* Y	X *AndAlso* Y	X *OrElse* Y	X *Xor* Y	Not *X*
F	F	F	F	F	F	F	T
F	T	F	T	F	T	T	T
T	F	F	T	F	T	T	F
T	T	T	T	T	T	F	F

Unlike the C-language, VBE does not have dedicated **bitwise logical operators**, but the *And*, *Or*, *Xor*, and *Not* operators also act as bitwise logical operators if the operands happen to be of an integer type. The bitwise operators operate on identically positioned bits in each of the two numeric expressions. For example, 9 *AND* 3 gives 1, since both numbers have a 1 only in their zero bit location. Similarly, 9 *Or* 3 gives 11, since the zero, first, and third bit positions have 1 in at least one of the two expressions.

Relational operators compare two expression to decide if they are equal (= operator), not equal (<> operator), less than (< operator), less than or equal to (<= operator), greater than (> operator), and greater than or equal to (>= operator). The result of the comparison is a true or false value.

Figure A.8

Illustration of the "For-Loop"

```
Dim i, sum As Integer

sum = 0

For i = 1 To 10
    sum = sum + i
Next
```

A.7 Looping and Conditional Statements

VBE provides the means to repeat the execution of a group of statements through the 'For-loop' or any variation of the 'While' statement. Figure A.8 shows an example of code that uses a **For-loop** to sum all the integer numbers from 1 to 10. Notice that when this For-loop starts, the index *i* is set to 1. This index is incremented by

1 (which is the default setting but could be any positive or negative increment value if explicitly specified by using the *step* statement) after each run through the loop. Before the statements in the body of the loop get executed again, the index value is always checked to see that it does not exceed the upper limit for the index (10 in this case). Once the check condition is no longer valid, the statement in the For-loop is skipped, and the next statement below the For-loop gets executed. After the execution of this For-loop the sum variable will have a value of 55, and the index *i* will have a value of 11.

The **Do-While** statement in VBE also allows for repeated execution. A basic form of the statement is shown in Figure A.9. In this example, the statements inside the Do-While-loop get executed as long as the test condition ($i <= 10$) is true. Once the test condition is no longer true, the statements in the body of the Do-While-loop are skipped, and the execution moves to the statements that follow the Do-While-loop. Similar to the For-loop code, the variable *sum* will have a value of 55, and *i* will have a value of 11 after the execution of this Do-While-loop. Note that VBE initializes the variables *i* and *sum* to zero when they are declared.

Figure A.9

Illustration of the Do-While statement

```
Dim i, sum As Integer

Do While (i <= 10)
        sum = sum + i
        i = i + 1
Loop
```

Similar to all programming languages, VBE supports the **If-Then** statement that offers conditional execution. In its basic form, the If-Then statement checks if the conditional statement is true. If this is the case, then the statements in the body of the If-Then statement are executed; otherwise, they are skipped. Figure A.10 shows such an example which uses both relational operators (less than) and logical operators (*And* in this case) in the conditional statement. In this example, both relational statements are true, and thus the logical combination of these statements through the *And* logical operator evaluates to be true. Hence, the conditional part of the If-Then is true, and the variable *d* will have a value of 50 instead of 0 after the execution of this statement.

Figure A.10

Illustration of the If-Then statement

```
Dim a, b, c, d As Integer

a = 10
b = 20
c = 30
d = 0

If (a < b) And (b < c) Then
    d = 50
End If
```

Now let us add an *Else* part to the *If-Then* statement, which applies if the conditional part evaluates to be false. Figure A.11 shows a variation of the case considered in Figure A.10. Since the variable *a* is greater than *b* here, the conditional statement evaluates to false, and the variable *d* is assigned a value of 60.

A more elaborate form of the *If-Then* statement is shown in Figure A.12. This form uses multiple *ElseIf* statements followed by a single *Else* statement. The same example considered before is expanded here to show this form.

Note that in this example if the condition in the *If* part of the statement evaluates to false, then the condition in the first *ElseIf* statement is evaluated. If that condition evaluates to false again, then the condition in the next *ElseIf* statement is evaluated. The statement following the *Else* statement is only evaluated if none of the other conditional statements evaluates to be true. While there is no limit on the number of *ElseIf* conditions, only one *Else* condition can be included in the If-Then

```
Dim a, b, c, d As Integer

a = 25
b = 20
c = 30
d = 0

If (a < b) And (b < c) Then
    d = 50
Else
    d = 60
End If
```

Figure A.11

If-Then statement with an Else part

Figure A.12

If-Then statement with multiple *ElseIf* statements

```
Dim a, b, c, d As Integer
a = 10
b = 20
c = 30
d = 0

If a < b And b < c Then
    d = 50
ElseIf a < b Then
    d = 70
ElseIf b < c Then
    d = 80
Else
    d = 60
End If
```

statement. Note that if any of the conditional statements evaluates to true, then all of the remaining conditional statements are not evaluated.

A more elegant method of branching is provided by the *Select Case* statement, an example of which is shown in Figure A.13. An expression, which should evaluate to one of the elementary data types (such as Boolean, integer, or string), is placed in the *Select Case* part of the statement followed by a number of possible branches—each corresponding to a different possible value of the evaluated expression.

Figure A.13

Select case statement

```
Select Case (a)
    Case 1
        b = 2
    Case 4
        b = 3
    Case 10
        b = 50
End Select
```

Similar to the If-Then statement, a *Case-Else* statement can be included which gets executed if none of the other listed cases matches the evaluated expression. Furthermore, the *Select Case* statement allows the use of a range of values or multiple expressions in each *Case* clause. For example, one can use the following code:

```
Case 5 To 8, 9, 20 To 25
```

A.8 Functions and Sub-Procedures

To provide modularity and ease of program flow, a program should be structured as a collection of procedures instead of a single procedure with a large number of statements. **Sub-procedures** and **functions** are code elements that are executed by calling their name. Arguments can be passed to both subs and functions in the call statement. The difference between a function and a sub-procedure is the way in which values are returned to the calling program. A sub-procedure cannot directly return a value, but it can include a return value in its argument list. On the other hand, a function can directly return a value.

To illustrate sub-procedures and functions, let us consider a simple example of writing a routine to add two numbers. We will implement this using both sub-procedures and functions. Figure A.14 shows the code listing for both methods, while Figure A.15 shows the calling statements that are needed to execute these procedures. Note that providing a code listing for a sub-procedure or a function does not cause that routine to execute. The routine has to be explicitly called in the calling statements, as shown in Figure A.15, to execute. In the code listing shown in Figure A.14, the *'Sub Procedure Method* is a comment line. It is a good practice to add **comment statements** to code to explain what the code is doing. Note that the comments statements are not translated by the compiler to machine code, and thus, they do not affect the compiled size of the code.

```
'Sub Procedure Method

Sub Addsub (ByVal v1 As Integer, ByVal v2 As Integer, ByRef v3 As Integer)

        v3 = v1 + v2

End Sub

'Function Method

Function Addfun (ByVal v1 As Integer, ByVal v2 As Integer) As Integer
        addfun = v1 + v2
End Function
```

Figure A.14

Illustration of a sub-procedure and a function

```
Dim x1, x2, x3 As Integer
x1 = 110
x2 = 322

Call Addsub (x1, x2, x3) 'Addition using a sub procedure

x3 = Addfun (x1, x2) 'Addition using a function
```

Figure A.15

Calling statements for sub-procedure and function

Note that the format of calling a sub-procedure through the use of the *Call* statement is optional, and it can be called by just using its name. Since a sub-procedure does not directly return a value to the calling function, the result of the addition of variables *v1* and *v2* in *Addsub* was assigned to the third argument, *v3*. This was not the case for the function method *Addfun*, since a function can return a value by assigning the result to the name of the function (as shown in this example) or by the use of the keyword *return* followed by the value that needs to be returned. Notice the use of the keyword *ByVal* in the code listings for these routines for all variables except *v3*. This keyword determines how the argument variables are passed to the called procedure. There are two ways to pass arguments to a procedure: *ByVal* and *ByRef*. The *ByVal* method is the default in VBE. ***ByVal*** means that a copy of the variable is passed to the called procedure. The procedure can not alter the original value of the passed variable. In the ***ByRef*** method, the called procedure can modify the value of the passed variable, since a reference to the location in memory where the variable is stored is passed to the routine. By analogy, you can think of *ByVal* versus *ByRef* as sending a copy versus an original document to a person. Even if the copy gets destroyed, nothing happens to the original document if you sent only the copy.

To return the result of the addition performed in the sub-procedure, *v3* was passed as *ByRef*. In this way, the *Addsub* procedure is able to place the addition result in the memory location corresponding to variable *x3* (the variable that was passed to the sub-procedure). If we change the *ByRef* designation for *v3* to *ByVal*, *x3* will have a value of zero after the sub-procedure finishes execution. As an alternative to passing a variable to get the result of an operation in a sub-procedure, we could assign the result to a global variable that is defined elsewhere in the file.

In certain situations, the passed arguments could be array elements or even represent a whole array. Passing a single array element to a procedure is identical to passing any other variable. The particular array element, such as *data(2)*, is simply listed as such in the argument list in the calling statement. If instead we want to pass the whole array to a procedure, then in the calling statement the array is

Figure A.16

Passing an array to a procedure

```
Dim data (10) As Integer 'Declaring an array with 11 elements (0-10)

Call sub1 (data(2)) 'Passing a single array element to sub procedure 1
Call sub2 (data)        'Passing the entire array to sub procedure 2

Sub sub1 (ByVal v1 As Integer) 'Sub listing with single array element
      …
End Sub
Sub sub2 (ByVal v1() As Integer) 'Sub listing with entire array
      …
End Sub
```

passed as the array name alone (for example *data*) with no parenthesis following the array name. In the procedure listing, the array is listed as a variable name with the *( )* following it, but the size of the array is not included within the parenthesis. This is illustrated in Figure A.16.

VBE allows **procedure overloading**, which means a given function or sub can be called in many different ways but using the same procedure name. These ways include a different number of arguments, different types of arguments, or a different ordering of the arguments in the calling statement. To implement procedure overloading, the code listing for each variation of the procedure is preceded with the ***Overloads*** keyword. For example, Suppose we have a sub-procedure named *sub1*, but we want to have two variations of calling it—one passing two arguments to it and the other passing just a single argument. Then the listing for these two variations is shown in Figure A.17.

Figure A.17

Illustration of procedure overloading

```
Public Overloads Sub sub1 (ByVal x1 As Integer, ByVal x2 As Integer)
    'Add needed code below
End Sub

Public Overloads Sub sub1 (ByVal x1 As Integer)
    'Add needed code below
End Sub
```

VBE has a large number of built-in functions for specific purposes. We already have talked about the string manipulation functions when we discussed variables. Other applications include **functions to perform data conversion** (i.e., to convert from one data type to another) such as the *CInt* function, which converts an expression to an integer data type, and mathematical functions.

Having talked about functions and sub-procedures, let us now have a look at how VBE organizes the different code elements in a Windows Forms Application. Assume we have developed an application that has a single button and a *TextBox* control similar to that shown in Figure A.4. Assume also we have developed in our application two simple functions, called *function1* and *function2*, that are called from the *Button1_Click* routine (the code that handles the event associated with clicking on the Button control). The code listing for these routines is shown in Figure A.18.

Notice the two drop-down lists just above the *Public Class Form1* line in Figure A.18. The left drop-down list has the entry *Form1* shown, while the right drop-down list has the entry (*Declarations*) shown. The left drop-down list lists all of the objects that are used in this application, while the right drop-down list shows all of the code elements (or procedures) associated with the selected object from the left list. For example, if we expand the right drop-down list, we see the two functions

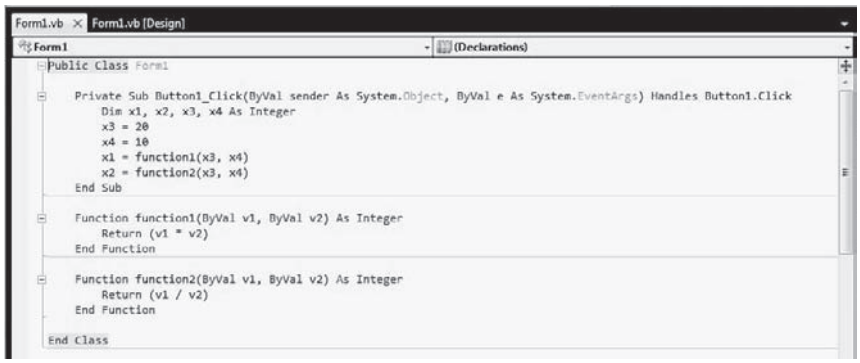


Figure A.18

Code listing for *Button1_Click* routine and two functions

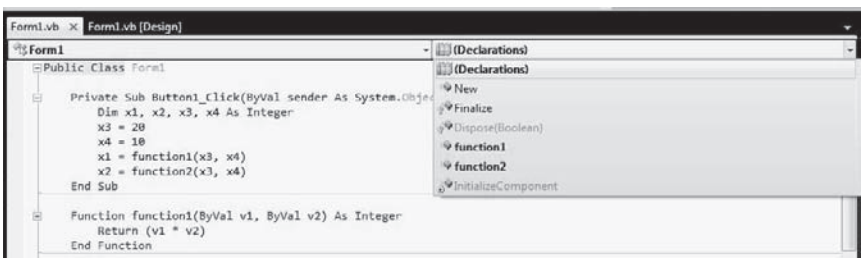


Figure A.19

Right drop-down list

that we developed listed in bold font (see Figure A.19). If we want to go to the listing for any of these functions, we simply select that function from the list.

Similarly, if we selected the *Button1* element from the left list and we expand the right list, we see all of the code elements associated with the *Button1* object (see Figure A.20). We notice that the *click* event is shown in bold, while the rest of the events on the list are not bold. An entry in bold font means that there is code associated with that entry, which is the case with the *click* event.

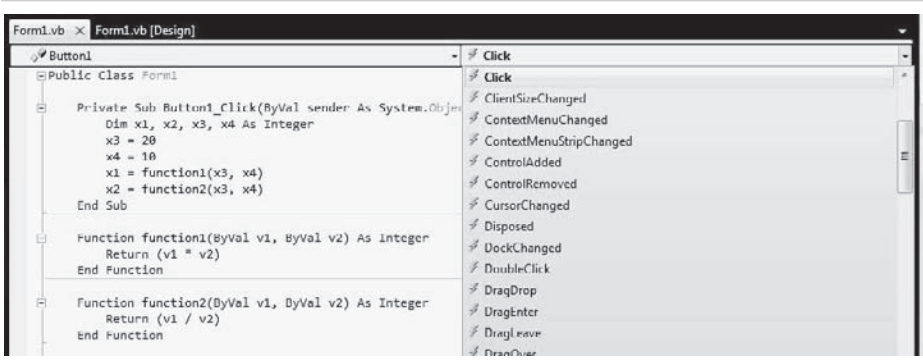


Figure A.20

Right drop-down list for *Button1* object

A.9 Objects and Classes

As mentioned before, all of the control elements in VBE are objects, and almost everything that is done in VBE is associated with objects. An **object** is an entity or a structure that contains both variables (and data) and procedures (or methods) that

operate on these variables. While classes and objects are used interchangeably, they mean different things. Microsoft defines **a class** as an abstract designation of something while an object is a realization of the thing that the class refers to. For example, we can create a class named *Identity* that contains data such as *name* and *age*, and procedures to operate on these data elements. An example VBE code for such a class is shown in Figure A.21. This class has two data members, *name* and *age;* a procedure named *setage;* and a property named *agevalue*. This class becomes an object when we create an instance of this class.

Figure A.21

Code for creating a class

```
Private Class Identity
        Public name As String
        Private age As Byte

        Public Sub setage (ByVal num As Byte) ' Setting age using a sub
            age = num
        End Sub

        Public Property agevalue () As Byte'setting/reading age using a
                                            property
            Get
                agevalue = age
            End Get
            Set (ByVal value As Byte)
                age = value
            End Set
        End Property
    End Class
```

Creating an object from a user-defined or VBE-defined class is done using the *New* keyword. For this example, an object of type *Identity* is created with the statement:

```
Dim citizen As Identity
citizen = New Identity
```

In this example, *Identity* is the class, and *citizen* is an object of that class. There is no limit on how many objects we can create from the same class. As an alternative, we can declare and create an object using a single statement:

```
Dim citizen As New Identity
```

To access the public members of the class, the dot operator is used. For example, if we want to specify the name of the *citizen* object to be Joe, this can done using the statement:

```
citizen.name="Joe"
```

Note that because the *age* data member is defined as private in Figure A.21, it cannot be accessed using the dot operator as was the case for the *name* data member. The listing in Figure A.21 provides two different ways of setting the age: one using the procedure *setage* and the other using the property *agevalue*. Using the procedure *setgage*, the *age* is set with the statement:

```
citizen.setage (21)
```

Or using the property *agevalue*, the *age* is set using

```
citizen.agevalue = 21
```

and is read using

```
Age1 = citizen.agevalue
```

where *Age1* is a variable. A special keyword in VBE is the ***ME* keyword**. This keyword can be conveniently used to refer to the current instance of a class or data structure. Thus, instead of passing the name of the current object to a function or sub-procedure, the *ME* keyword can be used.

VBE allows us to create a new class (called a ***derived* class**) from an already existing class called the *base* class. This process is called inheritance. Any class can be used as the base class, unless the keyword *NotInheritable* was used in creating the base class. However, the derived class can inherit from only one base class. Inheritance allows the creation of reusable code, where one class is based on another class. VBE is supplied with thousands of classes that one can use as a base for the derived classes.

As an example of creating a derived class, let us consider creating a class called *Information* that uses the *Identity* class we created before as the base class. The code listing for creating the *Information* class is shown in Figure A.22.

```
Private Class Information
        Inherits Identity
        Public Telphone As String
        Public Occupation As String
End Class
```

Figure A.22

Code listing for the derived class *Information*

As seen in Figure A.22, in addition to the *Identity* class, we added two new members (*telephone* and *occupation)* to the *Information* class. Assume we have created an object of this class called *record*. All members of the base class are now members of the derived class, and we access them as if we are dealing with the base class. For example, the *name* member of the base class can be accessed in the derived object *record* as

```
record.name = "Jack"
```

A.10 Error Handling

The Integrated Development Environment (IDE) of VBE does a good job of identifying syntax errors during the development of computer programs, but means should be included in the code to handle errors that occur during the execution of a program. These errors are called *exceptions* in VBE and could occur due to a fault in the code or unexpected behavior during program execution. A convenient way to do this is to use the ***Try/Catch*** statement to handle errors. Code that is prone to generate error (such as the possibility of dividing by zero) is included in the *Try* statement. The *Catch* part of the code has statements to handle the error (such as providing a message to inform the user what causes the error). If not included, a fatal error could occur which causes the program to crash. Figure A.23 shows an example of implementing a *Try/Catch* statement to check for the possibility of dividing by zero.

Since *v2* is set to zero here, an exception will be generated when evaluating *v1*. The exception will cause the program to display the message "dividing by zero"

Figure A.23

Try-Catch method of error handling

```
Dim v1, v2 As Integer
v2 = 0

Try
    v1 = 5 / v2
Catch
    MsgBox("dividing by zero")
End Try
```

that is sent to the screen using the *MsgBox* function. Implement the above code for evaluating *v1* without using a *Try/Catch* structure, and notice what happens when you execute the code.

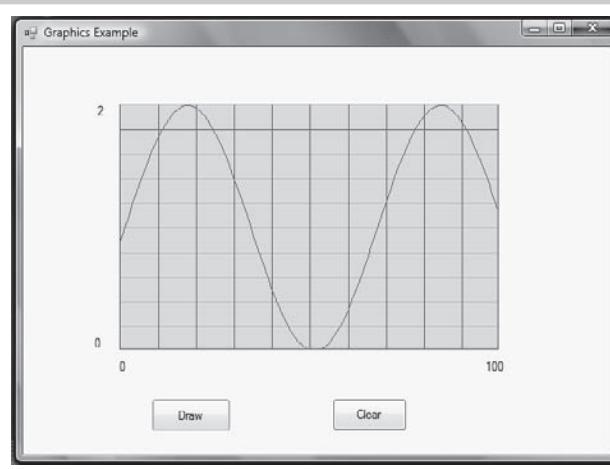
A.11 Graphics Programming

In many situations, it is desirable to graphically display a certain variable or an input data since the data trend can be easily seen in a graphical form. VBE supports the '***panel***' control which can be used to display data. This control creates a rectangular area on the form to which commands to create lines, arcs, etc. can be applied. Note that the location of any point in the panel area is defined by an (*x*,*y*) coordinate system in pixel units that is attached to the top-left corner of the panel with the positive *x*-axis pointing horizontally to the right and the positive *y*-axis pointing vertically downward. Thus, the top-left corner of the panel has a coordinate value of (0,0), while the lower-right corner has a coordinate value of (*Panel*.Width*, *Panel*.Height*), where *Panel*.Width* is the width property of the panel area in pixels, and *Panel*.Height* is the height property of the panel area in pixels. To draw lines in the panel area, one uses the function `DrawLine(pen, x1, y1, x2, y2)` which takes as an input the pen type to use and the start and the end (*x*,*y*) coordinates of the line. In actual usage, this function is called as `Panel*.CreateGraphics.DrawLine()`. To clear the graphics area, one uses the function `Panel*.CreateGraphics.Clear()`. Many other functions are provided in the *CreateGraphics* class to draw items such as arcs, rectangles, and curves.

The data to be plotted is usually specified in engineering units (such as meters or volts). Thus, before plotting, one needs to convert and scale the data so it can fit in the graphics area. The scaling easily can be done (for example, by multiplying the *x*-coordinate of each data point in engineering units by the ratio of the panel width to the range of *x* units). In a similar fashion, the *y*-coordinates can be scaled. Furthermore, to make the plotting easier, one can also transform the origin of the coordinate system (with simple mathematical manipulation) to be at the lower-left corner with the positive *y*-axis pointing upward. Figure A.24 shows an offset sine wave plotted in a panel window with the origin transformed to the lower-left corner.

Figure A.24

Offset sine wave



The gridlines in Figure A.24 are drawn as a series of horizontal and vertical lines, since VBE does not have a built-in function to draw a grid.

In situations where the panel window is covered by another form, the graphic display is erased. In this case, the graphic data needs to be re-drawn. An automatic way of doing this would be to activate the '*paint*' event associated with the panel control. Whenever the graphics display needs to be re-drawn, the *paint* event handler is then automatically called. Obviously, the user needs to provide all of the code that is needed in the event handler to redraw the graphics.

A.12 ToolBox Controls

The ***ToolBox*** which shows up in the *Designer* view in the Windows Forms application lists many types of controls, but only a few of them are needed to make simple applications. The controls are organized in several categories, including *Common Controls*, *Containers*, *Menus & Toolbars*, and *Dialogs*. We have already used the **Button** and the **TextBox** controls in the examples we have covered in this appendix. These controls are listed in the *Common Controls* category. Next, we will discuss additional controls that are useful in many applications. One such control is the ***ToolTip*** control (also included in the *Common Controls* category). When this control is added to a form, it allows the user to enter display information for each control on the form. This information will be displayed when the user moves the pointer over that control which has the associated *ToolTip* information. To illustrate this, assume we have a button control added to our form. We would like to display the message "*Hit This Button to Run the Program*" when the user moves the mouse over this button. If we have added the *ToolTip* control to our form, then we simply type the message in the *ToolTip on ToolTip1* property for the button control. The message will be displayed when the program is run and when the pointer moves over this button (see Figure A.25). Thus, the *ToolTip* control offers a simple and convenient method of adding display and help information to an application.

Figure A.25

Illustration of the *ToolTip* information display

In many applications, we need to select only one choice among several available. This can be handled using the ***Radio Button*** control. Figure A.26 shows three radio button controls labeled *Low*, *Medium*, and *High* placed in a ***GroupBox*** control. While not needed for a single set of radio button controls, the *GroupBox* container control offers a convenient method of separately grouping each set of radio buttons when we have two or more sets of them. When a particular button is pressed, the *Checked* property of that radio button is set to true. The code can check that property to tell if the button is pressed or not.

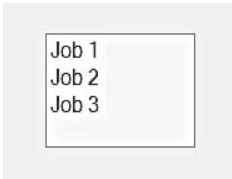
When we need to create a list of items to display or to select from a list of items, a ***ListBox*** control is used. Figure A.27 shows a *ListBox* that displays three items. Items are added to the list by using the *ListBox1.Items.Add("textstring")* method, where *textstring* is the item to be displayed in the list.

Figure A.26

Radio button controls

Figure A.27

ListBox for displaying a list of items



To get an item from a list of items, the *ListBox1.Items(index).ToString()* method is used, where *index* is the item number. For example, to retrieve *Job 3* in the above example, *index* should be set to 3.

A.13 File Input/Output

In many applications, we need the capability to store or retrieve information from a file. VBE offers the ***OpenFileDialog*** and the ***SaveFileDialog*** controls to open and save a file, respectively. The procedure *Save_Data()* in Figure A.28 shows a typical code for saving data to a file using the *SaveFileDialog* component. The resulting dialog when the *Save_Data()* procedure is called is shown in Figure A.29.

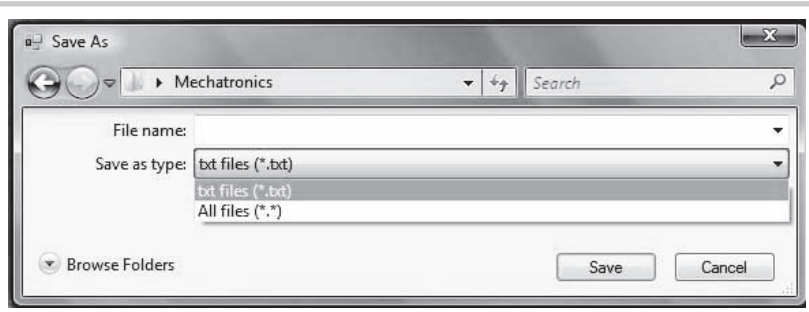
Figure A.28

Example code for saving data to a file

```
Private Sub Save_Data()
    Dim Npoints As Long
    Dim Fname$
    Dim w As StreamWriter
    Npoints = 100
    Dim saveFileDialog1 As New SaveFileDialog

    saveFileDialog1.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*"
    saveFileDialog1.FilterIndex = 1
    saveFileDialog1.RestoreDirectory = True

    If saveFileDialog1.ShowDialog() = DialogResult.OK Then
        Fname$ = saveFileDialog1.FileName

        Dim fs As New FileStream(Fname$, FileMode.OpenOrCreate)
        w = New StreamWriter(fs)
        ' Write data to file
        Dim i As Long

        For i = 1 To Npoints
            w.WriteLine(Str$(xdata(i)))
        Next i
        w.Close()
        fs.Close()
    End If
End Sub
```

Figure A.29

Save dialog interface corresponding to code shown in Figure A.28



In using these dialogs, the user sets the file extension type (such as .txt) of the files to open or save. This process is called *filtering*, as it allows the user to select which types of files in the current directory will appear in the dialog. In

Figure A.29, the "*Save as type*" drop-down list is expanded to show the file types that are displayed. In the example code, we included two file types: *txt files* and *All files*. Since the *FilterIndex* property is set to 1, the *txt files* choice is first displayed, since it is the first in the list of file types. The *RestoreDirectory* property of the *SaveFileDialog* restores the current directory after the dialog closes if it was set to true.

When the user clicks on the *Save* button in the dialog, the result of the dialog is OK, and the code proceeds to save the data to the user-specified or selected filename. This example stores 100 elements of the array *xdata*, which is assumed to be defined somewhere else in the code, to a file. Note that to read or write from a file, the *stream* method is used, which requires that the *System.Io* namespace be included in the VBE form file. This is done by including the statement `Imports System.IO` at the top of the *Form1.vb* file before the *Public Class Form1* statement.

System Response

Many physical systems can be represented by either a first-order or a second-order differential equation model. Examples of **first-order systems** include the model of the speed of a mass subjected to a force input, the model of the height of fluid in a tank with flow input, or the model of the temperature of a heated plate. Examples of **second-order systems** include the model of the position of a mass subjected to a force input and the model of the height of fluid in a coupled two-tank system. We will consider in this appendix both the time and frequency response of first- and second-order systems.

B.1 Time Response of First-Order Systems

The **time response** of a system is the output of the system as a function of time when subjected to an input signal (such as a step or a ramp). In general, the response of a system is the sum of the free response plus the forced response. The **free response** is the response of the system due to initial conditions, while the **forced response** is the response due to an external input. As an example, let us assume that our first-order system is represented by the differential equation:

$$m\dot{v} + bv = f(t) \tag{B.1}$$

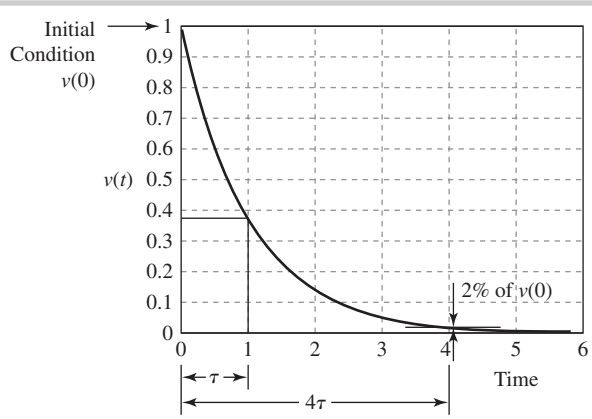
For positive m and b, the response of the system due to an initial condition $v(0)$ and a step input force of magnitude F is given by

$$v(t) = v(0)e^{-bt/m} + \frac{F}{b}(1 - e^{-bt/m}) \tag{B.2}$$

The first term in Equation (B.2) is the free response while the second term is the forced response. The free and the forced responses are shown in Figures B.1 and B.2, respectively.

Figure B.1

Free response of model given by Equation (B.1)



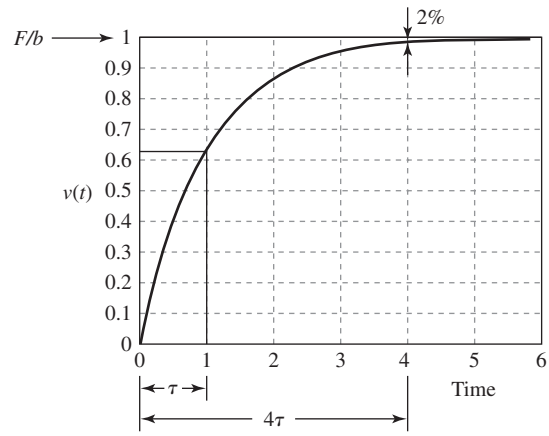


Figure B.2

Forced response of model given by Equation (B.1)

As seen in Figure B.1, the free response takes the form of an exponential decay with a **time constant** $\tau = m/b$. The response v starts at the initial condition $v(0)$, and after four time constants, the free response is only about 2% of the initial value. As time increases further, the free response goes to zero. The forced response starts at zero, and increases in an exponential fashion. After four time constants, the output is within 2% of the final steady value. From this, we see that for a first-order system the time constant of the system solely determines the responsiveness of the system. Systems that have a small time constant (such as DC motor-driven systems) have a fast response, while those that have a large time constant (such as thermal systems) have a slow response.

B.2 Time Response of Second-Order Systems

Similar to a first-order system, a second-order system response has free and forced components. The details of the response are dependent on the roots of the characteristic equation of the system model. While the following differential equation represents the dynamics of a mass, spring, and damper system subjected to a force input, any second-order system can be represented by such a model:

$$m\ddot{x} + c\dot{x} + kx = f(t) \tag{B.3}$$

For this model, the transfer function between $F(s)$ and $X(s)$ is given as

$$\frac{X(s)}{F(s)} = \frac{1}{ms^2 + cs + k} \tag{B.4}$$

and the **characteristic equation** is

$$ms^2 + cs + k = 0 \tag{B.5}$$

Equation (B.5) has two roots. These are

$$s_{1,2} = \frac{-c \pm \sqrt{c^2 - 4mk}}{2m} \tag{B.6}$$

The **roots** can be real or imaginary, depending on the sign of the quantity under the square root. They also can have positive or negative real parts. When the

root has a positive real part, the response of the system is unbounded or unstable, while a non-zero imaginary part means that the response of the system will be oscillatory. For the case of stable response (the roots lie in the negative half plane), we can characterize the type of response by determining the damping ratio of the characteristic equation. To do this, we write the characteristic Equation (B.5) in the form shown in Equation (B.7). We let $k/m = \omega_n^2$, and $c/m = 2\xi\omega_n$, where ω_n is defined as the **natural frequency** and ξ is the damping ratio. The **damping ratio** is defined as the ratio of c/c_c, where c_c is the critical damping, which is the value of the damping that causes the roots of the characteristic Equation (B.5) to have two repeated real roots or the term under the square root in Equation (B.6) to be zero:

$$s^2 + 2\xi\omega_n s + \omega_n^2 = 0 \tag{B.7}$$

Using the damping ratio ξ to characterize the response, we can have three cases.

Underdamped Case ($0 < \zeta < 1$) In this case the roots will have an imaginary part and the free response or the step-input response will be oscillatory. Figure B.3 shows the free response, and Figure B.4 shows the forced step response, respectively, for various values of ξ. The plots are shown for the case $\omega_n = 1$ rad/s, $m = 1$ kg, and $F = 1$ N.

Figure B.3

Free response of a second-order system for various values of ζ

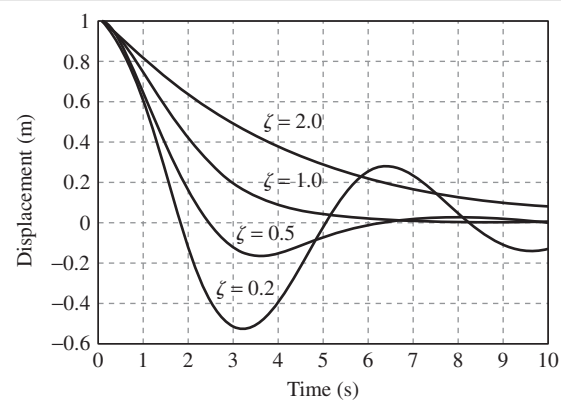
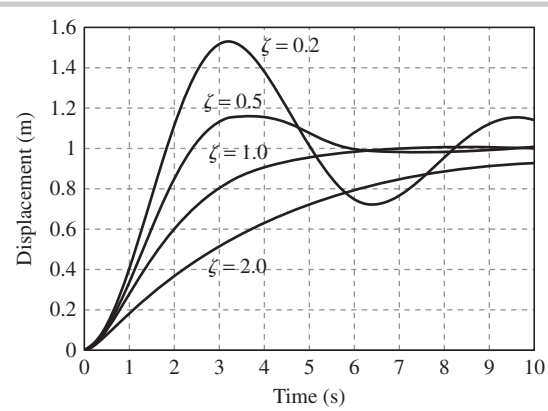


Figure B.4

Forced response of a second-order system under a unit step input for various values of ζ



Critically Damped Case ($\zeta = 1$) The roots in this case will be two repeated roots with no imaginary component. The response will not be oscillatory.

Overdamped Case ($\zeta > 1$) The roots will also be real in this case, but not repeated. The response will not be oscillatory and will be slower than the critically damped case.

Note that when the damping ratio is zero, the roots are purely imaginary and have no real component. The free response in this case will be oscillatory but will not decay with time.

Figure B.5 gives a graphical interpretation of the location of the root of a second-order system. For $0 < \zeta < 1$, the root has real and imaginary components. The vector from the origin to the root location has a length of ω_n and makes an angle of θ with the negative real axis. The real component magnitude is given by $\xi\omega_n$. Note that $\zeta = \cos(\theta)$. Thus, when ζ is 1, θ is zero, the root has no imaginary component; while when $\zeta = 0$, θ is 90°, the root has no real component. Note that the real part of the root is equal to $1/\tau$, where τ is the time constant. Thus, the further the root is away from the imaginary axis, the smaller the time constant is, and the faster the response would be. The imaginary component magnitude is equal to $\omega_d = \omega_n\sqrt{1 - \zeta^2}$, where ω_d is the **damped natural frequency**. The damped natural frequency ω_d is smaller than the natural frequency ω_n. Note that the further away the root is from the real axis, the higher the frequency of oscillation will be.

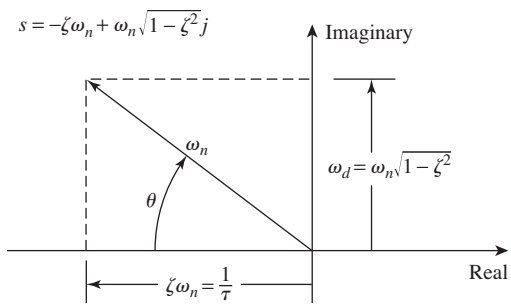
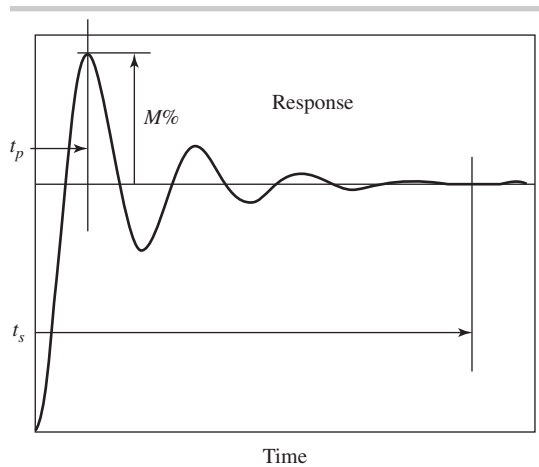


Figure B.5

Graphical interpretation of root location

The parameters of a second-order model can be obtained from analyzing the response plot of the system. Figure B.6 shows a typical plot and a few of the performance characteristics of the plot. These include the **percentage overshoot**, the **peak time**, and a **2% settling time**. These parameters are defined in the right side of Figure B.6.



Maximum percent overshoot:

$$M\% = 100e^{-\pi\zeta/\sqrt{1-\zeta^2}}$$

$$A = \ln\frac{100}{M\%}, \zeta = \frac{A}{\sqrt{\pi^2 + A^2}}$$

Peak time:

$$t_p = \frac{\pi}{\omega\sqrt{1 - \zeta^2}}$$

Settling time (2%):

$$t_s = \frac{4}{\zeta\omega_n}$$

Figure B.6

Performance characteristics of a second-order system

B.3 Frequency Response

The **frequency response** of a system is the response of the system to a periodic input signal (such as a sinusoidal input as a function of the frequency of the sinusoid). The frequency response is normally shown as two plots: a magnitude plot and a phase plot. In the **magnitude plot**, the ratio of the output of the input is plotted as a function of frequency. Normally, the magnitude plot is displayed in units of **dB** where 1 dB = 20 log (output to the input). In the **phase plot**, the angle between the output and the input signals is displayed as a function of frequency. To illustrate the process of generating the frequency response plots, let us consider the transfer function of a low-pass filter, which is given by $G(s) = 1/(1 + \tau s)$ (see Section 7.10). We replace the variable s in the transfer function with $j\omega$. Performing this, we obtain

$$G(j\omega) = \frac{1}{1 + j\omega\tau} \tag{B.8}$$

To get rid of the complex quantity in the denominator, we multiply the numerator and the denominator of Equation (B.8) by the complex conjugate of $1 + j\omega\tau$, which is $1 - j\omega\tau$. This gives

$$G(j\omega) = \left(\frac{1}{1 + j\omega\tau}\right)\frac{1 - j\omega\tau}{1 - j\omega\tau} = \frac{1 - j\omega\tau}{1 + \omega^2\tau^2} = \frac{1}{1 + \omega^2\tau^2} - \frac{j\omega\tau}{1 + \omega^2\tau^2} \tag{B.9}$$

The **magnitude** of the transfer function is then obtained by taking the square root of the sum of the squares of the real and imaginary parts in Equation (B.9). This gives

$$|M(j\omega)| = \sqrt{\left(\frac{1}{1 + \omega^2\tau^2}\right)^2 + \left(\frac{\omega\tau}{1 + \omega^2\tau^2}\right)^2} = \frac{1}{\sqrt{1 + \omega^2\tau^2}} \tag{B.10}$$

The **phase angle** is obtained from evaluating the inverse tangent of the ratio of the imaginary part to the real part or

$$\varphi = \tan^{-1}\left(\frac{-\tau\omega}{1}\right) = -\tan^{-1}(\tau\omega) \tag{B.11}$$

Equations (B.10) and (B.11) can be plotted as functions of the angular frequency ω to obtain the frequency response plot of the transfer function. These plots are given in Figures B.7 and B.8, respectively, where the magnitude plot is displayed in dB units.

From Equation (B.10), we see that when $\omega\tau$ is very small; the magnitude is approximately equal to 1. Since the logarithm of 1 is zero, then the magnitude in dB units is equal to 0 for small values of $\omega\tau$. Similarly, when $\omega\tau$ is very large, Equation (B.10) can be approximated by $(\omega\tau)^{-1}$. Hence, the magnitude in dB is

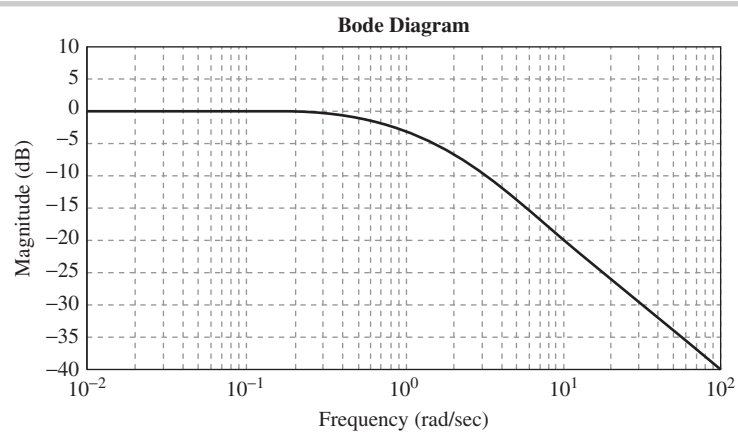


Figure B.7
Magnitude plot of a first-order system

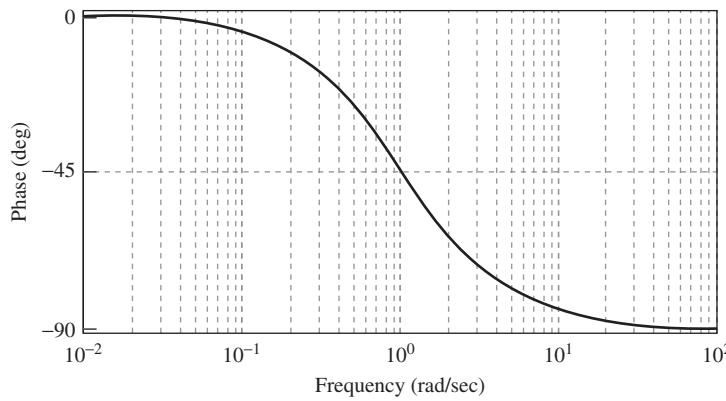


Figure B.8

Phase plot of a first-order system

equal to $-20 \log \omega - 20 \log \tau$. Thus, the slope of the frequency magnitude plot is -20 dB per decade, where a decade is a 10 times increase or decrease in the frequency. When $\omega = 1/\tau$, or is at the **corner frequency**, the magnitude has a value of -3 dB. Similarly for the phase plot, we can see from Equation (B.11) that when $\omega\tau$ is very small, its inverse tangent is close to 0°, while when $\omega\tau$ is very large, its inverse tangent is close to $-90°$. A **negative phase angle** means that the output signal lags the input signal, while a **positive phase angle** means that the output signal leads the input signal. The relationship between the phase angle (φ in degrees) and the **lead/lag time** (Δt in seconds) at any particular frequency (ω in rad/s) is given by

$$\Delta t = \frac{\varphi \pi}{180\omega} \tag{B.12}$$

As an example, at a frequency of 10 rad/s, the output signal from the low-pass filter will lag the input signal by 0.147 seconds.

MATLAB has two commands for generating the frequency response plots. They are the ***Bode*** and ***Bodemag***. The *Bode* command generates both the magnitude and phase plots, while the *Bodemag* generates the magnitude plot. Figure B.9 shows

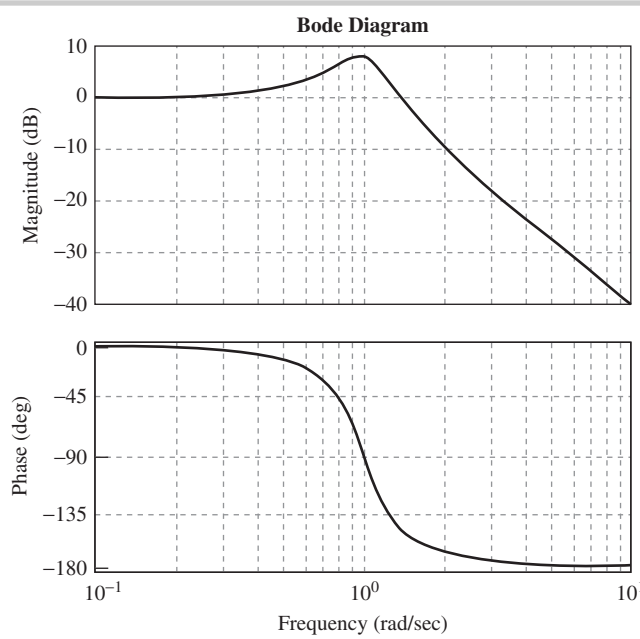


Figure B.9

Frequency response plot for an underdamped second-order system

the Bode plot for the second-order system given by Equation (B.3). Notice the hump in the magnitude plot, which is due to the fact that the system is underdamped. For this second-order system, the magnitude decreases at the rate of 40 dB per decade after the corner frequency (versus 20 dB per decade for a first-order system), and the phase plot approaches $-180°$ for large frequencies.

One property of linear systems is that the frequency response of the product of several transfer functions is the same at that obtained by summing the frequency response of each individual transfer function. This follows from the fact that $\log(ab) = \log(a) + \log(b)$.

APPENDIX C

MATLAB Simulation of Dynamic Systems

MATLAB is a software package used for modeling and simulating a variety of linear and nonlinear systems in both the continuous and discrete domains. The program is widely used in engineering programs throughout the world, so most of the readers of this textbook should be familiar with it. We will give in this appendix only a brief overview of how one can use MATLAB to obtain a solution for a dynamic model, and readers should consult any of the many available textbooks on this subject for further reading, see [42-43]. We also discuss block diagram representation and simulation in MATLAB.

C.1 Solution of Differential Equations in MATLAB

MATLAB offers several ways to obtain the solution for the set of differential equations that are obtained when a model of a dynamic system is obtained. The particular method depends on the characteristics of the system. The three common methods are listed here.

- State space solution methods for linear differential equation systems
- Direct integration using ODE solvers for nonlinear differential equation systems
- Transfer function methods for linear differential equation systems with zero initial condition

A brief outline of each method follows.

C.1.1 State-Space Solution Method

This method is applicable to linear differential equation systems. The obtained differential equation(s) are represented as a set of n first-order differential equations in the form

$$\dot{x} = Ax + Bu$$

and the input output relationship is represented as

$$y = Cx + Du$$

where x is the $n \times 1$ state space vector, u is the $m \times 1$ input vector, A is the $n \times n$ model coefficient matrix, B is the $n \times m$ input coefficients matrix, y is the $q \times 1$ output vector, C is the $q \times n$ output matrix, and D is the $q \times m$ output-input matrix. For single-input, single-output systems, $m = q = 1$. With the A, B, C, and D matrices specified, the **state-space model** is created with the MATLAB command

```
sys1 = ss(A,B,C,D)
```

where *sys1* is a user-defined name for the system. The response of the system then can be obtained in several ways. If we are interested in obtaining the response of the system to a predefined input signal (such as a step signal), then the ***step*** command can be used with the typical calling format:

```
step(sys1)
```

If we would like to obtain the response for a user-defined input vector *uv*, then the ***lsim*** command be used with the calling format:

```
lsim(sys1, uv, t)
```

where *uv* is a user-defined input vector specified over the time interval *t*.

As an illustration of this method, let us consider the differential equation model for a motor-driven geared system. The equation is

$$\tau_m = I_{\text{eff}}\ddot{\theta}_1 + b_{\text{eff}}\dot{\theta}_1 \tag{C.1}$$

This is a second-order linear differential equation, and thus, we need two state variables to represent this model. Let x_1 be the angular position θ_1 and x_2 be the angular velocity $\dot{\theta}_1$; then the following two first-order differential equations are equivalent to Equation (C.1).

$$\dot{x}_1 = x_2 \tag{C.2}$$

and

$$\dot{x}_2 = \frac{1}{I_{\text{eff}}}(\tau_m - b_{\text{eff}}x_2) \tag{C.3}$$

If the output of the system is the angular position θ_1, then the state-space matrices for this system are

$$A = \begin{bmatrix} 0 & 1 \\ 0 & \dfrac{-b_{\text{eff}}}{I_{\text{eff}}} \end{bmatrix}, B = \begin{bmatrix} 0 \\ \dfrac{1}{I_{\text{eff}}} \end{bmatrix}, C = [1\ 0], D = [0] \tag{C.4}$$

Letting $I_{\text{eff}} = 0.1\ \text{kg} \cdot \text{m}^2$, and $b_{\text{eff}} = 0.2\ \text{N} \cdot \text{m/sec}$, and $\tau_m = 1\ \text{N} \cdot \text{m}$, Figure C.1(a) shows the unit step response obtained by using the command

```
step(sys1,2)
```

for the state-space system defined by Equation (C.4), where 2 is the time duration of the simulation. Figure C.1(b) shows the corresponding step response when the angular speed was made as the output of the system (*C* in this case is equal to [0 1]).

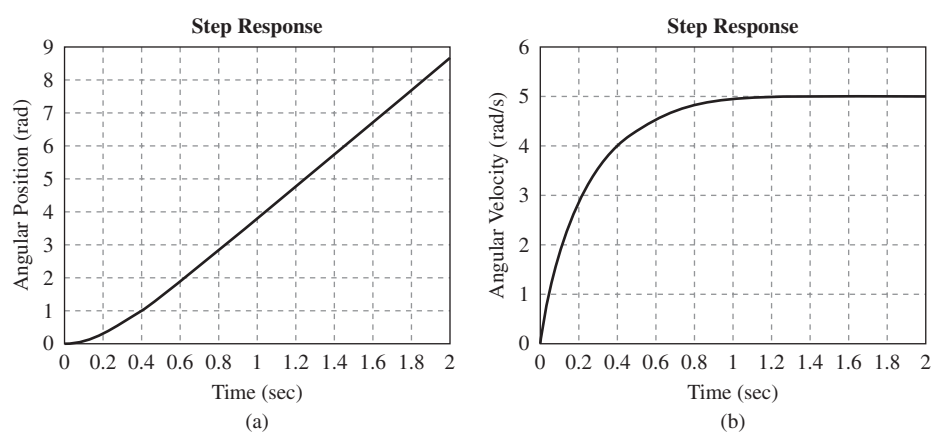


Figure C.1

Step response for system defined by Equation (C.4) (a) angular position response and (b) angular velocity response

C.1.2 Direct Integration Using ODE Solvers

For nonlinear differential equations, we can obtain the solution by direct integration using any of the ODE solvers in MATLAB. The user needs to define the differential equation model, the time span to perform the integration, and the initial conditions before the ODE solver is called. The differential equation model is defined using the *function* script in MATLAB. As an illustration of this process, let us consider the following nonlinear differential equation (Equation (C.5)) which represents the motion of a pendulum.

$$I_o \ddot{\theta} + k \sin\theta = 0 \tag{C.5}$$

This differential equation is represented in the MATLAB function $f(t,y)$ listed in Figure C.2.

```
function dydt = f(t,y)
dydt = [y(2); -k/IO * sin(y(1))];
end
```

Figure C.2
MATLAB function for Equation (C.5)

The function $f(t,y)$ is dependent on time and the vector y. The angular position of θ is $y(1)$ and the angular velocity $\dot{\theta}$ is $y(2)$. Note the output argument *dydt* defines a 2×1 vector, which gives expressions for the derivatives of y, i.e.,

$$\begin{bmatrix} \dot{y}(1) \\ \dot{y}(2) \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} y(2) \\ -\,{}^{k}/_{10} \sin(y(1)) \end{bmatrix} \tag{C.6}$$

To integrate this problem, the following statements are typed in MATLAB:

```
tspan = [0, 2.5];
y0 = [pi()/4; 0];
[t,y] = ode45(@f, tspan, y0);
```

where $y0$ is the initial-condition vector, and *ode45* is a differential equations solver based on the use of an explicit Runge-Kutta formula. The *ode45* solver is used to solve ordinary differential equations with high accuracy. MATLAB has other differential equations solvers such as *ode23*, but *ode45* is the preferred solver for most problems, since it offers higher accuracy than *ode23*. It should be noted that *ode45* is the default solver in Simulink.

Figure C.3 shows the solution of Equation (C.5) for the following set of parameters:

$I_O = 0.775\ \text{kg} \cdot \text{m}^2$ and $k = 15.696\ \text{N} \cdot \text{m/rad}$

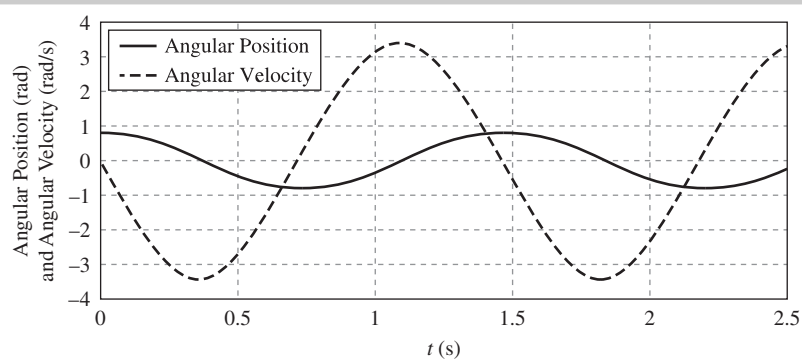


Figure C.3
Pendulum simulation

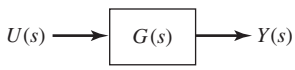
where the initial condition $\theta_0 = \pi/4$ and $\dot{\theta}_0 = 0$. Note that due to the lack of friction, the pendulum keeps oscillating between $\pi/4$ and $-\pi/4$. Note also how the pendulum angular velocity is maximum when the pendulum goes through the zero position and is zero at the extremes of motion.

C.1.3 Transfer Function Methods

For linear systems with zero initial conditions, the differential equation model can be transformed into transfer function form using the Laplace transform, and the transfer function commands in MATLAB can then be used to obtain a solution. The transfer function (see Section 9.4 for more details) of a system is defined as the ratio of the output to the input of the system under zero initial conditions. It is expressed as the ratio of two polynomials in the parameter s. Thus, the transfer function $G(s)$ between the input signal U and the output signal Y is written as

Figure C.4

A block diagram representation of a transfer function



$$G(s) = \frac{Y(s)}{U(s)} = \frac{b_0 s^m + b_1 s^{m-1} + \cdots + b_m}{a_0 s^n + a_1 s^{n-1} + \cdots + a_n} \tag{C.7}$$

where $m \leq n$. The transfer function relationship can be graphically represented using block diagram concepts. A block diagram of the transfer function given by Equation (C.7) is shown in Figure C.4. In a block diagram, input and output are indicated by the direction in which the arrows point.

The Laplace transform of a function $f(t)$ is defined as

$$F(s) = L(f(t)) = \int_0^\infty f(t)e^{-st}dt \tag{C.8}$$

where s is a complex number. This definition can be used to obtain the Laplace transform of given time functions. For example, the Laplace transform of the unit step function $u(t)$ is obtained as

$$F(s) = \int_0^\infty f(t)e^{-st}\,dt = \int_0^\infty 1e^{-st}\,dt = \left[-\frac{1}{s}e^{-st}\right]_0^\infty = \frac{1}{s} \tag{C.9}$$

The Laplace transform is usually covered in detail in textbooks on system dynamics (see [38]) or feedback control, so the details of it will not be covered here. We will only point out certain aspects of it as it relates to converting differential equations into algebraic equations. One such aspect is the **derivative property**, which gives an expression for the Laplace transform of the time derivative of a function of time. For the first time derivative, the property is

$$L\left(\frac{df(t)}{dt}\right) = s\,L(f(t)) - f(0) = s\,F(s) - f(0) \tag{C.10}$$

and for the second time derivative, the property is

$$L\left(\frac{d^2f(t)}{dt^2}\right) = s^2\,F(s) - s\,f(0) - \dot{f}(0) \tag{C.11}$$

As an illustration of this method, let us consider again the differential equation obtained for the dynamics of a motor-driven geared system. Applying the Laplace transform to Equation (C.1), and using Equation (C.11) with the assumption of zero initial conditions, we obtain

$$\tau_m(s) = I_{\text{eff}} s^2 \theta_1(s) + b_{\text{eff}}\, s\theta_1(s) \tag{C.12}$$

This equation can be written in transfer function form as

$$\frac{\theta_1(s)}{\tau_m(s)} = \frac{1}{I_{\text{eff}}\, s^2 + b_{\text{eff}}\, s} \tag{C.13}$$

The transfer function in Equation (C.13) can be defined in MATLAB using the ***Transfer Function* command** or *TF*(*NUM*, *DEN*) function, which creates a continuous time transfer function. *NUM* and *DEN* are row vectors that specify the numerator and denominator coefficients, respectively, of the transfer function in a descending order. Thus, the transfer function in Equation (C.13) is specified with the command

G1 = tf([1],[I_{eff} b_{eff} 0])

If we are interested in obtaining the response of the system to a predefined input signal (such as a step signal), then the *step* command can be used with the typical calling format

step(G1)

While multi-input, multi-output systems can be represented in transfer function form, the transfer function method is better suited for single-input, single-output systems.

C.2 Block Diagram Representation and Simulation in MATLAB

A block diagram representation of a dynamic system offers a graphical representation of the interaction of the different elements in the system. MATLAB offers (through the Simulink package) a mechanism for performing this representation in software. Simulink is an add-on to MATLAB and cannot run unless MATLAB is installed.

Simulink offers a library of many predefined elements (called blocks in software) that one can choose from (see Figure C.5). These blocks are grouped in different categories. For simulation of dynamic systems in the continuous-time domain, blocks from the *Continuous* category are used. This category includes a transfer-function block, a state-space block, a derivative block, an integrator block, and different types of time-delay blocks.

To build a dynamic model using Simulink, the user selects and drops the needed blocks into the Simulink model sheet. The blocks are then joined together

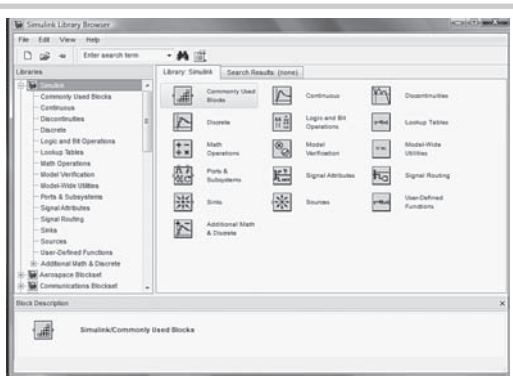


Figure C.5

Simulink block categories

to represent the desired signal flow between the blocks. Before the simulation is started, the **block parameters** for each block need to be defined so that the Simulink model corresponds to the dynamic system that is modeled. The block parameters can be either set to numeric values or left as variables that need to be defined in an *.m* file that is run before the simulation is started.

To illustrate the use of Simulink, let us build a model of the dynamics of the motor-driven geared system considered previously. The transfer function of this system was given in Equation (C.13). As seen in Figure C.6, we represent this transfer function in Simulink using two cascaded blocks: a *Transfer Function* block and an *Integrator* block. While we could have used a single transfer-function block, we choose this representation so we can access the velocity of the system, which is obtained from the output of the transfer-function block. Also, the given transfer function allows this representation because we can factor out an s term from the denominator. The model parameters I_{eff} and B_{eff} are left in variable form here. The values of these parameters need to be defined either in the *Command Window* in MATLAB or by running an *.m* file that defines these variables before the simulation is started, otherwise the model will not run. The Simulink model library includes many blocks to supply inputs. We have chosen the *Step* block here, which supplies a step torque input to the system. To capture the speed and position response of this model, two *Scope* blocks are included in the diagram. Once the model is built and the model parameters are defined, the simulation is started by pressing the arrow button on the menu list.

The speed and position response with $I_{\text{eff}} = 0.1$ and $B_{\text{eff}} = 0.2$ are shown in Figure C.7.

Figure C.6

Simulink representation of the model given by Equation (C.13)

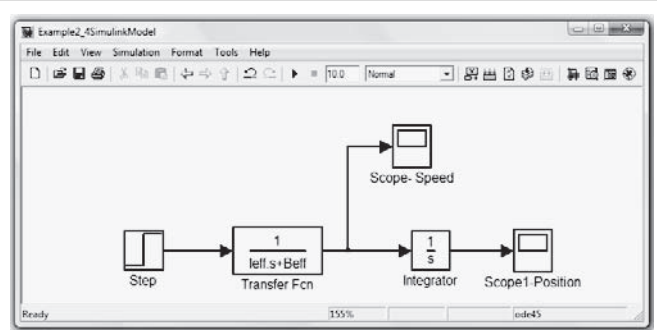
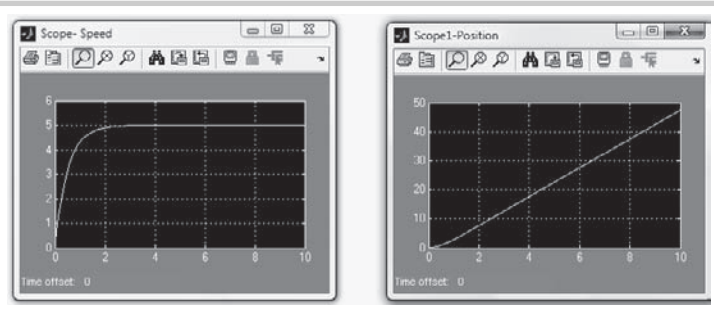


Figure C.7

Speed and position response for model in Figure C.6



7-Bit ASCII Code

ASCII codes

Dec	Hex	Code	Dec	Hex	Code	Dec	Hex	Code	Dec	Hex	Code
0	00	NUL	32	20	space	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	&	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	$	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	–	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

This page intentionally left blank

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z